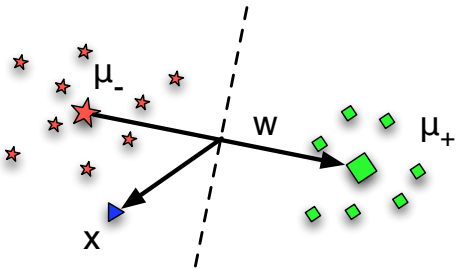

Fig. 1.20. A trivial classifier. Classification is carried out in accordance to which of the two means μ_- or μ_+ is closer to the test point x. Note that the sets of positive and negative labels respectively form a half space.

1.3.3 A Simple Classifier

We can use geometry to design another simple classification algorithm [SS02] for our problem. For simplicity we assume that the observations $x \in \mathbb{R}^d$, such as the bag-of-words representation of e-mails. We define the means μ_+ and μ_- to correspond to the classes $y \in \{\pm 1\}$ via

$$\mu_- := \frac{1}{m_-} \sum_{y_i=-1} x_i \text{ and } \mu_+ := \frac{1}{m_+} \sum_{y_i=1} x_i.$$

Here we used m_- and m_+ to denote the number of observations with label $y_i = -1$ and $y_i = +1$ respectively. An even simpler approach than using the nearest neighbor classifier would be to use the class label which corresponds to the mean closest to a new query x, as described in Figure 1.20.

For Euclidean distances we have

$$\|\mu_- - x\|^2 = \|\mu_-\|^2 + \|x\|^2 - 2\langle \mu_-, x\rangle \text{ and} \tag{1.19}$$

$$\|\mu_+ - x\|^2 = \|\mu_+\|^2 + \|x\|^2 - 2\langle \mu_+, x\rangle . \tag{1.20}$$

Here $\langle \cdot, \cdot \rangle$ denotes the standard dot product between vectors. Taking differences between the two distances yields

$$f(x) := \|\mu_+ - x\|^2 - \|\mu_- - x\|^2 = 2\langle \mu_- - \mu_+, x\rangle + \|\mu_-\|^2 - \|\mu_+\|^2 . \tag{1.21}$$

This is a *linear* function in x and its sign corresponds to the labels we estimate for x. Our algorithm sports an important property: The classification rule can be expressed via dot products. This follows from

$$\|\mu_+\|^2 = \langle \mu_+, \mu_+\rangle = m_+^{-2} \sum_{y_i=y_j=1} \langle x_i, x_j\rangle \text{ and } \langle \mu_+, x\rangle = m_+^{-1} \sum_{y_i=1} \langle x_i, x\rangle .$$

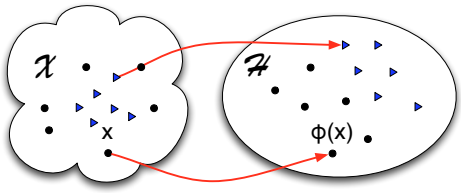

Fig. 1.21. The feature map ϕ maps observations x from $\mathcal{X}$ into a feature space $\mathcal{H}$. The map ϕ is a convenient way of encoding pre-processing steps systematically.

Analogous expressions can be computed for μ_-. Consequently we may express the classification rule (1.21) as

$$f(x) = \sum_{i=1}^{m} \alpha_i \langle x_i, x \rangle + b \tag{1.22}$$

where $b = m_-^{-2} \sum_{y_i=y_j=-1} \langle x_i, x_j \rangle - m_+^{-2} \sum_{y_i=y_j=1} \langle x_i, x_j \rangle$ and $\alpha_i = y_i/m_{y_i}$.

This offers a number of interesting extensions. Recall that when dealing with documents we needed to perform pre-processing to map e-mails into a vector space. In general, we may pick arbitrary maps $\phi : \mathcal{X} \to \mathcal{H}$ mapping the space of observations into a *feature space* $\mathcal{H}$, as long as the latter is endowed with a dot product (see Figure 1.21). This means that instead of dealing with $\langle x, x' \rangle$ we will be dealing with $\langle \phi(x), \phi(x') \rangle$.

As we will see in Chapter 6, whenever $\mathcal{H}$ is a so-called Reproducing Kernel Hilbert Space, the inner product can be abbreviated in the form of a kernel function $k(x, x')$ which satisfies

$$k(x, x') := \left\langle \phi(x), \phi(x') \right\rangle . \tag{1.23}$$

This small modification leads to a number of very powerful algorithm and it is at the foundation of an area of research called kernel methods. We will encounter a number of such algorithms for regression, classification, segmentation, and density estimation over the course of the book. Examples of suitable k are the polynomial kernel $k(x, x') = \langle x, x' \rangle^d$ for $d \in \mathbb{N}$ and the Gaussian RBF kernel $k(x, x') = e^{-\gamma \|x - x'\|^2}$ for $\gamma > 0$.

The upshot of (1.23) is that our basic algorithm can be *kernelized*. That is, we may rewrite (1.21) as

$$f(x) = \sum_{i=1}^{m} \alpha_i k(x_i, x) + b \tag{1.24}$$

where as before $\alpha_i = y_i/m_{y_i}$ and the offset b is computed analogously. As

Algorithm 1.3 The Perceptron

Perceptron($\mathbf{X}, \mathbf{Y}$) {reads stream of observations (x_i, y_i)}
  Initialize $w = 0$ and $b = 0$
  while There exists some (x_i, y_i) with $y_i(\langle w, x_i \rangle + b) \leq 0$ **do**
    $w \leftarrow w + y_i x_i$ and $b \leftarrow b + y_i$
  end while

Algorithm 1.4 The Kernel Perceptron

KernelPerceptron($\mathbf{X}, \mathbf{Y}$) {reads stream of observations (x_i, y_i)}
  Initialize $f = 0$
  while There exists some (x_i, y_i) with $y_i f(x_i) \leq 0$ **do**
    $f \leftarrow f + y_i k(x_i, \cdot) + y_i$
  end while

a consequence we have now moved from a fairly simple and pedestrian linear classifier to one which yields a nonlinear function $f(x)$ with a rather nontrivial decision boundary.

1.3.4 Perceptron

In the previous sections we assumed that our classifier had access to a training set of spam and non-spam emails. In real life, such a set might be difficult to obtain all at once. Instead, a user might want to have *instant* results whenever a new e-mail arrives and he would like the system to learn immediately from any corrections to mistakes the system makes.

To overcome both these difficulties one could envisage working with the following protocol: As emails arrive our algorithm classifies them as spam or non-spam, and the user provides feedback as to whether the classification is correct or incorrect. This feedback is then used to improve the performance of the classifier over a period of time.

This intuition can be formalized as follows: Our classifier maintains a parameter vector. At the t-th time instance it receives a data point x_t, to which it assigns a label $\hat{y}_t$ using its current parameter vector. The true label y_t is then revealed, and used to update the parameter vector of the classifier. Such algorithms are said to be *online*. We will now describe perhaps the simplest classifier of this kind namely the Perceptron [Heb49, Ros58].

Let us assume that the data points $x_t \in \mathbb{R}^d$, and labels $y_t \in \{\pm 1\}$. As before we represent an email as a bag-of-words vector and we assign $+1$ to spam emails and -1 to non-spam emails. The Perceptron maintains a weight

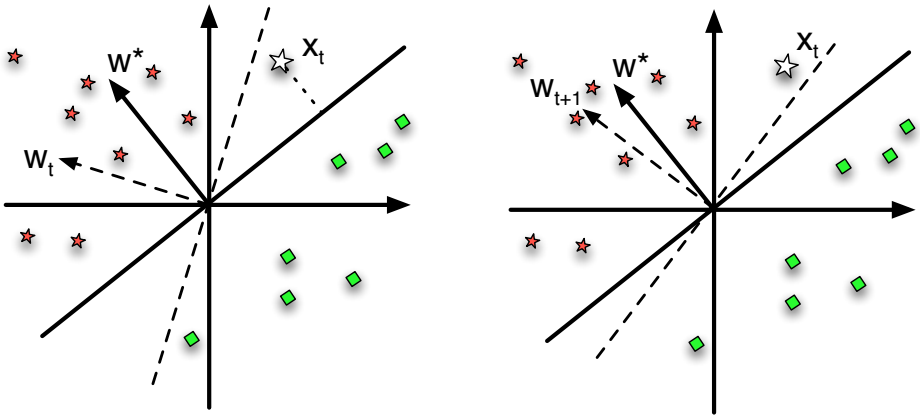

Fig. 1.22. The Perceptron without bias. Left: at time t we have a weight vector w_t denoted by the dashed arrow with corresponding separating plane (also dashed). For reference we include the linear separator w^* and its separating plane (both denoted by a solid line). As a new observation x_t arrives which happens to be mis-classified by the current weight vector w_t we perform an update. Also note the margin between the point x_t and the separating hyperplane defined by w^*. Right: This leads to the weight vector w_{t+1} which is more aligned with w^*.

vector $w \in \mathbb{R}^d$ and classifies x_t according to the rule

$$\hat{y}_t := \operatorname{sign}\{\langle w, x_t \rangle + b\}, \tag{1.25}$$

where $\langle w, x_t \rangle$ denotes the usual Euclidean dot product and b is an offset. Note the similarity of (1.25) to (1.21) of the simple classifier. Just as the latter, the Perceptron is a *linear* classifier which separates its domain $\mathbb{R}^d$ into two halfspaces, namely $\{x | \langle w, x \rangle + b > 0\}$ and its complement. If $\hat{y}_t = y_t$ then no updates are made. On the other hand, if $\hat{y}_t \neq y_t$ the weight vector is updated as

$$w \leftarrow w + y_t x_t \text{ and } b \leftarrow b + y_t. \tag{1.26}$$

Figure 1.22 shows an update step of the Perceptron algorithm. For simplicity we illustrate the case without bias, that is, where $b = 0$ and where it remains unchanged. A detailed description of the algorithm is given in Algorithm 1.3.

An important property of the algorithm is that it performs updates on w by multiples of the observations x_i on which it makes a mistake. Hence we may express w as $w = \sum_{i \in \text{Error}} y_i x_i$. Just as before, we can replace x_i and x by $\phi(x_i)$ and $\phi(x)$ to obtain a kernelized version of the Perceptron algorithm [FS99] (Algorithm 1.4).

If the dataset $(\mathbf{X}, \mathbf{Y})$ is linearly separable, then the Perceptron algorithm

eventually converges and correctly classifies all the points in $\mathbf{X}$. The rate of convergence however depends on the margin. Roughly speaking, the margin quantifies how linearly separable a dataset is, and hence how easy it is to solve a given classification problem.

Definition 1.6 (Margin) *Let $w \in \mathbb{R}^d$ be a weight vector and let $b \in \mathbb{R}$ be an offset. The margin of an observation $x \in \mathbb{R}^d$ with associated label y is*

$$\gamma(x, y) := y\left(\langle w, x\rangle + b\right). \tag{1.27}$$

Moreover, the margin of an entire set of observations $\mathbf{X}$ with labels $\mathbf{Y}$ is

$$\gamma(\mathbf{X}, \mathbf{Y}) := \min_i \gamma(x_i, y_i). \tag{1.28}$$

Geometrically speaking (see Figure 1.22) the margin measures the distance of x from the hyperplane defined by $\{x|\,\langle w, x\rangle + b = 0\}$. Larger the margin, the more well separated the data and hence easier it is to find a hyperplane with correctly classifies the dataset. The following theorem asserts that if there exists a linear classifier which can classify a dataset with a large margin, then the Perceptron will also correctly classify the same dataset after making a small number of mistakes.

Theorem 1.7 (Novikoff's theorem) *Let $(\mathbf{X}, \mathbf{Y})$ be a dataset with at least one example labeled $+1$ and one example labeled -1. Let $R := \max_t \|x_t\|$, and assume that there exists (w^*, b^*) such that $\|w^*\| = 1$ and $\gamma_t := y_t(\langle w^*, x_t\rangle + b^*) \geq \gamma$ for all t. Then, the Perceptron will make at most $\frac{(1+R^2)(1+(b^*)^2)}{\gamma^2}$ mistakes.*

This result is remarkable since it does *not* depend on the dimensionality of the problem. Instead, it only depends on the *geometry* of the setting, as quantified via the margin γ and the radius R of a ball enclosing the observations. Interestingly, a similar bound can be shown for Support Vector Machines [Vap95] which we will be discussing in Chapter 7.

Proof We can safely ignore the iterations where no mistakes were made and hence no updates were carried out. Therefore, without loss of generality assume that the t-th update was made after seeing the t-th observation and let w_t denote the weight vector after the update. Furthermore, for simplicity assume that the algorithm started with $w_0 = 0$ and $b_0 = 0$. By the update equation (1.26) we have

$$\begin{aligned}\langle w_t, w^*\rangle + b_t b^* &= \langle w_{t-1}, w^*\rangle + b_{t-1}b^* + y_t(\langle x_t, w^*\rangle + b^*)\\ &\geq \langle w_{t-1}, w^*\rangle + b_{t-1}b^* + \gamma.\end{aligned}$$

By induction it follows that $\langle w_t, w^* \rangle + b_t b^* \geq t\gamma$. On the other hand we made an update because $y_t(\langle x_t, w_{t-1} \rangle + b_{t-1}) < 0$. By using $y_t y_t = 1$,

$$\begin{aligned} \|w_t\|^2 + b_t^2 &= \|w_{t-1}\|^2 + b_{t-1}^2 + y_t^2 \|x_t\|^2 + 1 + 2y_t(\langle w_{t-1}, x_t \rangle + b_{t-1}) \\ &\leq \|w_{t-1}\|^2 + b_{t-1}^2 + \|x_t\|^2 + 1 \end{aligned}$$

Since $\|x_t\|^2 = R^2$ we can again apply induction to conclude that $\|w_t\|^2 + b_t^2 \leq t\left[R^2 + 1\right]$. Combining the upper and the lower bounds, using the Cauchy-Schwartz inequality, and $\|w^*\| = 1$ yields

$$\begin{aligned} t\gamma \leq \langle w_t, w^* \rangle + b_t b^* &= \left\langle \begin{bmatrix} w_t \\ b_t \end{bmatrix}, \begin{bmatrix} w^* \\ b^* \end{bmatrix} \right\rangle \\ &\leq \left\| \begin{bmatrix} w_t \\ b_t \end{bmatrix} \right\| \left\| \begin{bmatrix} w^* \\ b^* \end{bmatrix} \right\| = \sqrt{\|w_t\|^2 + b_t^2} \sqrt{1 + (b^*)^2} \\ &\leq \sqrt{t(R^2+1)} \sqrt{1 + (b^*)^2}. \end{aligned}$$

Squaring both sides of the inequality and rearranging the terms yields an upper bound on the number of updates and hence the number of mistakes. ■

The Perceptron was the building block of research on Neural Networks [Hay98, Bis95]. The key insight was to combine large numbers of such networks, often in a cascading fashion, to larger objects and to fashion optimization algorithms which would lead to classifiers with desirable properties. In this book we will take a complementary route. Instead of increasing the number of nodes we will investigate what happens when increasing the complexity of the feature map ϕ and its associated kernel k. The advantage of doing so is that we will reap the benefits from convex analysis and linear models, possibly at the expense of a slightly more costly function evaluation.

1.3.5 K-Means

All the algorithms we discussed so far are supervised, that is, they assume that labeled training data is available. In many applications this is too much to hope for; labeling may be expensive, error prone, or sometimes impossible. For instance, it is very easy to crawl and collect every page within the `www.purdue.edu` domain, but rather time consuming to assign a topic to each page based on its contents. In such cases, one has to resort to unsupervised learning. A prototypical unsupervised learning algorithm is K-means, which is clustering algorithm. Given $X = \{x_1, \ldots, x_m\}$ the goal of K-means is to partition it into k clusters such that each point in a cluster is similar to points from its own cluster than with points from some other cluster.