
Linear models

AFTER DEALING WITH logical models in the preceding chapters we now move on to a quite different kind of model. The models in this chapter and the next are defined in terms of the geometry of instance space. Geometric models most often assume that instances are described by d real-valued features, and thus $\mathcal{X} = \mathbb{R}^d$. For example, we could describe objects by their position on a map in terms of longitude and latitude ($d = 2$), or in the real world by longitude, latitude and altitude ($d = 3$). While most real-valued features are not intrinsically geometric – think of a person's age or an object's temperature – we can still imagine them being plotted in a d -dimensional Cartesian coordinate system. We can then use geometric concepts such as lines and planes to impose structure on this space, for instance in order to build a classification model. Alternatively, we can use the geometric notion of distance to represent similarity, on the basis that if two points are close together they have similar feature values and thus can be expected to behave similarly with respect to the property of interest. Such distance-based models are the subject of the next chapter. In this chapter we will look at models that can be understood in terms of lines and planes, commonly called *linear models*.

Linearity plays a fundamental role in mathematics and related disciplines, and the mathematics of linear models is well-understood (see [Background 7.1](#) for the most important concepts). In machine learning, linear models are of particular interest because of their simplicity (remember our rule of thumb 'everything should be made as

If x_1 and x_2 are two scalars or vectors of the same dimension and α and β are arbitrary scalars, then $\alpha x_1 + \beta x_2$ is called a *linear combination* of x_1 and x_2 . If f is a *linear function* of x , then

$$f(\alpha x_1 + \beta x_2) = \alpha f(x_1) + \beta f(x_2)$$

In words, the function value of a linear combination of some inputs is a linear combination of their function values. As a special case, if $\beta = 1 - \alpha$ we are taking a weighted average of x_1 and x_2 , and the linearity of f then means that the function value of the weighted average is the weighted average of the function values.

Linear functions take particular forms, depending on the domain and codomain of f . If x and $f(x)$ are scalars, it follows that f is of the form $f(x) = a + bx$ for some constants a and b ; a is called the *intercept* and b the *slope*. If $\mathbf{x} = (x_1, \dots, x_d)$ is a vector and $f(\mathbf{x})$ is a scalar, then f is of the form

$$f(\mathbf{x}) = a + b_1 x_1 + \dots + b_d x_d = a + \mathbf{b} \cdot \mathbf{x} \quad (7.1)$$

with $\mathbf{b} = (b_1, \dots, b_d)$. The equation $f(\mathbf{x}) = 0$ defines a plane in $\mathbb{R}^d$ perpendicular to the *normal vector* $\mathbf{b}$.

The most general case is where $f(\mathbf{x})$ is a d' -dimensional vector, in which case f is of the form $f(\mathbf{x}) = \mathbf{M}\mathbf{x} + \mathbf{t}$, where $\mathbf{M}$ is a d' -by- d matrix representing a *linear transformation* such as a rotation or a scaling, and $\mathbf{t}$ is a d' -vector representing a translation. In this case f is called an *affine transformation* (the difference between linear and affine transformations is that the former maps the origin to itself; notice that a linear function of the form of Equation 7.1 is a linear transformation only if the intercept is 0).

In all these forms we can avoid representing the intercept a or the translation $\mathbf{t}$ separately by using homogeneous coordinates. For instance, by writing $\mathbf{b}^\circ = (a, b_1, \dots, b_d)$ and $\mathbf{x}^\circ = (1, x_1, \dots, x_d)$ in Equation 7.1 we have $f(\mathbf{x}) = \mathbf{b}^\circ \cdot \mathbf{x}^\circ$ (see also Background 1.2 on p.24).

Examples of non-linear functions are the polynomials in x of degree $p > 1$: $g(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_p x^p = \sum_{i=0}^p a_i x^i$. Other non-linear functions can be approximated by a polynomial through their Taylor expansion. The *linear approximation* of a function g at x_0 is $g(x_0) + g'(x_0)(x - x_0)$, where $g'(x)$ is the derivative of x . A *piecewise linear* approximation is obtained by combining several linear approximations at different points x_0 .

Background 7.1. Linear models.

simple as possible, but not simpler' that we introduced on p.30). Here are a couple of manifestations of this simplicity.

☞ Linear models are *parametric*, meaning that they have a fixed form with a small number of numeric parameters that need to be learned from data. This is

different from tree or rule models, where the structure of the model (e.g., which features to use in the tree, and where) is not fixed in advance.

- ☞ Linear models are stable, which is to say that small variations in the training data have only limited impact on the learned model. Tree models tend to vary more with the training data, as the choice of a different split at the root of the tree typically means that the rest of the tree is different as well.
- ☞ Linear models are less likely to overfit the training data than some other models, largely because they have relatively few parameters. The flipside of this is that they sometimes lead to *underfitting*: e.g., imagine you are learning where the border runs between two countries from labelled samples, then a linear model is unlikely to give a good approximation.

The last two points can be summarised by saying that linear models have low variance but high bias. Such models are often preferable when you have limited data and want to avoid overfitting. High variance–low bias models such as decision trees are preferable if data is abundant but underfitting is a concern. It is usually a good idea to start with simple, high-bias models such as linear models and only move on to more elaborate models if the simpler ones appear to be underfitting.

Linear models exist for all predictive tasks, including classification, probability estimation and regression. Linear regression, in particular, is a well-studied problem that can be solved by the least-squares method, which is the topic of the next section. We will look at a number of other linear models in this chapter, including least-squares classification (also in [Section 7.1](#)), the perceptron in [Section 7.2](#), and the support vector machine in [Section 7.3](#). We will also find out how these models can be turned into probability estimators in [Section 7.4](#). Finally, [Section 7.5](#) briefly discusses how each of these methods could learn non-linear models by means of so-called kernel functions.

7.1 The least-squares method

We start by introducing a method that can be used to learn linear models for classification and regression. Recall that the regression problem is to learn a function estimator $\hat{f} : \mathcal{X} \rightarrow \mathbb{R}$ from examples $(x_i, f(x_i))$, where in this chapter we assume $\mathcal{X} = \mathbb{R}^d$. The differences between the actual and estimated function values on the training examples are called *residuals* $\epsilon_i = f(x_i) - \hat{f}(x_i)$. The *least-squares method*, introduced by Carl Friedrich Gauss in the late eighteenth century, consists in finding $\hat{f}$ such that $\sum_{i=1}^n \epsilon_i^2$ is minimised. The following example illustrates the method in the simple case of a single feature, which is called *univariate regression*.

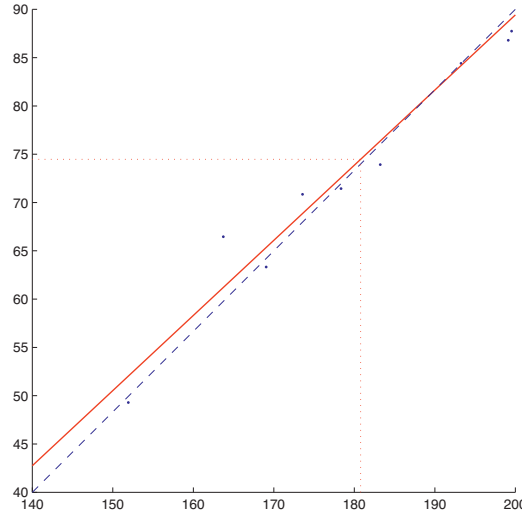


Figure 7.1. The red solid line indicates the result of applying linear regression to 10 measurements of body weight (on the y-axis, in kilograms) against body height (on the x-axis, in centimetres). The orange dotted lines indicate the average height $\bar{h} = 181$ and the average weight $\bar{w} = 74.5$; the regression coefficient $\hat{b} = 0.78$. The measurements were simulated by adding normally distributed noise with mean 0 and variance 5 to the true model indicated by the blue dashed line ($b = 0.83$).

Example 7.1 (Univariate linear regression). Suppose we want to investigate the relationship between people's height and weight. We collect n height and weight measurements (h_i, w_i) , $1 \leq i \leq n$. Univariate linear regression assumes a linear equation $w = a + bh$, with parameters a and b chosen such that the sum of squared residuals $\sum_{i=1}^n (w_i - (a + bh_i))^2$ is minimised. In order to find the parameters we take partial derivatives of this expression, set the partial derivatives to 0 and solve for a and b :

$$\begin{aligned} \frac{\partial}{\partial a} \sum_{i=1}^n (w_i - (a + bh_i))^2 &= -2 \sum_{i=1}^n (w_i - (a + bh_i)) = 0 & \Rightarrow \hat{a} = \bar{w} - \hat{b}\bar{h} \\ \frac{\partial}{\partial b} \sum_{i=1}^n (w_i - (a + bh_i))^2 &= -2 \sum_{i=1}^n (w_i - (a + bh_i))h_i = 0 \\ &\Rightarrow \hat{b} = \frac{\sum_{i=1}^n (h_i - \bar{h})(w_i - \bar{w})}{\sum_{i=1}^n (h_i - \bar{h})^2} \end{aligned}$$

So the solution found by linear regression is $w = \hat{a} + \hat{b}h = \bar{w} + \hat{b}(h - \bar{h})$; see Figure 7.1 for an example.

It is worthwhile to note that the expression for the *regression coefficient* or slope $\hat{b}$ derived in this example has n times the covariance between h and w in the numerator and n times the variance of h in the denominator. This is true in general: for a feature x and a target variable y , the regression coefficient is

$$\hat{b} = n \frac{\sigma_{xy}}{n\sigma_{xx}} = \frac{\sigma_{xy}}{\sigma_{xx}}$$

(Here I use σ_{xx} as an alternative notation for σ_x^2 , the variance of variable x .) This can be understood by noting that the covariance is measured in units of x times units of y (e.g., metres times kilograms in [Example 7.1](#)) and the variance in units of x squared (e.g., metres squared), so their quotient is measured in units of y per unit of x (e.g., kilograms per metre).

We can notice a few more useful things. The intercept $\hat{a}$ is such that the regression line goes through $(\bar{x}, \bar{y})$. Adding a constant to all x -values (a translation) will affect only the intercept but not the regression coefficient (since it is defined in terms of deviations from the mean, which are unaffected by a translation). So we could *zero-centre* the x -values by subtracting $\bar{x}$, in which case the intercept is equal to $\bar{y}$. We could even subtract $\bar{y}$ from all y -values to achieve a zero intercept, without changing the problem in an essential way.

Furthermore, suppose we replace x_i with $x'_i = x_i/\sigma_{xx}$ and likewise $\bar{x}$ with $\bar{x}' = \bar{x}/\sigma_{xx}$, then we have that $\hat{b} = \frac{1}{n} \sum_{i=1}^n (x'_i - \bar{x}') (y_i - \bar{y}) = \sigma_{x'y}$. In other words, if we *normalise* x by dividing all its values by x 's variance, we can take the covariance between the normalised feature and the target variable as regression coefficient. In other words, univariate linear regression can be understood as consisting of two steps:

1. normalisation of the feature by dividing its values by the feature's variance;
2. calculating the covariance of the target variable and the normalised feature.

We will see below how these two steps change when dealing with more than one feature.

Another important point to note is that the sum of the residuals of the least-squares solution is zero:

$$\sum_{i=1}^n (y_i - (\hat{a} + \hat{b}x_i)) = n(\bar{y} - \hat{a} - \hat{b}\bar{x}) = 0$$

The result follows because $\hat{a} = \bar{y} - \hat{b}\bar{x}$, as derived in [Example 7.1](#). While this property is intuitively appealing, it is worth keeping in mind that it also makes linear regression susceptible to *outliers*: points that are far removed from the regression line, often because of measurement errors.

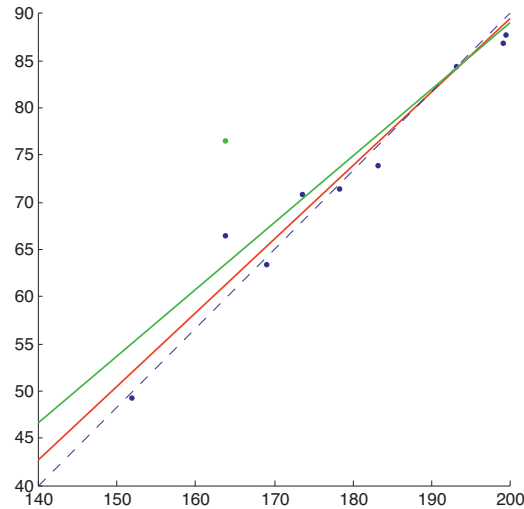


Figure 7.2. The effect of an outlier in univariate regression. One of the blue points got moved up 10 units to the green point, changing the red regression line to the green line.

Example 7.2 (The effect of outliers). Suppose that, as the result of a transcription error, one of the weight values in Figure 7.1 is increased by 10 kg. Figure 7.2 shows that this has a considerable effect on the least-squares regression line.

Despite this sensitivity to outliers, the least-squares method usually works surprisingly well for such a simple method. How can it be justified? One way to look at this is to assume that the true function f is indeed linear, but that the observed y -values are contaminated with random noise. That is, our examples are $(x_i, f(x_i) + \epsilon_i)$ rather than $(x_i, f(x_i))$, and we assume that $f(x) = ax + b$ for some a and b . If we knew a and b we could work out exactly what the residuals are, and if we knew σ^2 we could calculate the probability of observing that set of residuals. Since we don't know a and b we have to estimate them, and the estimate we want is the value of a and b that maximises the probability of the residuals. We will see in Chapter 9 that this so-called *maximum-likelihood estimate* is exactly the least-squares solution.

Variants of the least-squares method exist. Here we discussed *ordinary* least squares, which assumes that only the y -values are contaminated with random noise. *Total* least squares generalises this to the situation that both x - and y -values are noisy, but this does not necessarily have a unique solution.

$\mathbf{X}$ usually denotes an n -by- d data matrix containing n instances in rows described by d features or variables in columns. $\mathbf{X}_r$ denotes the r -th row of $\mathbf{X}$, $\mathbf{X}_c$ denotes the c -th column, and $\mathbf{X}_{rc}$ denotes the entry in the r -th row and c -th column. We also use i and j to range over rows and columns, respectively. The j -th column mean is defined as $\mu_j = \frac{1}{n} \sum_{i=1}^n \mathbf{X}_{ij}$; $\boldsymbol{\mu}^T$ is a row vector containing all column means. If $\mathbf{1}$ is an n -vector containing only ones, then $\mathbf{1}\boldsymbol{\mu}^T$ is an n -by- d matrix whose rows are $\boldsymbol{\mu}^T$; hence $\mathbf{X}' = \mathbf{X} - \mathbf{1}\boldsymbol{\mu}^T$ has mean zero in each column and is referred to as the **zero-centred** data matrix.

The **scatter matrix** is the d -by- d matrix $\mathbf{S} = \mathbf{X}'^T \mathbf{X}' = (\mathbf{X} - \mathbf{1}\boldsymbol{\mu}^T)^T (\mathbf{X} - \mathbf{1}\boldsymbol{\mu}^T) = \mathbf{X}^T \mathbf{X} - n\mathbf{M}$, where $\mathbf{M} = \boldsymbol{\mu}\boldsymbol{\mu}^T$ is a d -by- d matrix whose entries are products of column means $\mathbf{M}_{jc} = \mu_j \mu_c$. The **covariance matrix** of $\mathbf{X}$ is $\boldsymbol{\Sigma} = \frac{1}{n} \mathbf{S}$ whose entries are the pairwise covariances $\sigma_{jc} = \frac{1}{n} \sum_{i=1}^n (\mathbf{X}_{ij} - \mu_j)(\mathbf{X}_{ic} - \mu_c) = \frac{1}{n} \left(\sum_{i=1}^n \mathbf{X}_{ij} \mathbf{X}_{ic} - \mu_i \mu_c \right)$. Two uncorrelated features have a covariance close to 0; positively correlated features have a positive covariance, indicating a certain tendency to increase or decrease together; a negative covariance indicates that if one feature increases, the other tends to decrease and vice versa. $\sigma_{jj} = \frac{1}{n} \sum_{i=1}^n (\mathbf{X}_{ij} - \mu_j)^2 = \frac{1}{n} \left(\sum_{i=1}^n \mathbf{X}_{ij}^2 - \mu_j^2 \right)$ is the **variance** of column j , also denoted as σ_j^2 . The variance is always positive and indicates the spread of the values of a feature around their mean.

A small example clarifies these definitions:

$$\begin{aligned} \mathbf{X} &= \begin{pmatrix} 5 & 0 \\ 3 & 5 \\ 1 & 7 \end{pmatrix} & \mathbf{1}\boldsymbol{\mu}^T &= \begin{pmatrix} 3 & 4 \\ 3 & 4 \\ 3 & 4 \end{pmatrix} & \mathbf{X}' &= \begin{pmatrix} 2 & -4 \\ 0 & 1 \\ -2 & 3 \end{pmatrix} & \mathbf{G} &= \begin{pmatrix} 25 & 15 & 5 \\ 15 & 34 & 38 \\ 5 & 38 & 50 \end{pmatrix} \\ \mathbf{X}^T \mathbf{X} &= \begin{pmatrix} 35 & 22 \\ 22 & 74 \end{pmatrix} & \mathbf{M} &= \begin{pmatrix} 9 & 12 \\ 12 & 16 \end{pmatrix} & \mathbf{S} &= \begin{pmatrix} 8 & -14 \\ -14 & 26 \end{pmatrix} & \boldsymbol{\Sigma} &= \begin{pmatrix} 8/3 & -14/3 \\ -14/3 & 26/3 \end{pmatrix} \end{aligned}$$

We see that the two features are negatively correlated and that the second feature has the larger variance. Another way to calculate the scatter matrix is as a sum of outer products, one for each data point: $\mathbf{S} = \sum_{i=1}^n (\mathbf{X}_i - \boldsymbol{\mu}^T)^T (\mathbf{X}_i - \boldsymbol{\mu}^T)$. In our example we have

$$\begin{aligned} (\mathbf{X}_1 - \boldsymbol{\mu}^T)^T (\mathbf{X}_1 - \boldsymbol{\mu}^T) &= \begin{pmatrix} 2 \\ -4 \end{pmatrix} \begin{pmatrix} 2 & -4 \end{pmatrix} = \begin{pmatrix} 4 & -8 \\ -8 & 16 \end{pmatrix} \\ (\mathbf{X}_2 - \boldsymbol{\mu}^T)^T (\mathbf{X}_2 - \boldsymbol{\mu}^T) &= \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \\ (\mathbf{X}_3 - \boldsymbol{\mu}^T)^T (\mathbf{X}_3 - \boldsymbol{\mu}^T) &= \begin{pmatrix} -2 \\ 3 \end{pmatrix} \begin{pmatrix} -2 & 3 \end{pmatrix} = \begin{pmatrix} 4 & -6 \\ -6 & 9 \end{pmatrix} \end{aligned}$$

Background 7.2. Some more matrix notation.

Multivariate linear regression

In order to deal with an arbitrary number of features it will be useful to employ matrix notation (see [Background 7.2](#)). We can write univariate linear regression in matrix form as

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} a + \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} b + \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{pmatrix}$$

$$\mathbf{y} = \mathbf{a} + \mathbf{X}\mathbf{b} + \boldsymbol{\epsilon}$$

In the second form of this equation, $\mathbf{y}$, $\mathbf{a}$, $\mathbf{X}$ and $\boldsymbol{\epsilon}$ are n -vectors, and $\mathbf{b}$ is a scalar. In case of d features, all that changes is that $\mathbf{X}$ becomes an n -by- d matrix, and $\mathbf{b}$ becomes a d -vector of regression coefficients.

We can apply the by now familiar trick of using homogeneous coordinates to simplify these equations as follows:

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{pmatrix}$$

$$\mathbf{y} = \mathbf{X}^\circ \mathbf{w} + \boldsymbol{\epsilon}$$

with $\mathbf{X}^\circ$ an n -by- $(d+1)$ matrix whose first column is all 1s and the remaining columns are the columns of $\mathbf{X}$, and $\mathbf{w}$ has the intercept as its first entry and the regression coefficients as the remaining d entries. For convenience we will often blur the distinction between these two formulations and state the regression equation as $\mathbf{y} = \mathbf{X}\mathbf{w} + \boldsymbol{\epsilon}$ with $\mathbf{X}$ having d columns and $\mathbf{w}$ having d rows – from the context it will be clear whether we are representing the intercept by means of homogeneous coordinates, or have rather zero-centred the target and features to achieve a zero intercept.

In the univariate case we were able to obtain a closed-form solution for $\mathbf{w}$: can we do the same in the multivariate case? First, we are likely to need the covariances between every feature and the target variable. Consider the expression $\mathbf{X}^T \mathbf{y}$, which is an n -vector, the j -th entry of which is the product of the j -th row of $\mathbf{X}^T$ – i.e., the j -th column of $\mathbf{X}$, which is $(x_{1j}, \dots, x_{nj})$ – with $(y_1, \dots, y_n)$:

$$(\mathbf{X}^T \mathbf{y})_j = \sum_{i=1}^n x_{ij} y_i = \sum_{i=1}^n (x_{ij} - \mu_j)(y_i - \bar{y}) + n\mu_j \bar{y} = n(\sigma_{jy} + \mu_j \bar{y})$$

Assuming for the moment that every feature is zero-centred, we have $\mu_j = 0$ and thus $\mathbf{X}^T \mathbf{y}$ is an n -vector holding all the required covariances (times n).

In the univariate case we needed to normalise the features to have unit variance. In the multivariate case we can achieve this by means of a d -by- d scaling matrix: a

diagonal matrix with diagonal entries $1/n\sigma_{jj}$. If $\mathbf{S}$ is a diagonal matrix with diagonal entries $n\sigma_{jj}$, we can get the required scaling matrix by simply inverting $\mathbf{S}$. So our first stab at a solution for the *multivariate regression* problem is

$$\hat{\mathbf{w}} = \mathbf{S}^{-1} \mathbf{X}^T \mathbf{y} \quad (7.2)$$

As it turns out, the general case requires a more elaborate matrix instead of $\mathbf{S}$:

$$\hat{\mathbf{w}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \quad (7.3)$$

Let us try to understand the term $(\mathbf{X}^T \mathbf{X})^{-1}$ a bit better. Assume that the features are uncorrelated (meaning the covariance between every pair of different features is 0) in addition to being zero-centred. In the notation of [Background 7.2](#), the covariance matrix $\mathbf{\Sigma}$ is diagonal with entries σ_{jj} . Since $\mathbf{X}^T \mathbf{X} = n(\mathbf{\Sigma} + \mathbf{M})$, and since the entries of $\mathbf{M}$ are 0 because the columns of $\mathbf{X}$ are zero-centred, this matrix is also diagonal with entries $n\sigma_{jj}$ – in fact, it is the matrix $\mathbf{S}$ referred to above. In other words, assuming zero-centred and uncorrelated features, $(\mathbf{X}^T \mathbf{X})^{-1}$ reduces to our scaling matrix $\mathbf{S}^{-1}$. In the general case we cannot make any assumptions about the features, and $(\mathbf{X}^T \mathbf{X})^{-1}$ *acts as a transformation that decorrelates, centres and normalises the features*.

To make this a bit more concrete, the next example shows how this works out in the bivariate case.

Example 7.3 (Bivariate linear regression in matrix notation). First, we derive the basic expressions.

$$\begin{aligned} \mathbf{X}^T \mathbf{X} &= \begin{pmatrix} x_{11} & \cdots & x_{n1} \\ x_{12} & \cdots & x_{n2} \end{pmatrix} \begin{pmatrix} x_{11} & x_{12} \\ \vdots & \vdots \\ x_{n1} & x_{n2} \end{pmatrix} = n \begin{pmatrix} \sigma_{11} + \bar{x}_1^2 & \sigma_{12} + \bar{x}_1 \bar{x}_2 \\ \sigma_{12} + \bar{x}_1 \bar{x}_2 & \sigma_{22} + \bar{x}_2^2 \end{pmatrix} \\ (\mathbf{X}^T \mathbf{X})^{-1} &= \frac{1}{nD} \begin{pmatrix} \sigma_{22} + \bar{x}_2^2 & -\sigma_{12} - \bar{x}_1 \bar{x}_2 \\ -\sigma_{12} - \bar{x}_1 \bar{x}_2 & \sigma_{11} + \bar{x}_1^2 \end{pmatrix} \\ D &= (\sigma_{11} + \bar{x}_1^2)(\sigma_{22} + \bar{x}_2^2) - (\sigma_{12} + \bar{x}_1 \bar{x}_2)^2 \\ \mathbf{X}^T \mathbf{y} &= \begin{pmatrix} x_{11} & \cdots & x_{n1} \\ x_{12} & \cdots & x_{n2} \end{pmatrix} \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = n \begin{pmatrix} \sigma_{1y} + \bar{x}_1 \bar{y} \\ \sigma_{2y} + \bar{x}_2 \bar{y} \end{pmatrix} \end{aligned}$$

We now consider two special cases. The first is that $\mathbf{X}$ is in homogeneous coordinates, i.e., we are really dealing with a univariate problem. In that case we have

$x_{i1} = 1$ for $1 \leq i \leq n$; $\bar{x}_1 = 1$; and $\sigma_{11} = \sigma_{12} = \sigma_{1y} = 0$. We then obtain (we write x instead of x_2 , σ_{xx} instead of σ_{22} and σ_{xy} instead of σ_{2y}):

$$\begin{aligned} (\mathbf{X}^T \mathbf{X})^{-1} &= \frac{1}{n\sigma_{xx}} \begin{pmatrix} \sigma_{xx} + \bar{x}^2 & -\bar{x} \\ -\bar{x} & 1 \end{pmatrix} \\ \mathbf{X}^T \mathbf{y} &= n \begin{pmatrix} \bar{y} \\ \sigma_{xy} + \bar{x} \bar{y} \end{pmatrix} \\ \hat{\mathbf{w}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} &= \frac{1}{\sigma_{xx}} \begin{pmatrix} \sigma_{xx} \bar{y} - \sigma_{xy} \bar{x} \\ \sigma_{xy} \end{pmatrix} \end{aligned}$$

This is the same result as obtained in [Example 7.1](#).

The second special case we consider is where we assume x_1 , x_2 and y to be zero-centred, which means that the intercept is zero and $\mathbf{w}$ contains the two regression coefficients. In this case we obtain

$$\begin{aligned} (\mathbf{X}^T \mathbf{X})^{-1} &= \frac{1}{n(\sigma_{11}\sigma_{22} - \sigma_{12}^2)} \begin{pmatrix} \sigma_{22} & -\sigma_{12} \\ -\sigma_{12} & \sigma_{11} \end{pmatrix} \\ \mathbf{X}^T \mathbf{y} &= n \begin{pmatrix} \sigma_{1y} \\ \sigma_{2y} \end{pmatrix} \\ \hat{\mathbf{w}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} &= \frac{1}{(\sigma_{11}\sigma_{22} - \sigma_{12}^2)} \begin{pmatrix} \sigma_{22}\sigma_{1y} - \sigma_{12}\sigma_{2y} \\ \sigma_{11}\sigma_{2y} - \sigma_{12}\sigma_{1y} \end{pmatrix} \end{aligned}$$

The last expression shows, e.g., that the regression coefficient for x_1 may be non-zero even if x_1 doesn't correlate with the target variable ($\sigma_{1y} = 0$), on account of the correlation between x_1 and x_2 ($\sigma_{12} \neq 0$).

Notice that if we do assume $\sigma_{12} = 0$ then the components of $\hat{\mathbf{w}}$ reduce to σ_{jy}/σ_{jj} , which brings us back to [Equation 7.2](#). *Assuming uncorrelated features effectively decomposes a multivariate regression problem into d univariate problems*. We shall see several other examples of decomposing multivariate learning problems into univariate problems in this book – in fact, we have already seen an example in the form of the [naive Bayes](#) classifier in [Chapter 1](#). So, you may wonder, why take feature correlation into account at all?

The answer is that ignoring feature correlation can be harmful in certain situations. Consider [Figure 7.3](#): on the left, there is little correlation among the features, and as a result the samples provide a lot of information about the true function. On

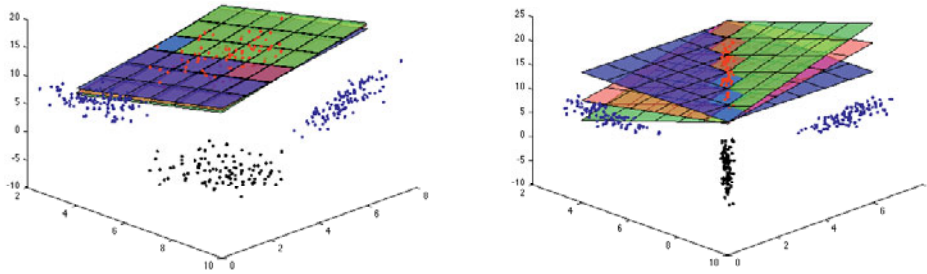


Figure 7.3. (left) Regression functions learned by linear regression. The true function is $y = x_1 + x_2$ (red plane). The red points are noisy samples of this function; the black points show them projected onto the (x_1, x_2) -plane. The green plane indicates the function learned by linear regression; the blue plane is the result of decomposing the problem into two univariate regression problems (blue points). Both are good approximations of the true function. (right) The same function, but now x_1 and x_2 are highly (negatively) correlated. The samples now give much less information about the true function: indeed, from the univariate decomposition it appears that the function is constant.

the right, the features are highly negatively correlated in such a way that the sampled values $y = x_1 + x_2 + \epsilon$ appear nearly constant, as any increase in one feature is accompanied by a nearly equal decrease in the other. As a result, decomposing the problem into two univariate regression problems leads to learning a nearly constant function. To be fair, taking the full covariance matrix into account doesn't do so well either in this example. However, although we will not explore the details here, one advantage of the full covariance approach is that it allows us to recognise that we can't place much confidence in our estimates of the regression parameters in this situation. The computational cost of computing the closed-form solution in Equation 7.3 lies in inverting the d -by- d matrix $\mathbf{X}^T \mathbf{X}$, which can be prohibitive in high-dimensional feature spaces.

Regularised regression

We have just seen a situation in which least-squares regression can become *unstable*: i.e., highly dependent on the training data. Instability is a manifestation of a tendency to overfit. *Regularisation* is a general method to avoid such overfitting by applying additional constraints to the weight vector. A common approach is to make sure the weights are, on average, small in magnitude: this is referred to as *shrinkage*. To show how this can be achieved, we first write down the least-squares regression problem as an optimisation problem:

$$\mathbf{w}^* = \arg \min_{\mathbf{w}} (\mathbf{y} - \mathbf{X}\mathbf{w})^T (\mathbf{y} - \mathbf{X}\mathbf{w})$$

The right-hand side is just a way to write the sum of squared residuals as a dot product. The regularised version of this optimisation is then as follows:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} (\mathbf{y} - \mathbf{X}\mathbf{w})^T (\mathbf{y} - \mathbf{X}\mathbf{w}) + \lambda \|\mathbf{w}\|^2 \quad (7.4)$$

where $\|\mathbf{w}\|^2 = \sum_i w_i^2$ is the squared norm of the vector $\mathbf{w}$, or, equivalently, the dot product $\mathbf{w}^T \mathbf{w}$; λ is a scalar determining the amount of regularisation. This regularised problem still has a closed-form solution:

$$\hat{\mathbf{w}} = (\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T \mathbf{y} \quad (7.5)$$

where $\mathbf{I}$ denotes the identity matrix with 1s on the diagonal and 0s everywhere else. Comparing this with Equation 7.3 on p.202 we see that regularisation amounts to adding λ to the diagonal of $\mathbf{X}^T \mathbf{X}$, a well-known trick to improve the numerical stability of matrix inversion. This form of least-squares regression is known as *ridge regression*.

An interesting alternative form of regularised regression is provided by the *lasso*, which stands for ‘least absolute shrinkage and selection operator’. It replaces the ridge regularisation term $\sum_i w_i^2$ with the sum of absolute weights $\sum_i |w_i|$. (Using terminology that will be introduced in Definition 8.2 on p.235: lasso uses L_1 regularisation where ridge regression uses the L_2 norm.) The result is that some weights are shrunk, but others are set to 0, and so the lasso regression favours *sparse solutions*. It should be added that lasso regression is quite sensitive to the regularisation parameter λ , which is usually set on hold-out data or in cross-validation. Also, there is no closed-form solution and so some numerical optimisation technique must be applied.

Using least-squares regression for classification

So far we have used the least-squares method to construct function approximators. Interestingly, we can also use linear regression to learn a binary classifier by encoding the two classes as real numbers. For instance, we can label the *Pos* positive examples with $y^{\oplus} = +1$ and the *Neg* negative examples with $y^{\ominus} = -1$. It then follows that $\mathbf{X}^T \mathbf{y} = \text{Pos } \boldsymbol{\mu}^{\oplus} - \text{Neg } \boldsymbol{\mu}^{\ominus}$, where $\boldsymbol{\mu}^{\oplus}$ and $\boldsymbol{\mu}^{\ominus}$ are d -vectors containing each feature’s mean values for the positive and negative examples, respectively.

Example 7.4 (Univariate least-squares classifier). In the univariate case we have $\sum_i x_i y_i = \text{Pos } \mu^{\oplus} - \text{Neg } \mu^{\ominus}$; we also know (see Example 7.3) that $\sum_i x_i y_i = n(\sigma_{xy} + \bar{x} \bar{y})$, and so $\sigma_{xy} = \text{pos } \mu^{\oplus} - \text{neg } \mu^{\ominus} - \bar{x} \bar{y}$. Since $\bar{x} = \text{pos } \mu^{\oplus} + \text{neg } \mu^{\ominus}$ and $\bar{y} = \text{pos} - \text{neg}$, we can rewrite the covariance between x and y as $\sigma_{xy} =$

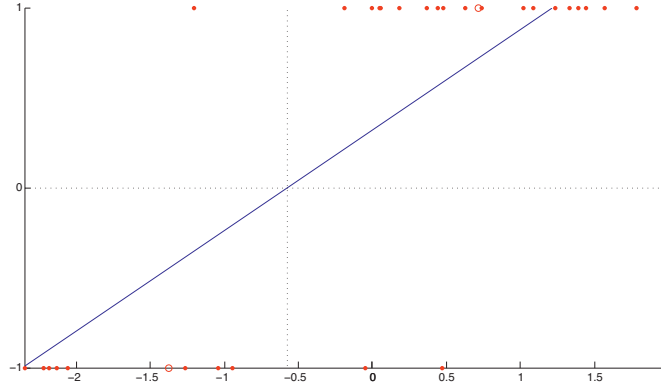


Figure 7.4. Using univariate linear regression to obtain a decision boundary. The 10 negative examples are labelled with $y^\ominus = -1$ and the 20 positive examples are labelled $y^\oplus = +1$. $\mu^\ominus$ and $\mu^\oplus$ are indicated by red circles. The blue line is the linear regression line $y = \bar{y} + \hat{b}(x - \bar{x})$, and the crosshair indicates the decision boundary $x_0 = \bar{x} - \bar{y}/\hat{b}$. This results in three examples being misclassified – notice that this is the best that can be achieved with the given data.

$2pos \cdot neg (\mu^\oplus - \mu^\ominus)$, and so the slope of the regression line is

$$\hat{b} = 2pos \cdot neg \frac{\mu^\oplus - \mu^\ominus}{\sigma_{xx}} \quad (7.6)$$

This equation shows that the slope of the regression line increases with the separation between the classes (measured as the distance between the class means in proportion to the feature's variance), but also decreases if the class distribution becomes skewed.

The regression equation $y = \bar{y} + \hat{b}(x - \bar{x})$ can then be used to obtain a decision boundary. We need to determine the point (x_0, y_0) such that y_0 is half-way between $y^\oplus$ and $y^\ominus$ (i.e., $y_0 = 0$ in our case). We then have

$$x_0 = \bar{x} + \frac{y_0 - \bar{y}}{\hat{b}} = \bar{x} - \frac{pos - neg}{2pos \cdot neg} \frac{\sigma_{xx}}{\mu^\oplus - \mu^\ominus}$$

That is, if there are equal numbers of positive and negative examples we simply threshold the feature at the feature mean $\bar{x}$; in case of unequal class distribution we shift this threshold to the left or right as appropriate (Figure 7.4).

In the general case, the *least-squares classifier* learns the decision boundary $\mathbf{w} \cdot \mathbf{x} = t$ with

$$\mathbf{w} = (\mathbf{X}^T \mathbf{X})^{-1} (Pos \mu^\oplus - Neg \mu^\ominus) \quad (7.7)$$

We would hence assign class $\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x} - t)$ to instance $\mathbf{x}$, where

$$\text{sign}(x) = \begin{cases} +1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Various simplifying assumptions can be made, including zero-centred features, equal-variance features, uncorrelated features and equal class prevalences. In the simplest case, when all these assumptions are made, Equation 7.7 reduces to $\mathbf{w} = c(\boldsymbol{\mu}^{\oplus} - \boldsymbol{\mu}^{\ominus})$ where c is some scalar that can be incorporated in the decision threshold t . We recognise this as the *basic linear classifier* that was introduced in the Prologue. Equation 7.7 thus tells us how to adapt the basic linear classifier, using the least-squares method, in order to take feature correlation and unequal class distributions into account.

In summary, *a general way of constructing a linear classifier with decision boundary $\mathbf{w} \cdot \mathbf{x} = t$ is by constructing $\mathbf{w}$ as $\mathbf{M}^{-1}(n^{\oplus} \boldsymbol{\mu}^{\oplus} - n^{\ominus} \boldsymbol{\mu}^{\ominus})$* , with different possible choices of $\mathbf{M}$, $n^{\oplus}$ and $n^{\ominus}$. The full covariance approach with $\mathbf{M} = \mathbf{X}^T \mathbf{X}$ has time complexity $O(n^2 d)$ for construction of $\mathbf{M}$ and $O(d^3)$ for inverting it,¹ so this approach becomes unfeasible with large numbers of features.

7.2 The perceptron

Recall from Chapter 1 that labelled data is called *linearly separable* if there exists a linear decision boundary separating the classes. The least-squares classifier may find a perfectly separating decision boundary if one exists, but this is not guaranteed. To see this, suppose that the basic linear classifier achieves perfect separation for a given training set. Now, move all but one of the positive points away from the negative class. The decision boundary will also move away from the negative class, at some point crossing the one positive that remains fixed. By construction, the modified data is still linearly separable, since the original decision boundary separates it; however, the statistics of the modified data are such that the basic linear classifier will misclassify the one positive outlier.

A linear classifier that will achieve perfect separation on linearly separable data is the *perceptron*, originally proposed as a simple neural network. The perceptron iterates over the training set, updating the weight vector every time it encounters an incorrectly classified example. For example, let $\mathbf{x}_i$ be a misclassified positive example, then we have $y_i = +1$ and $\mathbf{w} \cdot \mathbf{x}_i < t$. We therefore want to find $\mathbf{w}'$ such that $\mathbf{w}' \cdot \mathbf{x}_i > \mathbf{w} \cdot \mathbf{x}_i$, which moves the decision boundary towards and hopefully past $\mathbf{x}_i$. This can be achieved by calculating the new weight vector as $\mathbf{w}' = \mathbf{w} + \eta \mathbf{x}_i$, where $0 < \eta \leq 1$ is the *learning rate*. We then have $\mathbf{w}' \cdot \mathbf{x}_i = \mathbf{w} \cdot \mathbf{x}_i + \eta \mathbf{x}_i \cdot \mathbf{x}_i > \mathbf{w} \cdot \mathbf{x}_i$ as required. Similarly, if $\mathbf{x}_j$ is a misclassified

¹A more sophisticated algorithm can achieve $O(d^{2.8})$, but this is probably the best we can do.

negative example, then we have $y_j = -1$ and $\mathbf{w} \cdot \mathbf{x}_j > t$. In this case we calculate the new weight vector as $\mathbf{w}' = \mathbf{w} - \eta \mathbf{x}_j$, and thus $\mathbf{w}' \cdot \mathbf{x}_j = \mathbf{w} \cdot \mathbf{x}_j - \eta \mathbf{x}_j \cdot \mathbf{x}_j < \mathbf{w} \cdot \mathbf{x}_j$. The two cases can be combined in a single update rule:

$$\mathbf{w}' = \mathbf{w} + \eta y_i \mathbf{x}_i \quad (7.8)$$

The perceptron training algorithm is given in [Algorithm 7.1](#). It iterates through the training examples until all examples are correctly classified. The algorithm can easily be turned into an *online* algorithm that processes a stream of examples, updating the weight vector only if the last received example is misclassified. The perceptron is guaranteed to converge to a solution if the training data is linearly separable, but it won't converge otherwise. [Figure 7.5](#) gives a graphical illustration of the perceptron training algorithm. In this particular example I initialised the weight vector to the basic linear classifier, which means the learning rate does have an effect on how quickly we move away from the initial decision boundary. However, if the weight vector is initialised to the zero vector, it is easy to see that the learning rate is just a constant factor that does not affect convergence. We will set it to 1 in the remainder of this section.

The key point of the perceptron algorithm is that, every time an example $\mathbf{x}_i$ is misclassified, we add $y_i \mathbf{x}_i$ to the weight vector. After training has completed, each example has been misclassified zero or more times – denote this number α_i for example $\mathbf{x}_i$.

Algorithm 7.1: *Perceptron(D, η)* – train a perceptron for linear classification.

Input : labelled training data D in homogeneous coordinates;
learning rate η .

Output : weight vector $\mathbf{w}$ defining classifier $\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x})$.

```

1  $\mathbf{w} \leftarrow \mathbf{0}$ ; // Other initialisations of the weight vector are possible
2  $\text{converged} \leftarrow \text{false}$ ;
3 while  $\text{converged} = \text{false}$  do
4    $\text{converged} \leftarrow \text{true}$ ;
5   for  $i = 1$  to  $|D|$  do
6     if  $y_i \mathbf{w} \cdot \mathbf{x}_i \leq 0$  // i.e.,  $\hat{y}_i \neq y_i$ 
7       then
8          $\mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x}_i$ ;
9          $\text{converged} \leftarrow \text{false}$ ; // We changed  $\mathbf{w}$  so haven't converged yet
10      end
11  end
12 end

```

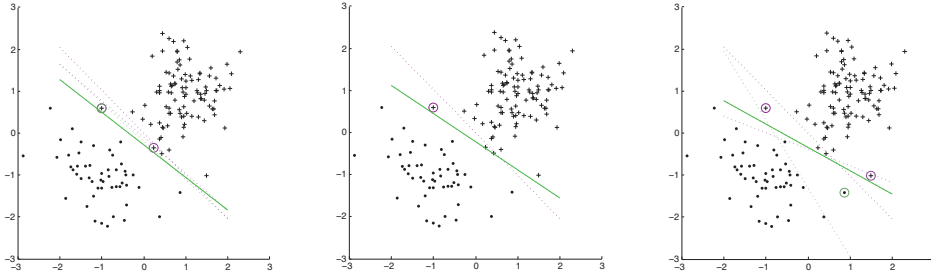


Figure 7.5. (left) A perceptron trained with a small learning rate ($\eta = 0.2$). The circled examples are the ones that trigger the weight update. **(middle)** Increasing the learning rate to $\eta = 0.5$ leads in this case to a rapid convergence. **(right)** Increasing the learning rate further to $\eta = 1$ may lead to too aggressive weight updating, which harms convergence. The starting point in all three cases was the basic linear classifier.

Using this notation the weight vector can be expressed as

$$\mathbf{w} = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i \quad (7.9)$$

In other words, the weight vector is a linear combination of the training instances. The perceptron shares this property with, e.g., the basic linear classifier:

$$\mathbf{w}_{blc} = \mu^{\oplus} - \mu^{\ominus} = \frac{1}{Pos} \sum_{\mathbf{x}^{\oplus} \in Tr^{\oplus}} \mathbf{x}^{\oplus} - \frac{1}{Neg} \sum_{\mathbf{x}^{\ominus} \in Tr^{\ominus}} \mathbf{x}^{\ominus} = \sum_{\mathbf{x}^{\oplus} \in Tr^{\oplus}} \alpha^{\oplus} c(\mathbf{x}^{\oplus}) \mathbf{x}^{\oplus} + \sum_{\mathbf{x}^{\ominus} \in Tr^{\ominus}} \alpha^{\ominus} c(\mathbf{x}^{\ominus}) \mathbf{x}^{\ominus} \quad (7.10)$$

Algorithm 7.2: DualPerceptron(D) – perceptron training in dual form.

Input : labelled training data D in homogeneous coordinates.

Output : coefficients α_i defining weight vector $\mathbf{w} = \sum_{i=1}^{|D|} \alpha_i y_i \mathbf{x}_i$.

```

1  $\alpha_i \leftarrow 0$  for  $1 \leq i \leq |D|$ ;
2 converged  $\leftarrow$  false;
3 while converged = false do
4   converged  $\leftarrow$  true;
5   for  $i = 1$  to  $|D|$  do
6     if  $y_i \sum_{j=1}^{|D|} \alpha_j y_j \mathbf{x}_i \cdot \mathbf{x}_j \leq 0$  then
7        $\alpha_i \leftarrow \alpha_i + 1$ ;
8       converged  $\leftarrow$  false;
9     end
10  end
11 end
```

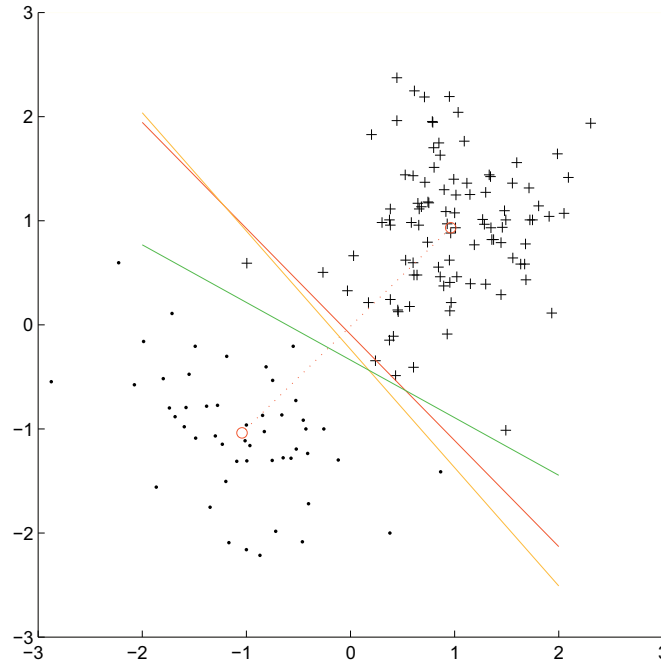


Figure 7.6. Three differently trained linear classifiers on a data set of 100 positives (top-right) and 50 negatives (bottom-left): the **basic linear classifier in red**, the **least-squares classifier in orange** and the **perceptron in green**. Notice that the perceptron perfectly separates the training data, but its heuristic approach may lead to overfitting in certain situations.

where $c(\mathbf{x})$ is the true class of example $\mathbf{x}$ (i.e., +1 or -1), $\alpha^{\oplus} = 1/Pos$ and $\alpha^{\ominus} = 1/Neg$. *In the dual, instance-based view of linear classification we are learning instance weights α_i rather than feature weights w_j .* In this dual perspective, an instance $\mathbf{x}$ is classified as $\hat{y} = \text{sign}(\sum_{i=1}^n \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x})$. This means that, during training, the only information needed about the training data is all pairwise dot products: the n -by- n matrix $\mathbf{G} = \mathbf{XX}^T$ containing these dot products is called the **Gram matrix**. Algorithm 7.2 gives the dual form of the perceptron training algorithm. We will encounter this instance-based perspective again when we discuss support vector machines in the next section.

Figure 7.6 demonstrates the difference between the basic linear classifier, the least-squares classifier and the perceptron on some random data. For this particular data set, neither the basic linear classifier nor the least-squares classifier achieves perfect separation, but the perceptron does. One difference with other linear methods is that we cannot derive a closed-form solution for the weight vector learned by the perceptron, so it is a more heuristic approach.

The perceptron can easily be turned into a linear function approximator (Algorithm 7.3). To this end the update rule is changed to $\mathbf{w}' = \mathbf{w} + (y_i - \hat{y}_i)^2 \mathbf{x}_i$, which uses squared

residuals. This is unlikely to converge to the exact function, so the algorithm simply runs for a fixed number of training epochs (an epoch is one complete run through the training data). Alternatively, one could run the algorithm until a bound on the sum of squared residuals is reached.

7.3 Support vector machines

Linearly separable data admits infinitely many decision boundaries that separate the classes, but intuitively some of these are better than others. For example, the left and middle decision boundaries in Figure 7.5 seem to be unnecessarily close to some of the positives; while the one on the right leaves a bit more space on either side, it doesn't seem particularly good either. To make this a bit more precise, recall that in Section 2.2 we defined the *margin* of an example assigned by a scoring classifier as $c(x)\hat{s}(x)$, where $c(x)$ is $+1$ for positive examples and -1 for negative examples and $\hat{s}(x)$ is the score of example x . If we take $\hat{s}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} - t$, then a true positive $\mathbf{x}_i$ has margin $\mathbf{w} \cdot \mathbf{x}_i - t > 0$ and a true negative $\mathbf{x}_j$ has margin $-(\mathbf{w} \cdot \mathbf{x}_j - t) > 0$. For a given training set and decision boundary, let m^+ be the smallest margin of any positive, and m^- the smallest margin of any negative, then we want the sum of these to be as large as possible. This sum is independent of the decision threshold t , as long as we keep the nearest positives and negatives at the right sides of the decision boundary, and so we re-adjust t such that m^+ and m^- become equal. Figure 7.7 depicts this graphically in a two-dimensional instance space. The training examples nearest to the decision boundary are called *support vectors*: as we shall see, the decision boundary of a support vector machine (SVM) is defined as a linear combination of the support vectors.

The *margin* is thus defined as $m/||\mathbf{w}||$, where m is the distance between the decision boundary and the nearest training instances (at least one of each class) as

Algorithm 7.3: *PerceptronRegression(D, T)* – train a perceptron for regression.

Input : labelled training data D in homogeneous coordinates;
maximum number of training epochs T .

Output : weight vector $\mathbf{w}$ defining function approximator $\hat{y} = \mathbf{w} \cdot \mathbf{x}$.

```

1  $\mathbf{w} \leftarrow \mathbf{0}$ ;  $t \leftarrow 0$ ;
2 while  $t < T$  do
3   for  $i = 1$  to  $|D|$  do
4      $\mathbf{w} \leftarrow \mathbf{w} + (y_i - \hat{y}_i)^2 \mathbf{x}_i$ ;
5   end
6    $t \leftarrow t + 1$ ;
7 end
```

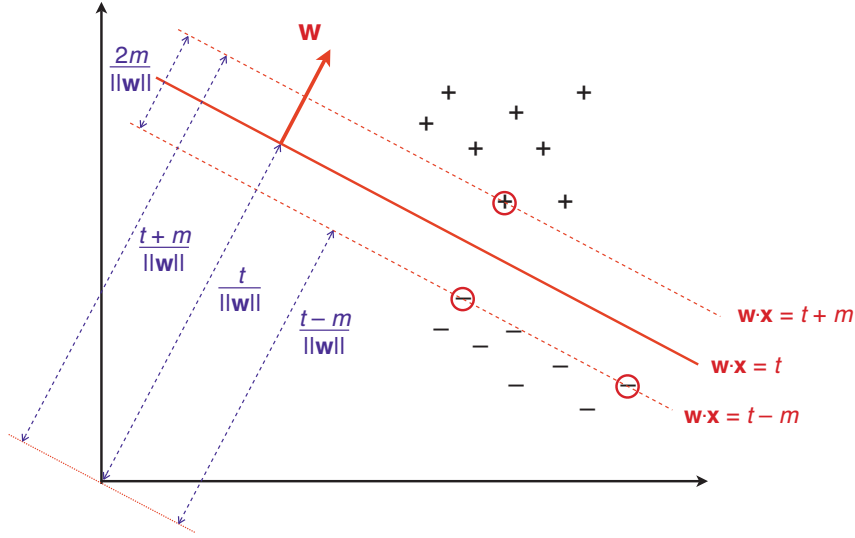


Figure 7.7. The geometry of a support vector classifier. The circled data points are the support vectors, which are the training examples nearest to the decision boundary. The support vector machine finds the decision boundary that maximises the margin $m/||\mathbf{w}||$.

measured along $\mathbf{w}$. Since we are free to rescale t , $||\mathbf{w}||$ and m , it is customary to choose $m = 1$. Maximising the margin then corresponds to minimising $||\mathbf{w}||$ or, more conveniently, $\frac{1}{2}||\mathbf{w}||^2$, provided of course that none of the training points fall inside the margin. This leads to a quadratic, constrained optimisation problem:

$$\mathbf{w}^*, t^* = \arg \min_{\mathbf{w}, t} \frac{1}{2} ||\mathbf{w}||^2 \quad \text{subject to } y_i(\mathbf{w} \cdot \mathbf{x}_i - t) \geq 1, 1 \leq i \leq n$$

We will approach this using the method of Lagrange multipliers (see [Background 7.3](#)). Adding the constraints with multipliers α_i for each training example gives the Lagrange function

$$\begin{aligned} \Lambda(\mathbf{w}, t, \alpha_1, \dots, \alpha_n) &= \frac{1}{2} ||\mathbf{w}||^2 - \sum_{i=1}^n \alpha_i (y_i(\mathbf{w} \cdot \mathbf{x}_i - t) - 1) \\ &= \frac{1}{2} ||\mathbf{w}||^2 - \sum_{i=1}^n \alpha_i y_i (\mathbf{w} \cdot \mathbf{x}_i) + \sum_{i=1}^n \alpha_i y_i t + \sum_{i=1}^n \alpha_i \\ &= \frac{1}{2} \mathbf{w} \cdot \mathbf{w} - \mathbf{w} \cdot \left(\sum_{i=1}^n \alpha_i y_i \mathbf{x}_i \right) + t \left(\sum_{i=1}^n \alpha_i y_i \right) + \sum_{i=1}^n \alpha_i \end{aligned}$$

While this looks like a formidable formula, some further analysis will allow us to derive the simpler dual form of the Lagrange function.

By taking the partial derivative of the Lagrange function with respect to t and setting it to 0 we find that for the optimal threshold t we have $\sum_{i=1}^n \alpha_i y_i = 0$. Similarly, by

Optimisation is a broad term denoting the problem of finding the best item or value among a set of alternatives. We have already seen a very simple, unconstrained form of optimisation in [Example 7.1](#) on [p.197](#), where we found the values of a and b minimising the sum of squared residuals $f(a, b) = \sum_{i=1}^n (w_i - (a + bh_i))^2$; this can be denoted as

$$a^*, b^* = \underset{a, b}{\operatorname{argmin}} f(a, b)$$

f is called the *objective function*; it can be linear, quadratic (as in this case), or more complex. We found the minimum of f by setting the partial derivatives of f with respect to a and b to 0, and solving for a and b ; the vector of these partial derivatives is called the *gradient* and denoted ∇f , so a succinct way of defining the unconstrained optimisation problem is: find a and b such that $\nabla f(a, b) = \mathbf{0}$. In this particular case the objective function is *convex*, which essentially means that there is a unique global minimum. This is, however, not always the case.

A *constrained optimisation* problem is one where the alternatives are subject to constraints, for instance

$$a^*, b^* = \underset{a, b}{\operatorname{argmin}} f(a, b) \quad \text{subject to } g(a, b) = c$$

If the relationship expressed by the constraint is linear, say $a - b = 0$, we can of course eliminate one of the variables and solve the simpler, unconstrained problem. However, this may not be possible if the constraints are non-linear. *Lagrange multipliers* are a powerful way of dealing with the general case. We form the Lagrange function defined by

$$\Lambda(a, b, \lambda) = f(a, b) - \lambda(g(a, b) - c)$$

where λ is the Lagrange multiplier, and solve the unconstrained problem $\nabla \Lambda(a, b, \lambda) = \mathbf{0}$. Since $\nabla_{a, b} \Lambda(a, b, \lambda) = \nabla f(a, b) - \lambda \nabla g(a, b)$ and $\nabla_{\lambda} \Lambda(a, b, \lambda) = g(a, b) - c$, this is a succinct way of requiring (i) that the gradients of f and g point in the same direction, and (ii) that the constraint is satisfied. We can include multiple equality constraints and also inequality constraints, each with their own Lagrange multiplier.

From the Lagrange function it is possible to derive a *dual* optimisation problem where we find the optimal values of the Lagrange multipliers. In general, the solution to the dual problem is only a lower bound on the solution to the *primal* problem, but under a set of conditions known as the *Karush–Kuhn–Tucker conditions* (KKT) the two solutions become equal. The quadratic optimisation problem posed by support vector machines is usually solved in its dual form.

Background 7.3. Basic concepts and terminology in mathematical optimisation.

taking the partial derivative of the Lagrange function with respect to $\mathbf{w}$ we see that the Lagrange multipliers define the weight vector as a linear combination of the training examples:

$$\frac{\partial}{\partial \mathbf{w}} \Lambda(\mathbf{w}, t, \alpha_1, \dots, \alpha_n) = \frac{\partial}{\partial \mathbf{w}} \frac{1}{2} \mathbf{w} \cdot \mathbf{w} - \frac{\partial}{\partial \mathbf{w}} \mathbf{w} \cdot \left(\sum_{i=1}^n \alpha_i y_i \mathbf{x}_i \right) = \mathbf{w} - \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i$$

Since this partial derivative is 0 for an optimal weight vector we conclude $\mathbf{w} = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i$ – the same expression as we derived for the perceptron in [Equation 7.9](#) on p.209. For the perceptron, the instance weights α_i are non-negative integers denoting the number of times an example has been misclassified in training. For a support vector machine, the α_i are non-negative reals. What they have in common is that, if $\alpha_i = 0$ for a particular example $\mathbf{x}_i$, that example could be removed from the training set without affecting the learned decision boundary. In the case of support vector machines this means that $\alpha_i > 0$ only for the support vectors: the training examples nearest to the decision boundary.

Now, by plugging the expressions $\sum_{i=1}^n \alpha_i y_i = 0$ and $\mathbf{w} = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i$ back into the Lagrangian we are able to eliminate $\mathbf{w}$ and t , and hence obtain the dual optimisation problem, which is entirely formulated in terms of the Lagrange multipliers:

$$\begin{aligned} \Lambda(\alpha_1, \dots, \alpha_n) &= -\frac{1}{2} \left(\sum_{i=1}^n \alpha_i y_i \mathbf{x}_i \right) \cdot \left(\sum_{i=1}^n \alpha_i y_i \mathbf{x}_i \right) + \sum_{i=1}^n \alpha_i \\ &= -\frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^n \alpha_i \end{aligned}$$

The dual problem is to maximise this function under positivity constraints and one equality constraint:

$$\begin{aligned} \alpha_1^*, \dots, \alpha_n^* &= \operatorname{argmax}_{\alpha_1, \dots, \alpha_n} -\frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^n \alpha_i \\ &\text{subject to } \alpha_i \geq 0, 1 \leq i \leq n \text{ and } \sum_{i=1}^n \alpha_i y_i = 0 \end{aligned}$$

The dual form of the optimisation problem for support vector machines illustrates two important points. First, it shows that searching for the maximum-margin decision boundary is equivalent to searching for the support vectors: they are the training examples with non-zero Lagrange multipliers, and through $\mathbf{w} = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i$ they completely determine the decision boundary. Secondly, it shows that the optimisation problem is entirely defined by pairwise dot products between training instances: the entries of the Gram matrix. As we shall see in [Section 7.5](#), this paves the way for a powerful adaptation of support vector machines that allows them to learn non-linear decision boundaries.

The following example makes these issues a bit more concrete by showing detailed calculations on some toy data.

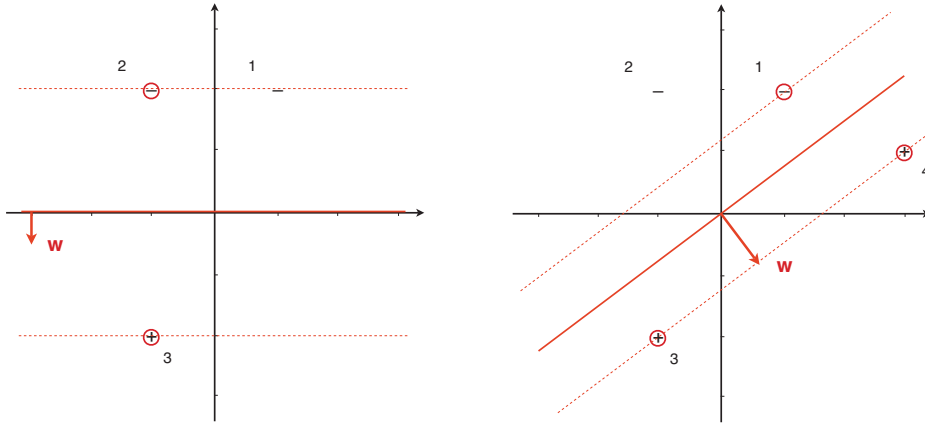


Figure 7.8. (left) A maximum-margin classifier built from three examples, with $\mathbf{w} = (0, -1/2)$ and margin 2. The circled examples are the support vectors: they receive non-zero Lagrange multipliers and define the decision boundary. **(right)** By adding a second positive the decision boundary is rotated to $\mathbf{w} = (3/5, -4/5)$ and the margin decreases to 1.

Example 7.5 (Two maximum-margin classifiers and their support vectors).

Let the data points and labels be as follows (see Figure 7.8 (left)):

$$\mathbf{X} = \begin{pmatrix} 1 & 2 \\ -1 & 2 \\ -1 & -2 \end{pmatrix} \quad \mathbf{y} = \begin{pmatrix} -1 \\ -1 \\ +1 \end{pmatrix} \quad \mathbf{X}' = \begin{pmatrix} -1 & -2 \\ 1 & -2 \\ -1 & -2 \end{pmatrix}$$

The matrix $\mathbf{X}'$ on the right incorporates the class labels; i.e., the rows are $y_i \mathbf{x}_i$. The Gram matrix is (without and with class labels):

$$\mathbf{X}\mathbf{X}^T = \begin{pmatrix} 5 & 3 & -5 \\ 3 & 5 & -3 \\ -5 & -3 & 5 \end{pmatrix} \quad \mathbf{X}'\mathbf{X}'^T = \begin{pmatrix} 5 & 3 & 5 \\ 3 & 5 & 3 \\ 5 & 3 & 5 \end{pmatrix}$$

The dual optimisation problem is thus

$$\begin{aligned} & \arg \max_{\alpha_1, \alpha_2, \alpha_3} -\frac{1}{2} (5\alpha_1^2 + 3\alpha_1\alpha_2 + 5\alpha_1\alpha_3 + 3\alpha_2\alpha_1 + 5\alpha_2^2 + 3\alpha_2\alpha_3 + 5\alpha_3\alpha_1 \\ & \quad \quad \quad + 3\alpha_3\alpha_2 + 5\alpha_3^2) + \alpha_1 + \alpha_2 + \alpha_3 \\ & = \arg \max_{\alpha_1, \alpha_2, \alpha_3} -\frac{1}{2} (5\alpha_1^2 + 6\alpha_1\alpha_2 + 10\alpha_1\alpha_3 + 5\alpha_2^2 + 6\alpha_2\alpha_3 + 5\alpha_3^2) + \alpha_1 + \alpha_2 + \alpha_3 \end{aligned}$$

subject to $\alpha_1 \geq 0, \alpha_2 \geq 0, \alpha_3 \geq 0$ and $-\alpha_1 - \alpha_2 + \alpha_3 = 0$. While in practice such problems are solved by dedicated quadratic optimisation solvers, here we will show how to solve this toy problem by hand.

Using the equality constraint we can eliminate one of the variables, say α_3 , and simplify the objective function to

$$\begin{aligned} \arg \max_{\alpha_1, \alpha_2, \alpha_3} & -\frac{1}{2} (5\alpha_1^2 + 6\alpha_1\alpha_2 + 10\alpha_1(\alpha_1 + \alpha_2) + 5\alpha_2^2 + 6\alpha_2(\alpha_1 + \alpha_2) + 5(\alpha_1 + \alpha_2)^2) \\ & + 2\alpha_1 + 2\alpha_2 \\ & = \arg \max_{\alpha_1, \alpha_2, \alpha_3} -\frac{1}{2} (20\alpha_1^2 + 32\alpha_1\alpha_2 + 16\alpha_2^2) + 2\alpha_1 + 2\alpha_2 \end{aligned}$$

Setting partial derivatives to 0 we obtain $-20\alpha_1 - 16\alpha_2 + 2 = 0$ and $-16\alpha_1 - 16\alpha_2 + 2 = 0$ (notice that, because the objective function is quadratic, these equations are guaranteed to be linear). We therefore obtain the solution $\alpha_1 = 0$ and $\alpha_2 = \alpha_3 = 1/8$. We then have $\mathbf{w} = 1/8(\mathbf{x}_3 - \mathbf{x}_2) = \begin{pmatrix} 0 \\ -1/2 \end{pmatrix}$, resulting in a margin of $1/||\mathbf{w}|| = 2$. Finally, t can be obtained from any support vector, say $\mathbf{x}_2$, since $y_2(\mathbf{w} \cdot \mathbf{x}_2 - t) = 1$; this gives $-1 \cdot (-1 - t) = 1$, hence $t = 0$. The resulting maximum-margin classifier is depicted in Figure 7.8 (left). Notice that the first example $\mathbf{x}_1$ is not a support vector, even though it is on the margin: this is because removing it will not affect the decision boundary.

We now add an additional positive at (3, 1). This gives the following data matrices:

$$\mathbf{X}' = \begin{pmatrix} -1 & -2 \\ 1 & -2 \\ -1 & -2 \\ 3 & 1 \end{pmatrix} \quad \mathbf{X}'\mathbf{X}'^T = \begin{pmatrix} 5 & 3 & 5 & -5 \\ 3 & 5 & 3 & 1 \\ 5 & 3 & 5 & -5 \\ -5 & 1 & -5 & 10 \end{pmatrix}$$

It can be verified by similar calculations to those above that the margin decreases to 1 and the decision boundary rotates to $\mathbf{w} = \begin{pmatrix} 3/5 \\ -4/5 \end{pmatrix}$ (Figure 7.8 (right)). The Lagrange multipliers now are $\alpha_1 = 1/2$, $\alpha_2 = 0$, $\alpha_3 = 1/10$ and $\alpha_4 = 2/5$. Thus, only $\mathbf{x}_3$ is a support vector in both the original and the extended data set.

Soft margin SVM

If the data is not linearly separable, then the constraints $\mathbf{w} \cdot \mathbf{x}_i - t \geq 1$ posed by the examples are not jointly satisfiable. However, there is a very elegant way of adapting the optimisation problem such that it admits a solution even in this case. The idea is to introduce *slack variables* ξ_i , one for each example, which allow some of them to be inside the margin or even at the wrong side of the decision boundary – we will call these *margin errors*. Thus, we change the constraints to $\mathbf{w} \cdot \mathbf{x}_i - t \geq 1 - \xi_i$ and add the sum of

all slack variables to the objective function to be minimised, resulting in the following *soft margin* optimisation problem:

$$\begin{aligned} \mathbf{w}^*, t^*, \xi_i^* = \arg \min_{\mathbf{w}, t, \xi_i} & \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n \xi_i \\ \text{subject to } & y_i (\mathbf{w} \cdot \mathbf{x}_i - t) \geq 1 - \xi_i \text{ and } \xi_i \geq 0, 1 \leq i \leq n \end{aligned} \quad (7.11)$$

C is a user-defined parameter trading off margin maximisation against slack variable minimisation: a high value of C means that margin errors incur a high penalty, while a low value permits more margin errors (possibly including misclassifications) in order to achieve a large margin. If we allow more margin errors we need fewer support vectors, hence C controls to some extent the ‘complexity’ of the SVM and hence is often referred to as the *complexity parameter*. It can be seen as a form of regularisation similar to that discussed in the context of least-squares regression.

The Lagrange function is then as follows:

$$\begin{aligned} \Lambda(\mathbf{w}, t, \xi_i, \alpha_i, \beta_i) &= \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n \xi_i - \sum_{i=1}^n \alpha_i (y_i (\mathbf{w} \cdot \mathbf{x}_i - t) - (1 - \xi_i)) - \sum_{i=1}^n \beta_i \xi_i \\ &= \frac{1}{2} \mathbf{w} \cdot \mathbf{w} - \mathbf{w} \cdot \left(\sum_{i=1}^n \alpha_i y_i \mathbf{x}_i \right) + t \left(\sum_{i=1}^n \alpha_i y_i \right) + \sum_{i=1}^n \alpha_i + \sum_{i=1}^n (C - \alpha_i - \beta_i) \xi_i \\ &= \Lambda(\mathbf{w}, t, \alpha_i) + \sum_{i=1}^n (C - \alpha_i - \beta_i) \xi_i \end{aligned}$$

For an optimal solution every partial derivative with respect to ξ_i should be 0, from which it follows that $C - \alpha_i - \beta_i = 0$ for all i , and hence the added term vanishes from the dual problem. Furthermore, since both α_i and β_i are positive, this means that α_i cannot be larger than C , which manifests itself as an additional upper bound on α_i in the dual problem:

$$\begin{aligned} \alpha_1^*, \dots, \alpha_n^* = \arg \max_{\alpha_1, \dots, \alpha_n} & -\frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j + \sum_{i=1}^n \alpha_i \\ \text{subject to } & 0 \leq \alpha_i \leq C \text{ and } \sum_{i=1}^n \alpha_i y_i = 0 \end{aligned} \quad (7.12)$$

This is a remarkable and beautiful result. It follows from the particular way that slack variables were added to the optimisation problem in Equation 7.11. By restricting the slack variables to be positive and adding them to the objective function to be minimised, they function as penalty terms, measuring deviations on the wrong side of the margin only. Furthermore, the fact that the β_i multipliers do not appear in the dual objective follows from the fact that the penalty term in the primal objective is linear in ξ_i . In effect, these slack variables implement what was called *hinge loss* in Figure 2.6 on p.63: a margin $z > 1$ incurs no penalty, and a margin $z = 1 - \xi \leq 1$ incurs a penalty $\xi = 1 - z$.

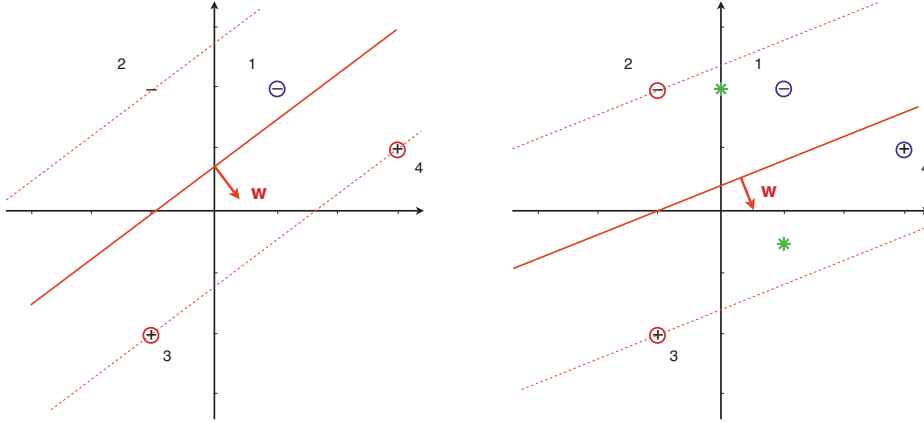


Figure 7.9. (left) The soft margin classifier learned with $C = 5/16$, at which point $\mathbf{x}_2$ is about to become a support vector. **(right)** The soft margin classifier learned with $C = 1/10$: all examples contribute equally to the weight vector. The asterisks denote the class means, and the decision boundary is parallel to the one learned by the basic linear classifier.

What is the significance of the upper bound C on the α_i multipliers? Since $C - \alpha_i - \beta_i = 0$ for all i , $\alpha_i = C$ implies $\beta_i = 0$. The β_i multipliers come from the $\xi_i \geq 0$ constraint, and a multiplier of 0 means that the lower bound is not reached, i.e., $\xi_i > 0$ (analogous to the fact that $\alpha_j = 0$ means that $\mathbf{x}_j$ is not a support vector and hence $\mathbf{w} \cdot \mathbf{x}_j - t > 1$). In other words, a solution to the soft margin optimisation problem in dual form divides the training examples into three cases:

- $\alpha_i = 0$ these are outside or on the margin;
- $0 < \alpha_i < C$ these are the support vectors on the margin;
- $\alpha_i = C$ these are on or inside the margin.

Notice that we still have $\mathbf{w} = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i$, and so both second and third case examples participate in spanning the decision boundary.

Example 7.6 (Soft margins). We continue [Example 7.5](#), where we saw that adding the positive example $\mathbf{x}_4 = (3, 1)$ to the first three examples significantly reduced the margin from 2 to 1. We will now show that soft margin classifiers with larger margins are learned with sufficiently large complexity parameter C .

Recall that the Lagrange multipliers for the classifier in [Figure 7.8 \(right\)](#) are $\alpha_1 = 1/2$, $\alpha_2 = 0$, $\alpha_3 = 1/10$ and $\alpha_4 = 2/5$. So α_1 is the largest multiplier, and as long as $C > \alpha_1 = 1/2$ no margin errors are tolerated. For $C = 1/2$ we have $\alpha_1 = C$,

and hence for $C < 1/2$ we have that $\mathbf{x}_1$ becomes a margin error and the optimal classifier is a soft margin classifier. Effectively, with decreasing C the decision boundary and the upper margin move upward, while the lower margin stays the same.

The upper margin reaches $\mathbf{x}_2$ for $C = 5/16$ (Figure 7.9 (left)), at which point we have $\mathbf{w} = \begin{pmatrix} 3/8 \\ -1/2 \end{pmatrix}$, $t = 3/8$ and the margin has increased to 1.6. Furthermore, we have $\xi_1 = 6/8$, $\alpha_1 = C = 5/16$, $\alpha_2 = 0$, $\alpha_3 = 1/16$ and $\alpha_4 = 1/4$.

If we now decrease C further, the decision boundary starts to rotate clockwise, so that $\mathbf{x}_4$ becomes a margin error as well, and only $\mathbf{x}_2$ and $\mathbf{x}_3$ are support vectors. The boundary rotates until $C = 1/10$, at which point we have $\mathbf{w} = \begin{pmatrix} 1/5 \\ -1/2 \end{pmatrix}$, $t = 1/5$ and the margin has increased to 1.86. Furthermore, we have $\xi_1 = 4/10$ and $\xi_4 = 7/10$, and all multipliers have become equal to C (Figure 7.9 (right)).

Finally, when C decreases further the decision boundary stays where it is, but the norm of the weight vector gradually decreases and all points become margin errors.

Example 7.6 illustrates an important point: for low enough C , all examples receive the same multiplier C , and hence we have $\mathbf{w} = C \sum_{i=1}^n y_i \mathbf{x}_i = C(\text{Pos} \cdot \boldsymbol{\mu}^{\oplus} - \text{Neg} \cdot \boldsymbol{\mu}^{\ominus})$, where $\boldsymbol{\mu}^{\oplus}$ and $\boldsymbol{\mu}^{\ominus}$ are the means of the positive and negative examples, respectively. In other words, *a minimal-complexity soft margin classifier summarises the classes by their class means in a way very similar to the basic linear classifier*. For intermediate values of C the decision boundary is spanned by the support vectors and the per-class means of the margin errors.

In summary, support vector machines are linear classifiers that construct the unique decision boundary that maximises the distance to the nearest training examples (the support vectors). The complexity parameter C can be used to adjust the number and severity of allowed margin violations. Training an SVM involves solving a large quadratic optimisation problem and is usually best left to a dedicated numerical solver.

7.4 Obtaining probabilities from linear classifiers

As we have seen, a linear classifier produces scores $\hat{s}(\mathbf{x}_i) = \mathbf{w} \cdot \mathbf{x}_i - t$ that are thresholded at 0 in order to classify examples. Owing to the geometric nature of linear classifiers, such scores can be used to obtain the (signed) distance of $\mathbf{x}_i$ from the decision boundary. To see this, notice that the length of the projection of $\mathbf{x}_i$ onto $\mathbf{w}$ is $\|\mathbf{x}_i\| \cos \theta$, where

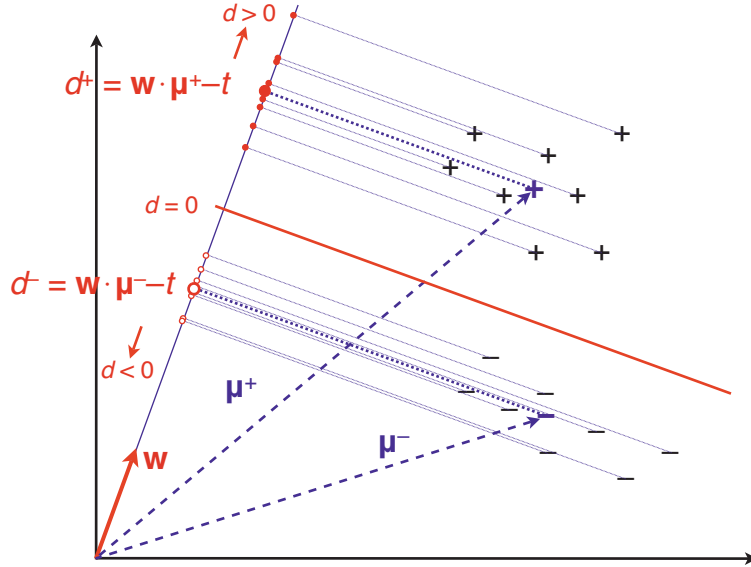


Figure 7.10. We can think of a linear classifier as a projection onto the direction given by $\mathbf{w}$, here assumed to be a unit vector. $\mathbf{w} \cdot \mathbf{x} - t$ gives the signed distance from the decision boundary on the projection line. Also indicated are the class means μ^+ and μ^- , and the corresponding mean distances d^+ and d^- .

θ is the angle between $\mathbf{x}_i$ and $\mathbf{w}$. Since $\mathbf{w} \cdot \mathbf{x}_i = \|\mathbf{w}\| \|\mathbf{x}_i\| \cos \theta$, we can write this length as $(\mathbf{w} \cdot \mathbf{x}_i) / \|\mathbf{w}\|$. This gives the following signed distance:

$$d(\mathbf{x}_i) = \frac{\hat{s}(\mathbf{x}_i)}{\|\mathbf{w}\|} = \frac{\mathbf{w} \cdot \mathbf{x}_i - t}{\|\mathbf{w}\|} = \mathbf{w}' \cdot \mathbf{x}_i - t'$$

with $\mathbf{w}' = \mathbf{w} / \|\mathbf{w}\|$ rescaled to unit length and $t' = t / \|\mathbf{w}\|$ the corresponding rescaled intercept. The sign of this quantity tells us which side of the decision boundary we are on: positive distances for points on the ‘positive’ side of the decision boundary (the direction in which $\mathbf{w}$ points) and negative distances on the other side (Figure 7.10).

This geometric interpretation of the scores produced by linear classifiers offers an interesting possibility for turning them into probabilities, a process that was called *calibration* in Section 2.3. Let $\bar{d}^+$ denote the mean distance of the positive examples to the decision boundary: i.e., $\bar{d}^+ = \mathbf{w} \cdot \mu^+ - t$, where μ^+ is the mean of the positive examples and $\mathbf{w}$ is unit length (although the latter assumption is not strictly necessary, as it will turn out that the weight vector will be rescaled). It would not be unreasonable to expect that the distance of positive examples to the decision boundary is normally distributed around this mean:² that is, when plotting a histogram of these distances,

²For instance, with sufficiently many examples this could be justified by the *central limit theorem*: the sum of a large number of identically distributed independent random variables is approximately normally distributed.

we would expect the familiar bell curve to appear. Under this assumption, the probability density function of d is $P(d|\oplus) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(d-\bar{d}^\oplus)^2}{2\sigma^2}\right)$ (see [Background 9.1](#) on p.267 if you need to remind yourself about the normal distribution). Similarly, the distances of negative examples to the decision boundary can be expected to be normally distributed around $\bar{d}^\ominus = \mathbf{w} \cdot \boldsymbol{\mu}^\ominus - t$, with $\bar{d}^\ominus < 0 < \bar{d}^\oplus$. We will assume that both normal distributions have the same variance σ^2 .

Suppose we now observe a point $\mathbf{x}$ with distance $d(\mathbf{x})$. We classify this point as positive if $d(\mathbf{x}) > 0$ and as negative if $d(\mathbf{x}) < 0$, but we want to attach a probability $\hat{p}(\mathbf{x}) = P(\oplus|d(\mathbf{x}))$ to these predictions. Using Bayes' rule we obtain

$$P(\oplus|d(\mathbf{x})) = \frac{P(d(\mathbf{x})|\oplus)P(\oplus)}{P(d(\mathbf{x})|\oplus)P(\oplus) + P(d(\mathbf{x})|\ominus)P(\ominus)} = \frac{LR}{LR + 1/clr}$$

where LR is the likelihood ratio obtained from the normal score distributions, and clr is the class ratio. We will assume for simplicity that $clr = 1$ in the derivation below. Furthermore, assume for now that $\sigma^2 = 1$ and $\bar{d}^\oplus = -\bar{d}^\ominus = 1/2$ (we will relax this in a moment). We then have

$$\begin{aligned} LR &= \frac{P(d(\mathbf{x})|\oplus)}{P(d(\mathbf{x})|\ominus)} = \frac{\exp(-(d(\mathbf{x}) - 1/2)^2/2)}{\exp(-(d(\mathbf{x}) + 1/2)^2/2)} \\ &= \exp(-(d(\mathbf{x}) - 1/2)^2/2 + (d(\mathbf{x}) + 1/2)^2/2) = \exp(d(\mathbf{x})) \end{aligned}$$

and so

$$P(\oplus|d(\mathbf{x})) = \frac{\exp(d(\mathbf{x}))}{\exp(d(\mathbf{x})) + 1} = \frac{\exp(\mathbf{w} \cdot \mathbf{x} - t)}{\exp(\mathbf{w} \cdot \mathbf{x} - t) + 1}$$

So, in order to obtain probability estimates from a linear classifier outputting distance scores d , we convert d into a probability by means of the mapping $d \mapsto \frac{\exp(d)}{\exp(d)+1}$ (or, equivalently, $d \mapsto \frac{1}{1+\exp(-d)}$). This S-shaped or *sigmoid* function is called the *logistic function*; it finds applications in a wide range of areas ([Figure 7.11](#)).

Suppose now that $\bar{d}^\oplus = -\bar{d}^\ominus$ as before, but we do not assume anything about the magnitude of these mean distances or of σ^2 . In this case we have

$$\begin{aligned} LR &= \exp\left(\frac{-(d(\mathbf{x}) - \bar{d}^\oplus)^2 + (d(\mathbf{x}) - \bar{d}^\ominus)^2}{2\sigma^2}\right) \\ &= \exp\left(\frac{2\bar{d}^\oplus d(\mathbf{x}) - (\bar{d}^\oplus)^2 - 2\bar{d}^\ominus d(\mathbf{x}) + (\bar{d}^\ominus)^2}{2\sigma^2}\right) = \exp(\gamma d(\mathbf{x})) \end{aligned}$$

with $a = (\bar{d}^\oplus - \bar{d}^\ominus)/\sigma^2$ a scaling factor that rescales the weight vector so that the mean distances per class are one unit of variance apart. In other words, by taking the scaling factor γ into account, we can drop our assumption that $\mathbf{w}$ is a unit vector.

If we also drop the assumption that $\bar{d}^\oplus$ and $\bar{d}^\ominus$ are symmetric around the decision

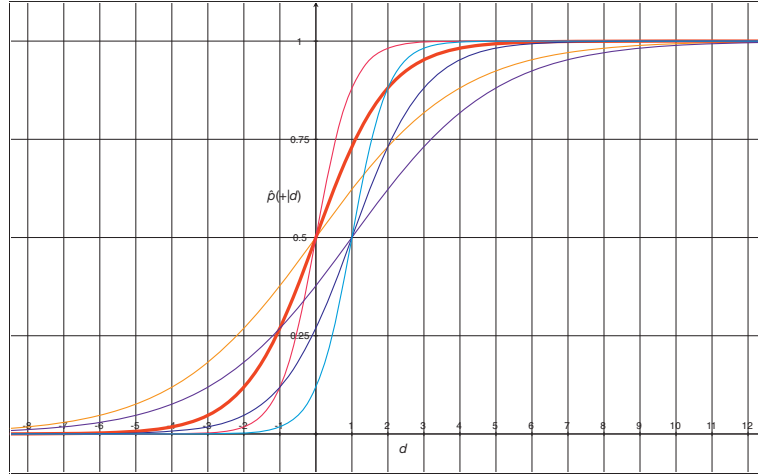


Figure 7.11. The logistic function, a useful function for mapping distances from a linear decision boundary into an estimate of the positive posterior probability. The **fat red line** indicates the standard logistic function $\hat{p}(d) = \frac{1}{1+\exp(-d)}$; this function can be used to obtain probability estimates if the two classes are equally prevalent and the class means are equidistant from the decision boundary and one unit of variance apart. The steeper and flatter **red lines** show how the function changes if the class means are 2 and 1/2 units of variance apart, respectively. The three **blue lines** show how these curves change if $d_0 = 1$, which means that the positives are on average further away from the decision boundary.

boundary, then we obtain the most general form

$$LR = \frac{P(d(\mathbf{x})|\oplus)}{P(d(\mathbf{x})|\ominus)} = \exp(\gamma(d(\mathbf{x}) - d_0)) \quad (7.13)$$

$$\gamma = \frac{\bar{d}^{\oplus} - \bar{d}^{\ominus}}{\sigma^2} = \frac{\mathbf{w} \cdot (\boldsymbol{\mu}^{\oplus} - \boldsymbol{\mu}^{\ominus})}{\sigma^2}, \quad d_0 = \frac{\bar{d}^{\oplus} + \bar{d}^{\ominus}}{2} = \frac{\mathbf{w} \cdot (\boldsymbol{\mu}^{\oplus} + \boldsymbol{\mu}^{\ominus})}{2} - t$$

d_0 has the effect of moving the decision boundary from $\mathbf{w} \cdot \mathbf{x} = t$ to $\mathbf{x} = (\boldsymbol{\mu}^{\oplus} + \boldsymbol{\mu}^{\ominus})/2$, that is, halfway between the two class means. The logistic mapping thus becomes $d \mapsto \frac{1}{1+\exp(-\gamma(d-d_0))}$, and the effect of the two parameters is visualised in **Figure 7.11**.

Example 7.7 (Logistic calibration of a linear classifier). Logistic calibration has a particularly simple form for the basic linear classifier, which has $\mathbf{w} = \boldsymbol{\mu}^{\oplus} - \boldsymbol{\mu}^{\ominus}$. It follows that

$$\bar{d}^{\oplus} - \bar{d}^{\ominus} = \frac{\mathbf{w} \cdot (\boldsymbol{\mu}^{\oplus} - \boldsymbol{\mu}^{\ominus})}{\|\mathbf{w}\|} = \frac{\|\boldsymbol{\mu}^{\oplus} - \boldsymbol{\mu}^{\ominus}\|^2}{\|\boldsymbol{\mu}^{\oplus} - \boldsymbol{\mu}^{\ominus}\|} = \|\boldsymbol{\mu}^{\oplus} - \boldsymbol{\mu}^{\ominus}\|$$

and hence $\gamma = \|\boldsymbol{\mu}^{\oplus} - \boldsymbol{\mu}^{\ominus}\|/\sigma^2$. Furthermore, $d_0 = 0$ as $(\boldsymbol{\mu}^{\oplus} + \boldsymbol{\mu}^{\ominus})/2$ is already on the decision boundary. So in this case logistic calibration does not move the

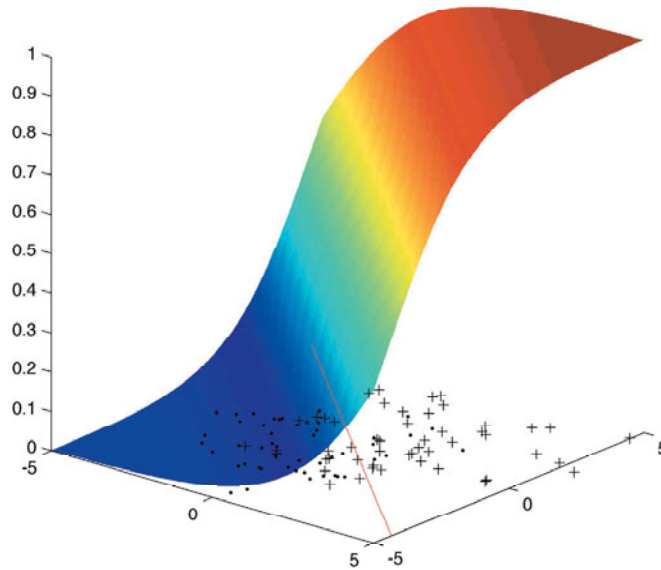


Figure 7.12. The surface shows the sigmoidal probability estimates resulting from logistic calibration of the basic linear classifier on random data satisfying the assumptions of logistic calibration.

decision boundary, and only adjusts the steepness of the sigmoid according to the separation of the classes. Figure 7.12 illustrates this for some data sampled from two normal distributions with the same diagonal covariance matrix.

To summarise: in order to get calibrated probability estimates out of a linear classifier, we first calculate the mean distances $\bar{d}^+$ and $\bar{d}^-$ and the variance σ^2 , and from those the location parameter d_0 and the scaling parameter γ . The likelihood ratio is then $LR = \exp(\gamma(d(\mathbf{x}) - d_0)) = \exp(\gamma(\mathbf{w} \cdot \mathbf{x} - t - d_0))$. Since the logarithm of the likelihood ratio is linear in $\mathbf{x}$, such models are called *log-linear models*. Notice that $\gamma(\mathbf{w} \cdot \mathbf{x} - t - d_0) = \mathbf{w}' \cdot \mathbf{x} - t'$ with $\mathbf{w}' = \gamma\mathbf{w}$ and $t' = \gamma(t + d_0)$. This means that the logistic calibration procedure can change the location of the decision boundary but not its direction. However, there may be an alternative weight vector with a different direction that assigns a higher likelihood to the data. Finding the maximum-likelihood linear classifier using the logistic model is called *logistic regression*, and will be discussed in Section 9.3.

As an alternative to logistic calibration, we can also use the isotonic calibration

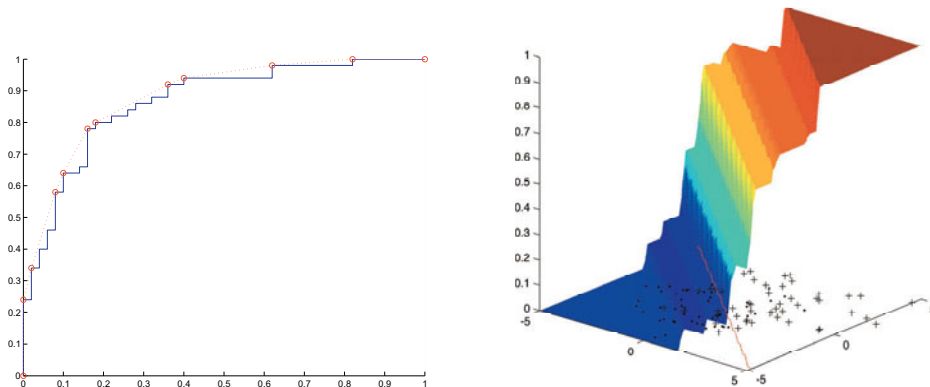


Figure 7.13. (left) ROC curve and convex hull of the same model and data as in Figure 7.12. (right) The convex hull can be used as a non-parametric calibration method. Each segment of the convex hull corresponds to a plateau of the probability surface.

method discussed in Section 2.3. Figure 7.13 (left) shows the ROC curve of the basic linear classifier on the data in Figure 7.12 as well as its convex hull. We can then construct a piecewise linear calibration function with plateaus corresponding to the convex hull segments, as shown in Figure 7.13 (right). In contrast with the logistic method this calibration method is non-parametric and hence does not make any assumptions about the data. In order to avoid overfitting, non-parametric methods typically need more data than parametric methods. It is interesting to note that no grading takes place on the plateaus, which are rather similar to the segments of a grouping model. In other words, convex hull calibration can potentially produce a hybrid between grouping and grading models.

7.5 Going beyond linearity with kernel methods

In this chapter we have looked at linear methods for classification and regression. Starting with the least-squares method for regression, we have seen how to adapt it to binary classification, resulting in a version of the basic linear classifier that takes feature correlation into account by constructing the matrix $(\mathbf{X}^T \mathbf{X})^{-1}$ and is sensitive to unequal class distributions. We then looked at the heuristic perceptron algorithm for linearly separable data, and the support vector machine which finds the unique decision boundary with maximum margin and which can be adapted to non-separable data. In this section we show that these techniques can be adapted to learn non-linear decision boundaries. The main idea is simple (and was already explored in Example 1.9 on p.43): to transform the data non-linearly to a feature space in which linear classification can be applied.

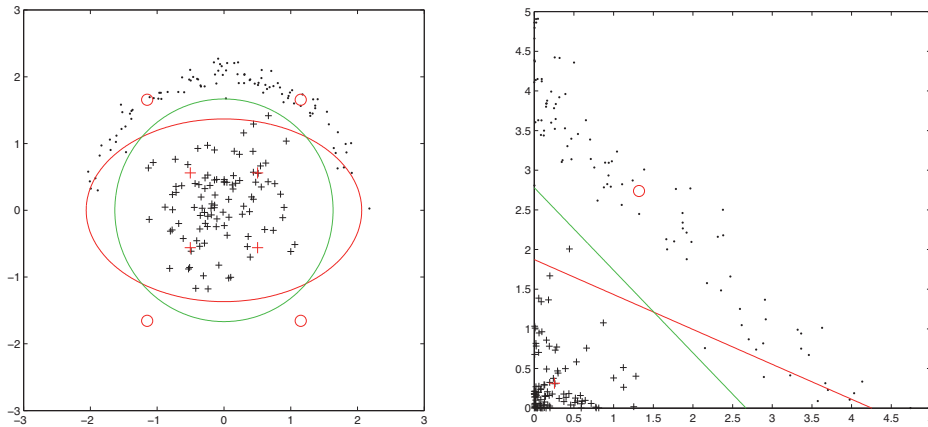


Figure 7.14. (left) Decision boundaries learned by the **basic linear classifier** and the **perceptron** using the square of the features. (right) Data and decision boundaries in the transformed feature space.

Example 7.8 (Learning a quadratic decision boundary). The data in Figure 7.14 (left) is not linearly separable, but both classes have a clear circular shape. Figure 7.14 (right) shows the same data with the feature values squared. In this transformed feature space the data has become linearly separable, and the perceptron is able to separate the classes. The resulting decision boundary in the original space is a near-circle. Also shown is the decision boundary learned by the basic linear classifier in the quadratic feature space, corresponding to an ellipse in the original space.

In general, mapping points back from the feature space to the instance space is non-trivial. E.g., in this example each class mean in feature space maps back to four points in the original space, owing to the quadratic mapping.

It is customary to call the transformed space the *feature space* and the original space the *input space*. The approach thus appears to be to transform the training data to feature space and learn a model there. In order to classify new data we transform that to feature space as well and apply the model. However, the remarkable thing is that in many cases the feature space does not have to be explicitly constructed, as we can perform all necessary operations in input space.

Take the perceptron algorithm in dual form, for example (Algorithm 7.2 on p.209). The algorithm is a simple counting algorithm – the only operation that is somewhat involved is testing whether example $\mathbf{x}_i$ is correctly classified by evaluating $y_i \sum_{j=1}^{|D|} \alpha_j y_j \mathbf{x}_i$.

$\mathbf{x}_j$. The key component of this calculation is the dot product $\mathbf{x}_i \cdot \mathbf{x}_j$. Assuming bivariate examples $\mathbf{x}_i = (x_i, y_i)$ and $\mathbf{x}_j = (x_j, y_j)$ for notational simplicity, the dot product can be written as $\mathbf{x}_i \cdot \mathbf{x}_j = x_i x_j + y_i y_j$. The corresponding instances in the quadratic feature space are (x_i^2, y_i^2) and (x_j^2, y_j^2) , and their dot product is

$$(x_i^2, y_i^2) \cdot (x_j^2, y_j^2) = x_i^2 x_j^2 + y_i^2 y_j^2$$

This is almost equal to

$$(\mathbf{x}_i \cdot \mathbf{x}_j)^2 = (x_i x_j + y_i y_j)^2 = (x_i x_j)^2 + (y_i y_j)^2 + 2x_i x_j y_i y_j$$

but not quite because of the third term of cross-products. We can capture this term by extending the feature vector with a third feature $\sqrt{2}x_i y_i$. This gives the following feature space:

$$\begin{aligned} \phi(\mathbf{x}_i) &= (x_i^2, y_i^2, \sqrt{2}x_i y_i) & \phi(\mathbf{x}_j) &= (x_j^2, y_j^2, \sqrt{2}x_j y_j) \\ \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j) &= x_i^2 x_j^2 + y_i^2 y_j^2 + 2x_i x_j y_i y_j = (\mathbf{x}_i \cdot \mathbf{x}_j)^2 \end{aligned}$$

We now define $\kappa(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^2$, and replace $\mathbf{x}_i \cdot \mathbf{x}_j$ with $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ in the dual perceptron algorithm to obtain the *kernel perceptron* (Algorithm 7.4), which is able to learn the kind of non-linear decision boundaries illustrated in Example 7.8.

The introduction of kernels opens up a whole range of possibilities. Clearly we can define a polynomial kernel of any degree p as $\kappa(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^p$. This transforms

Algorithm 7.4: *KernelPerceptron*(D, κ) – perceptron training algorithm using a kernel.

Input : labelled training data D in homogeneous coordinates;
 kernel function κ .

Output : coefficients α_i defining non-linear decision boundary.

```

1  $\alpha_i \leftarrow 0$  for  $1 \leq i \leq |D|$ ;
2 converged  $\leftarrow$  false;
3 while converged = false do
4   converged  $\leftarrow$  true;
5   for  $i = 1$  to  $|D|$  do
6     if  $y_i \sum_{j=1}^{|D|} \alpha_j y_j \kappa(\mathbf{x}_i, \mathbf{x}_j) \leq 0$  then
7        $\alpha_i \leftarrow \alpha_i + 1$ ;
8       converged  $\leftarrow$  false;
9     end
10  end
11 end
```

a d -dimensional input space into a high-dimensional feature space, such that each new feature is a product of p terms (possibly repeated). If we include a constant, say $\kappa(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j + 1)^p$, we would get all lower-order terms as well. So, for example, in a bivariate input space and setting $p = 2$ the resulting feature space is

$$\phi(\mathbf{x}) = (x^2, y^2, \sqrt{2}xy, \sqrt{2}x, \sqrt{2}y, 1)$$

with linear as well as quadratic features.

But we are not restricted to polynomial kernels. An often-used kernel is the *Gaussian kernel*, defined as

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \exp\left(\frac{-\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}\right) \quad (7.14)$$

where σ is a parameter known as the *bandwidth*. To understand the Gaussian kernel a bit better, notice that $\kappa(\mathbf{x}, \mathbf{x}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}) = \|\phi(\mathbf{x})\|^2$ for any kernel obeying a number of standard properties referred to as ‘positive semi-definiteness’. In this case we have $\kappa(\mathbf{x}, \mathbf{x}) = 1$, which means that all points $\phi(\mathbf{x})$ lie on a hypersphere around the feature space origin – which is however of infinite dimension, so geometric considerations don’t help us much here. It is more helpful to think of a Gaussian kernel as imposing a Gaussian (i.e., multivariate normal, see [Background 9.1](#) on p.267) surface on each support vector in instance space, so that the decision boundary is defined in terms of those Gaussian surfaces.

Kernel methods are best known in combination with support vector machines. Notice that the soft margin optimisation problem ([Equation 7.12](#) on p.217) is defined in terms of dot products between training instances and hence the ‘kernel trick’ can be applied:

$$\begin{aligned} \alpha_1^*, \dots, \alpha_n^* = \operatorname{argmax}_{\alpha_1, \dots, \alpha_n} & -\frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \alpha_j y_i y_j \kappa(\mathbf{x}_i, \mathbf{x}_j) + \sum_{i=1}^n \alpha_i \\ & \text{subject to } 0 \leq \alpha_i \leq C \text{ and } \sum_{i=1}^n \alpha_i y_i = 0 \end{aligned}$$

One thing to keep in mind is that the decision boundary learned with a non-linear kernel cannot be represented by a simple weight vector in input space. Thus, in order to classify a new example $\mathbf{x}$ we need to evaluate $y_i \sum_{j=1}^n \alpha_j y_j \kappa(\mathbf{x}, \mathbf{x}_j)$ which is an $O(n)$ computation involving all training examples, or at least the ones with non-zero multipliers α_j . This is why support vector machines are a popular choice as a kernel method, since they naturally promote sparsity in the support vectors. Although we have restricted attention to numerical features here, it is worth stressing that kernels can be defined over discrete structures, including trees, graphs, and logical formulae, and thus open the way to extending geometric models to non-numerical data.

7.6 Linear models: Summary and further reading

After considering logical models in the previous three chapters we had a good look at linear models in this chapter. Logical models are inherently non-numerical, and so deal with numerical features by using thresholds to convert them into two or more intervals. Linear models are almost diametrically opposite in that they can deal with numerical features directly but need to pre-process non-numerical features.³ Geometrically, linear models use lines and planes to build the model, which essentially means that a certain increase or decrease in one of the features has the same effect, regardless of that feature's value or any of the other features. They are simple and robust to variations in the training data, but sometimes suffer from underfitting as a consequence.

☞ In Section 7.1 we considered the least-squares method that was originally conceived to solve a regression problem. This classical method, which derives its name from minimising the sum of squared residuals between predicted and actual function values, is described in innumerable introductory mathematics and engineering texts (and was one of the example programs I remember running on my father's Texas Instruments TI-58 programmable calculator). We first had a look at the problem in univariate form, and then derived the general solution as $\hat{\mathbf{w}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$, where $(\mathbf{X}^T \mathbf{X})^{-1}$ is a transformation that decorrelates, centres and normalises the features. We then discussed regularised versions of linear regression: ridge regression was introduced by [Hoerl and Kennard \(1970\)](#), and the lasso which naturally leads to sparse solutions was introduced by [Tibshirani \(1996\)](#). We saw how the least-squares method could be applied to binary classification by encoding the classes by +1 and -1, leading to the solution $\hat{\mathbf{w}} = (\mathbf{X}^T \mathbf{X})^{-1} (\text{Pos } \mu^{\oplus} - \text{Neg } \mu^{\ominus})$. This generalises the basic linear classifier by taking feature correlation and unequal class prevalence into account, but at a considerably increased computational cost (quadratic in the number of instances and cubic in the number of features).

☞ Section 7.2 presented another classical linear model, the perceptron. Unlike the least-squares method, which always finds the optimal solution in terms of sum of squared residuals, the perceptron is a heuristic algorithm that depends, for one thing, on the order in which the examples are presented. Invented by [Rosenblatt \(1958\)](#), its convergence for linearly separable data was proved by [Novikoff \(1962\)](#), who gave an upper bound on the number of mistakes made before the perceptron converged. [Minsky and Papert \(1969\)](#) proved further formal properties of the perceptron, but also demonstrated the limitations of a linear classifier. These were overcome with the development, over an extended period of time and with contributions from many people, of the multilayer perceptron and its

³Ways to pre-process non-numerical features for use in linear models are discussed in [Chapter 10](#).

back-propagation training algorithm (Rumelhart, Hinton and Williams, 1986). In this section we also learned about the dual, instance-based view of linear classification in which we are learning instance weights rather than feature weights. For the perceptron these weights are the number of times the example has been misclassified during training.

- ☞ Maximum-margin classification with support vector machines was the topic of Section 7.3. The approach was proposed by Boser, Guyon and Vapnik (1992). Using the dual formulation, the instance weights are non-zero only for the support vectors, which are the training instances on the margin. The soft-margin generalisation is due to Cortes and Vapnik (1995). Margin errors are allowed, but the total margin error is added as a regularisation term to the objective function to be minimised, weighted by the complexity parameter C ; all instances inside the margin receive instance weight C . As we have seen, by making C sufficiently small the support vector machine summarises the classes by their unweighted class means and hence is very similar to the basic linear classifier. A general introduction to SVMs is provided by Cristianini and Shawe-Taylor (2000). The sequential minimal optimisation algorithm is an often-used solver which iteratively selects pairs of multipliers to optimise analytically and is due to Platt (1998).
- ☞ In Section 7.4 we considered two methods to turn linear classifiers into probability estimators by converting the signed distance from the decision boundary into class probabilities. One well-known method is to use the logistic function, either straight out of the box or by fitting location and spread parameters to the data. Although this is often presented as a simple trick, we saw how it can be justified by assuming that the distances per class are normally distributed with the same variance; this latter assumption is needed to make the transformation monotonic. A non-parametric alternative is to use the ROC convex hull to obtain calibrated probability estimates. As was already mentioned in the summary of Chapter 2, the approach has its roots in isotonic regression (Best and Chakravarti, 1990) and was introduced to the machine learning community by Zadrozny and Elkan (2002). Fawcett and Niculescu-Mizil (2007) and Flach and Matsubara (2007) show its equivalence to calibration by means of the ROC convex hull.
- ☞ Finally, Section 7.5 discussed briefly how to go beyond linearity with kernel methods. The ‘kernel trick’ can be applied to any learning algorithm that can be entirely described in terms of dot products, which includes most approaches discussed in this chapter. The beauty is that we are implicitly classifying in a high-dimensional feature space, without having to construct the space explicitly. I

gave the kernel perceptron as a simple example of a kernelised algorithm; in the next chapter we will see another example. [Shawe-Taylor and Cristianini \(2004\)](#) provide an excellent reference bringing together a wealth of material on the use of kernels in machine learning, and [Gärtner \(2009\)](#) discusses how kernel methods can be applied to structured, non-numerical data.



Distance-based models

MANY FORMS OF LEARNING are based on generalising from training data to unseen data by exploiting the similarities between the two. With grouping models such as decision trees these similarities take the form of an equivalence relation or partition of the instance space: two instances are similar whenever they end up in the same segment of this partition. In this chapter we consider learning methods that utilise more graded forms of similarity. There are many different ways in which similarity can be measured, and in [Section 8.1](#) we take a look at the most important of them. [Section 8.2](#) is devoted to a discussion of two key concepts in distance-based machine learning: neighbours and exemplars. In [Section 8.3](#) we consider what is perhaps the best-known distance-based learning method: the nearest-neighbour classifier. [Section 8.4](#) investigates K -means clustering and close relatives, and [Section 8.5](#) looks at hierarchical clustering by constructing dendrograms. Finally, in [Section 8.6](#) we discuss how several of these methods can be extended using the kind of kernels that we saw in the previous chapter.

8.1 So many roads...

It may seem odd at first that there should be many ways to measure distance. I am not referring to the fact that distance can be measured on different scales (kilometres, miles, nautical miles, and so on), as such changes of scale are simple monotonic trans-

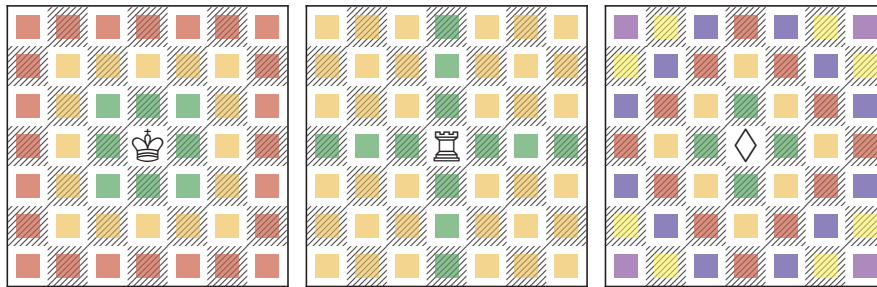


Figure 8.1. (left) Distance as experienced by a King on a chessboard: green squares are one move away, orange ones two moves and red ones three moves. The shape formed by equidistant squares from the current position is itself a square. (middle) A Rook can travel any number of squares in one move, but only horizontally or vertically. No square is further than two moves away. (right) The (fictional) KRook combines the restrictions of King and Rook: it can move only one square at a time, and only horizontally or vertically. Equidistant squares now form a lozenge.

formations and do not fundamentally alter the distance measure. A better intuition is obtained by taking the mode of travel into account. Clearly, when travelling from Bristol to Amsterdam by train you travel a larger distance than when travelling by plane, because planes are less restricted in their paths than trains. We will explore this a bit further by considering the game of chess.

In chess, each piece is governed by a set of rules that restrict its possible moves. These restrictions can be directional: for instance, King and Queen can move horizontally, vertically and diagonally, while a Bishop can only move diagonally, a Rook only horizontally and vertically, and pawns only upwards. King and pawn are further restricted by the fact that they can move only one square at a time, whereas Queen, Rook and Bishop can move any number of squares in a single allowed direction. Finally, a Knight moves according to a very specific pattern (one diagonal step and one horizontal or vertical step in a single move).

Although these pieces move around on the same board, they experience distances in very different ways. For example, the next square down is one move away for King, Queen and Rook; three moves away for a Knight; and unreachable for Bishop or pawn. This is, of course, very similar to our experience in the real world. Trains and cars can only move along tracks or roads, like a Bishop, which leaves remote places unreachable. A mountain range can mean large detours when travelling by car, train or on foot, but is easy to cross when flying. On an underground, two stations a few streets away may be only reachable with several changes of line, not unlike the way a Knight can reach a nearby square only in two or three moves. And on foot we are most flexible but

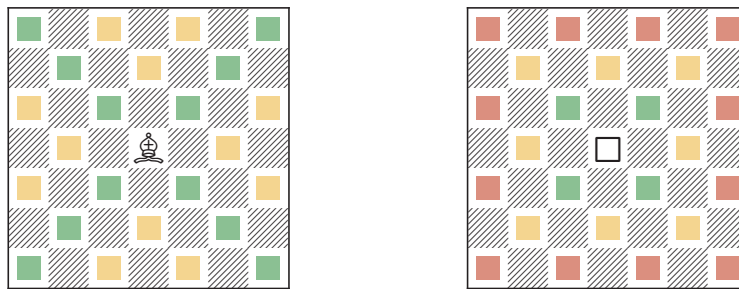


Figure 8.2. (left) The Bishop's world: squares are either one or two moves away, or else unreachable. **(right)** The fictional Bing combines the restrictions of King and Bishop: it can only move one square at a time, and only diagonally. Equidistant squares now form a punctuated square.

also slow, like a King.

Figure 8.1 visualises the distances experienced by King and Rook. Both can reach all parts of the chessboard, but a Rook can travel much faster. In fact, a Rook can reach any square in either one or two moves (assuming no other pieces are in its way). All squares one move away form a cross, and the remaining squares are one additional move away. A King will often have to travel more than two moves to reach a particular square (although there are also squares that the King can reach in one move while a Rook needs two). The squares one move away form a small square shape around the current position; those two moves away form a larger square around the smaller square; and so on. Figure 8.1 (right) shows a piece that doesn't exist in chess, but could. It combines the restrictions of King and Rook, and I therefore call it a KRoook. Like a King, it can only move one square at a time; and like a Rook, it can only move horizontally and vertically. For the KRoook, equidistant squares form a sort of lozenge around the current position.

Figure 8.2 (left) visualises the Bishop's moves. The Bishop is somewhat similar to the Rook in that some squares (those of the same colour as its current square) are never more than two moves away; however, the remaining squares of the other colour are unreachable. Combining the restrictions of the Bishop (only diagonal moves) with those of the King (one square per move) we obtain another fictional piece, the Bing (Figure 8.2 (right)). We could say that the world of Bishops and Bings is rotated 45 degrees, compared with the world of Rooks and KRoooks.

What's the relevance of all this when trying to understand distance-based machine learning, you may ask? Well, the rank (row) and file (column) on a chessboard is not unlike a discrete or categorical feature in machine learning (in fact, since ranks and files are ordered, they are *ordinal features*, as we will further discuss in Chapter 10). We can switch to real-valued features by imagining a 'continuous' chessboard with

infinitely many, infinitesimally narrow ranks and files. Squares now become points, and distances are not expressed as the number of squares travelled, but simply as a real number on some scale. If we now look at the shapes obtained by connecting equidistant points, we see that many of these carry over from the discrete to the continuous case. For a King, for example, all points a given fixed distance away still form a square around the current position; and for a KRook they still form a square rotated 45 degrees. As it happens, these are special cases of the following generic concept.

Definition 8.1 (Minkowski distance). If $\mathcal{X} = \mathbb{R}^d$, the **Minkowski distance** of order $p > 0$ is defined as

$$\text{Dis}_p(\mathbf{x}, \mathbf{y}) = \left(\sum_{j=1}^d |x_j - y_j|^p \right)^{1/p} = \|\mathbf{x} - \mathbf{y}\|_p$$

where $\|\mathbf{z}\|_p = \left(\sum_{j=1}^d |z_j|^p \right)^{1/p}$ is the **p -norm** (sometimes denoted L_p norm) of the vector $\mathbf{z}$. We will often refer to Dis_p simply as the p -norm. 

So, the **2-norm** refers to the familiar **Euclidean distance**

$$\text{Dis}_2(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{j=1}^d (x_j - y_j)^2} = \sqrt{(\mathbf{x} - \mathbf{y})^T (\mathbf{x} - \mathbf{y})}$$

which measures distance ‘as the crow flies’. Two other values of p can be related back to the chess example. The **1-norm** denotes **Manhattan distance**, also called **cityblock distance**:

$$\text{Dis}_1(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^d |x_j - y_j|$$

This is the distance if we can only travel along coordinate axes: similar to a taxi in Manhattan or other cities whose streets follow a regular grid pattern, but also the distance experienced by our fictional KRook piece. If we now let p grow larger, the distance will be more and more dominated by the largest coordinate-wise distance, from which we can infer that $\text{Dis}_\infty(\mathbf{x}, \mathbf{y}) = \max_j |x_j - y_j|$. This is the distance experienced by the King on a chessboard, who can move diagonally as well as horizontally and vertically but only one step at a time; it is also called **Chebyshev distance**. **Figure 8.3 (left)** visualises equidistant points from the origin using Minkowski distances of various orders. It can be seen that Euclidean distance is the only Minkowski distance that is rotation-invariant – in other words, special significance is given to the directions of the coordinate axes whenever $p \neq 2$. Minkowski distances do not refer to a particular choice of origin and are therefore translation-invariant, but none of them are scaling-invariant.

You will sometimes see references to the **0-norm** (or L_0 norm) which counts the number of non-zero elements in a vector. The corresponding distance then counts the

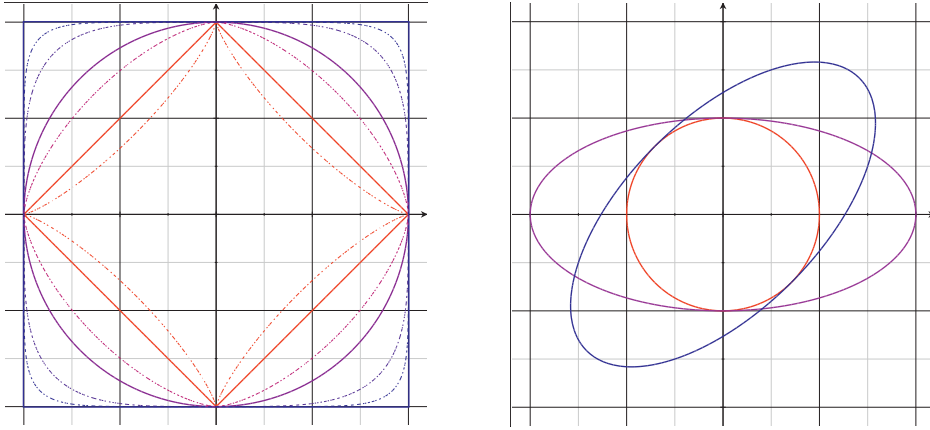


Figure 8.3. (left) Lines connecting points at order- p Minkowski distance 1 from the origin for (from inside) $p = 0.8$; $p = 1$ (Manhattan distance, the **rotated square in red**); $p = 1.5$; $p = 2$ (Euclidean distance, the **violet circle**); $p = 4$; $p = 8$; and $p = \infty$ (Chebyshev distance, the **blue rectangle**). Notice that for points on the coordinate axes all distances agree. For the other points, our reach increases with p ; however, if we require a rotation-invariant distance metric then Euclidean distance is our only choice. (right) The rotated ellipse $\mathbf{x}^T \mathbf{R}^T \mathbf{S}^2 \mathbf{R} \mathbf{x} = 1/4$; the axis-parallel ellipse $\mathbf{x}^T \mathbf{S}^2 \mathbf{x} = 1/4$; and the circle $\mathbf{x}^T \mathbf{x} = 1/4$ ($\mathbf{R}$ and $\mathbf{S}$ as in [Example 8.1](#)).

number of positions in which vectors $\mathbf{x}$ and $\mathbf{y}$ differ. This is not strictly a Minkowski distance; however, we can define it as

$$\text{Dis}_0(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^d (x_j - y_j)^0 = \sum_{j=1}^d I[x_j \neq y_j]$$

under the understanding that $x^0 = 0$ for $x = 0$ and 1 otherwise. This is actually the distance experienced by a Rook on the chessboard: if both rank and file are different the square is two moves away, if only one of them is different the square is one move away. If $\mathbf{x}$ and $\mathbf{y}$ are binary strings, this is also called the **Hamming distance**. Alternatively, we can see the Hamming distance as the number of bits that need to be flipped to change $\mathbf{x}$ into $\mathbf{y}$; for non-binary strings of unequal length this can be generalised to the notion of **edit distance** or **Levenshtein distance**.

Do all of these mathematical constructs make sense as a notion of distance? In order to answer that question we can draw up a list of properties that a proper distance measure should have, such as non-negativity and symmetry. The generally agreed-upon list defines what is known as a metric.

Definition 8.2 (Distance metric). Given an instance space $\mathcal{X}$, a **distance metric** is a function $\text{Dis} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ such that for any $x, y, z \in \mathcal{X}$:

1. distances between a point and itself are zero: $\text{Dis}(x, x) = 0$;

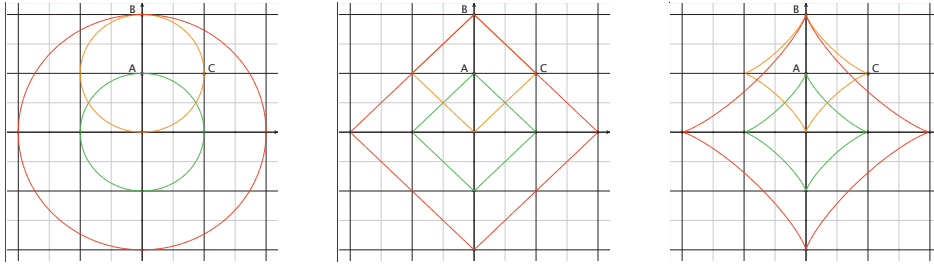


Figure 8.4. (left) The green circle connects points the same Euclidean distance (i.e., Minkowski distance of order $p = 2$) away from the origin as A. The orange circle shows that B and C are equidistant from A. The red circle demonstrates that C is closer to the origin than B, which conforms to the triangle inequality. (middle) With Manhattan distance ($p = 1$), B and C are equally close to the origin and also equidistant from A. (right) With $p < 1$ (here, $p = 0.8$) C is further away from the origin than B; since both are again equidistant from A, it follows that travelling from the origin to C via A is quicker than going there directly, which violates the triangle inequality.

2. all other distances are larger than zero: if $x \neq y$ then $\text{Dis}(x, y) > 0$;
3. distances are symmetric: $\text{Dis}(y, x) = \text{Dis}(x, y)$;
4. detours can not shorten the distance: $\text{Dis}(x, z) \leq \text{Dis}(x, y) + \text{Dis}(y, z)$.

If the second condition is weakened to a non-strict inequality – i.e., $\text{Dis}(x, y)$ may be zero even if $x \neq y$ – the function Dis is called a **pseudo-metric**. 

The last condition is called the **triangle inequality** (or sub-additivity, as it really concerns the interaction between distance and addition). Figure 8.4 investigates this for Minkowski distances of various orders. The triangle inequality dictates that the distance from the origin to C is no more than the sum of the distances from the origin to A ($\text{Dis}(O, A)$) and from A to C ($\text{Dis}(A, C)$). B is at the same distance from A as C, regardless of the distance measure used; so $\text{Dis}(O, A) + \text{Dis}(A, C)$ is equal to the distance from the origin to B. So, if we draw a circle around the origin through B, the triangle inequality dictates that C not be outside that circle. As we see in the left figure for Euclidean distance, B is the only point where the circles around the origin and around A intersect, so everywhere else the triangle inequality is a strict inequality.

The middle figure shows the same situation for Manhattan distance ($p = 1$). Now, B and C are in fact equidistant from the origin, and so travelling via A to C is no longer a detour, but just one of the many shortest routes. However, if we now decrease p further, we see that C ends up outside the red shape, and is thus further away than B when seen from the origin, whereas of course the sum of the distances from the origin to A and from A to C is still equal to the distance from the origin to B. At this point, our intuition breaks down: Minkowski distances with $p < 1$ are simply not very useful as distances since they all violate the triangle inequality.

Sometimes it is useful to use different scales for different coordinates if they are traversed with different speeds. For instance, for people horizontal distances can be traversed more easily than vertical differences, and consequently it is more realistic to use an ellipse rather than a circle to identify points that can be reached in a fixed amount of time, with the major axis of the ellipse indicating directions that can be traversed at larger speed. The ellipse can also be rotated, so that the major axis is not aligned with any of the coordinates: for instance, this could be the direction of a motorway, or the wind direction. Mathematically, while hyper-spheres (circles in $d \geq 2$ dimensions) of radius r can be defined by the equation $\mathbf{x}^T \mathbf{x} = r^2$, hyper-ellipses are defined by $\mathbf{x}^T \mathbf{M} \mathbf{x} = r^2$, where $\mathbf{M}$ is a matrix describing the appropriate rotation and scaling.

Example 8.1 (Elliptical distance). Consider the following matrices

$$\mathbf{R} = \begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ -1/\sqrt{2} & 1/\sqrt{2} \end{pmatrix} \quad \mathbf{S} = \begin{pmatrix} 1/2 & 0 \\ 0 & 1 \end{pmatrix} \quad \mathbf{M} = \begin{pmatrix} 5/8 & -3/8 \\ -3/8 & 5/8 \end{pmatrix}$$

The matrix $\mathbf{R}$ describes a clockwise rotation of 45 degrees, and the diagonal matrix $\mathbf{S}$ scales the x -axis by a factor 1/2. The equation

$$(\mathbf{S}\mathbf{R}\mathbf{x})^T (\mathbf{S}\mathbf{R}\mathbf{x}) = \mathbf{x}^T \mathbf{R}^T \mathbf{S}^T \mathbf{S} \mathbf{R} \mathbf{x} = \mathbf{x}^T \mathbf{R}^T \mathbf{S}^2 \mathbf{R} \mathbf{x} = \mathbf{x}^T \mathbf{M} \mathbf{x} = 1/4$$

describes a shape which, after clockwise rotation of 45 degrees and scaling of the x -axis by a factor 1/2, is a circle with radius 1/2 – i.e., the ‘ascending’ ellipse in [Figure 8.3 \(right\)](#). The ellipse equation is $(5/8)x^2 + (5/8)y^2 - (3/4)xy = 1/2$.

Often, the shape of the ellipse is estimated from data as the inverse of the covariance matrix: $\mathbf{M} = \Sigma^{-1}$. This leads to the definition of the *Mahalanobis distance*

$$\text{Dis}_M(\mathbf{x}, \mathbf{y} | \Sigma) = \sqrt{(\mathbf{x} - \mathbf{y})^T \Sigma^{-1} (\mathbf{x} - \mathbf{y})} \quad (8.1)$$

Using the covariance matrix in this way has the effect of decorrelating and normalising the features, as we saw in [Section 7.1](#). Clearly, Euclidean distance is a special case of Mahalanobis distance with the identity matrix $\mathbf{I}$ as covariance matrix: $\text{Dis}_M(\mathbf{x}, \mathbf{y}) = \text{Dis}_M(\mathbf{x}, \mathbf{y} | \mathbf{I})$.

8.2 Neighbours and exemplars

Now that we understand the basics of measuring distance in instance space, we proceed to consider the key ideas underlying distance-based models. The two most

important of these are: formulating the model in terms of a number of prototypical instances or *exemplars*, and defining the decision rule in terms of the nearest exemplars or *neighbours*. We can understand these concepts by revisiting our old friend, the basic linear classifier. This classifier uses the two class means $\mu^{\oplus}$ and $\mu^{\ominus}$ as exemplars, as a summary of all we need to know about the training data in order to build the classifier. A fundamental property of the mean of a set of vectors is that it minimises the sum of squared Euclidean distances to those vectors.

Theorem 8.1 (The arithmetic mean minimises squared Euclidean distance). *The arithmetic mean μ of a set of data points D in a Euclidean space is the unique point that minimises the sum of squared Euclidean distances to those data points.*

Proof. We will show that $\arg \min_{\mathbf{y}} \sum_{\mathbf{x} \in D} \|\mathbf{x} - \mathbf{y}\|^2 = \mu$, where $\|\cdot\|$ denotes the 2-norm. We find this minimum by taking the gradient (the vector of partial derivatives with respect to y_i) of the sum and setting it to the zero vector:

$$\nabla_{\mathbf{y}} \sum_{\mathbf{x} \in D} \|\mathbf{x} - \mathbf{y}\|^2 = -2 \sum_{\mathbf{x} \in D} (\mathbf{x} - \mathbf{y}) = -2 \sum_{\mathbf{x} \in D} \mathbf{x} + 2|D|\mathbf{y} = \mathbf{0}$$

from which we derive $\mathbf{y} = \frac{1}{|D|} \sum_{\mathbf{x} \in D} \mathbf{x} = \mu$. 

Notice that minimising the sum of squared Euclidean distances of a given set of points is the same as minimising the *average* squared Euclidean distance. You may wonder what happens if we drop the square here: wouldn't it be more natural to take the point that minimises total Euclidean distance as exemplar? This point is known as the *geometric median*, as for univariate data it corresponds to the median or 'middle value' of a set of numbers. However, for multivariate data there is no closed-form expression for the geometric median, which needs to be calculated by successive approximation. This computational advantage is the main reason why distance-based methods tend to use squared Euclidean distance.

In certain situations it makes sense to restrict an exemplar to be one of the given data points. In that case, we speak of a *medoid*, to distinguish it from a *centroid* which is an exemplar that doesn't have to occur in the data. Finding a medoid requires us to calculate, for each data point, the total distance to all other data points, in order to choose the point that minimises it. Regardless of the distance metric used, this is an $O(n^2)$ operation for n points, so for medoids there is no computational reason to prefer one distance metric over another. Figure 8.5 shows a set of 10 data points where the different ways of determining exemplars all give different results. In particular, the mean and squared 2-norm medoid can be overly sensitive to outliers.

Once we have determined the exemplars, the basic linear classifier constructs the decision boundary as the perpendicular bisector of the line segment connecting the two exemplars. An alternative, distance-based way to classify instances without direct

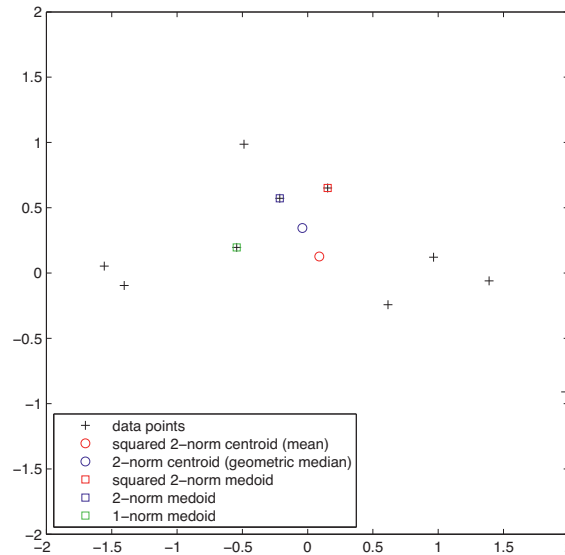


Figure 8.5. A small data set of 10 points, with circles indicating centroids and squares indicating medoids (the latter must be data points), for different distance metrics. Notice how the outlier on the bottom-right ‘pulls’ the mean away from the geometric median; as a result the corresponding medoid changes as well.

reference to a decision boundary is by the following decision rule: if $\mathbf{x}$ is nearest to $\mu^{\oplus}$ then classify it as positive, otherwise as negative; or equivalently, classify an instance to the class of the *nearest* exemplar. If we use Euclidean distance as our closeness measure, simple geometry tells us we get exactly the same decision boundary (Figure 8.6 (left)).

So *the basic linear classifier can be interpreted from a distance-based perspective as constructing exemplars that minimise squared Euclidean distance within each class, and then applying a nearest-exemplar decision rule*. This change of perspective opens up many new possibilities. For example, we can investigate what the decision boundary looks like if we use Manhattan distance for the decision rule (Figure 8.6 (right)). It turns out that the decision boundary can only run along a number of fixed angles: in two dimensions these are horizontal, vertical and at (plus or minus) 45 degrees. This can be understood as follows. Suppose the two exemplars have different x - and y -coordinates, then they span a rectangle (I’ll assume a tall rectangle, as in the figure). Imagine yourself in the centre of that rectangle, then clearly you are at equal distances from both exemplars (in fact, that same point is part of the 2-norm decision boundary). Now, imagine that you move one horizontal step, then you will move closer to one exemplar and away from the other; in order to compensate for that, you will also need to make a vertical step. So, within the rectangle, you maintain equal distance

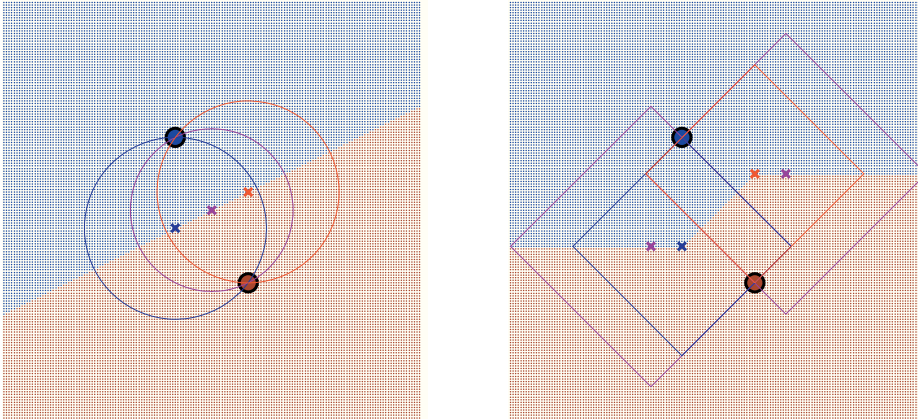


Figure 8.6. (left) For two exemplars the nearest-exemplar decision rule with Euclidean distance results in a linear decision boundary coinciding with the perpendicular bisector of the line connecting the two exemplars. The crosses denote different locations on the decision boundary, and the circles centred at those locations demonstrate that the exemplars are equidistant from each of them. When travelling along the decision boundary from bottom-left to top-right, these circles first shrink then grow again after passing the location halfway between the two exemplars. **(right)** Using Manhattan distance the circles are replaced by diamonds. Travelling from left to right, the diamonds shrink along the left-most horizontal segment of the decision boundary, then stay the same size along the 45-degree segment, and then grow again along the right-most horizontal segment.

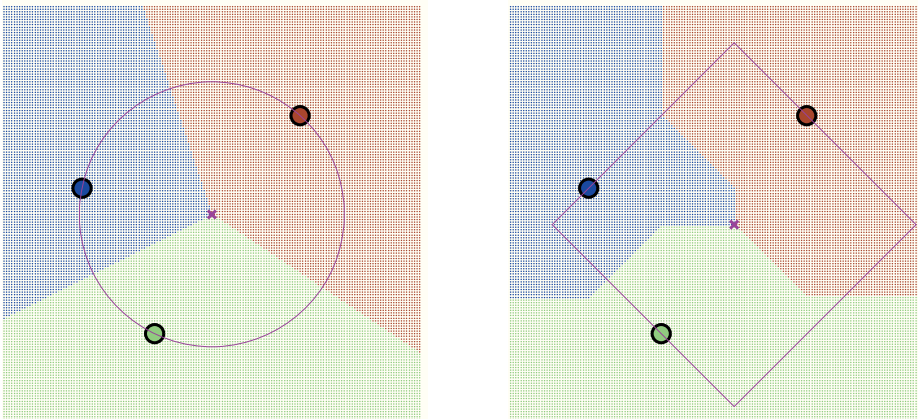


Figure 8.7. (left) Decision regions defined by the 2-norm nearest-exemplar decision rule for three exemplars. **(right)** With Manhattan distance the decision regions become non-convex.

from the exemplars by moving at a 45 degree angle. Once you reach the perimeter of the rectangle you will walk away from both exemplars by making horizontal steps, so from there the decision boundary runs horizontally.

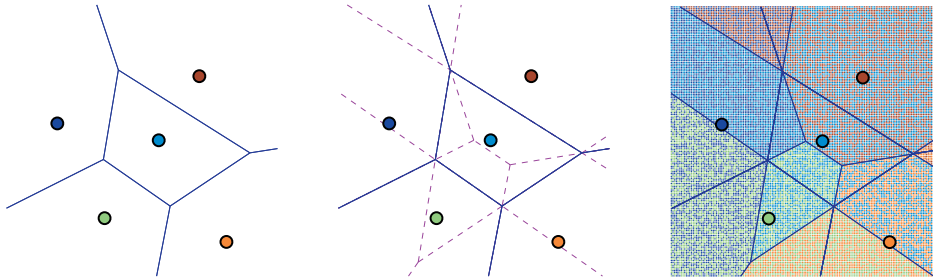


Figure 8.8. (left) Voronoi tessellation for five exemplars. (middle) Taking the two nearest exemplars into account leads to a further subdivision of each Voronoi cell. (right) The shading indicates which exemplars contribute to which cell.

Another useful consequence of switching to the distance-based perspective is that the nearest-exemplar decision rule works equally well for more than two exemplars, which gives us a multi-class version of the basic linear classifier.¹ Figure 8.7 (left) illustrates this for three exemplars. Each decision region is now bounded by two line segments. As you would expect, the 2-norm decision boundaries are more regular than the 1-norm ones: mathematicians say that the 2-norm decision regions are *convex*, which means that linear interpolation between any two points in the region can never go outside it. Clearly, this doesn't hold for 1-norm decision regions (Figure 8.7 (right)).

Increasing the number of exemplars further means that some of the regions become closed convex 'cells' (we are assuming Euclidean distance for the remainder of this section), giving rise to what is known as a *Voronoi tessellation*. Since the number of classes is typically much lower than the number of exemplars, decision rules often take more than one nearest exemplar into account. This increases the number of decision regions further.

Example 8.2 (Two neighbours know more than one). Figure 8.8 (left) gives a Voronoi tessellation for five exemplars. Each line segment is part of the perpendicular bisector of two exemplars. There are $\binom{5}{2} = 10$ pairs of exemplars, but two of these pairs are too far away from each other so we observe only eight line segments in the Voronoi tessellation.

If we now also take the second-nearest exemplars into account, each Voronoi cell is further subdivided: for instance, since the central point has four neighbours, the central cell is divided into four subregions (Figure 8.8 (middle)). You can think of those additional line segments as being part of the Voronoi tessellation.

¹In information retrieval this is often called the *Rocchio classifier*.

tion that results when the central point is removed. The other exemplars have only three immediate neighbours and so their cells are divided into three subregions. We thus obtain 16 ‘2-nearest exemplar’ decision regions, each of which is defined by a different pair of nearest and second-nearest exemplars.

Figure 8.8 (right) shades each of these regions according to the two nearest exemplars spanning it. Notice that we gave each of the two exemplars the same weight, and so there are pairs of adjacent regions (across each of the original Voronoi boundaries) receiving the same shading, resulting in eight different shadings in all. This will be relevant later on, when we discuss the refinement of nearest-neighbour classifiers.

To summarise, the main ingredients of distance-based models are

- ☞ distance metrics, which can be Euclidean, Manhattan, Minkowski or Mahalanobis, among many others;
- ☞ exemplars: centroids that find a centre of mass according to a chosen distance metric, or medoids that find the most centrally located data point; and
- ☞ distance-based decision rules, which take a vote among the k nearest exemplars.

In the next sections these ingredients are combined in various ways to obtain supervised and unsupervised learning algorithms.

8.3 Nearest-neighbour classification

In the previous section we saw how to generalise the basic linear classifier to more than two classes, by learning an exemplar for each class and using the nearest-exemplar decision rule to classify new data. In fact, the most commonly used distance-based classifier is even more straightforward than that: it simply uses each training instance as an exemplar. Consequently, ‘training’ this classifier requires nothing more than memorising the training data. This extremely simple classifier is known as the *nearest-neighbour classifier*. Its decision regions are made up of the cells of a Voronoi tessellation, with piecewise linear decision boundaries selected from the Voronoi boundaries (since adjacent cells may be labelled with the same class).

What are the properties of the nearest-neighbour classifier? First, notice that, unless the training set contains identical instances from different classes, we will be able to separate the classes perfectly on the training set – not really a surprise, as we memorised all training examples! Furthermore, by choosing the right exemplars we can more or less represent any decision boundary, or at least an arbitrarily close piecewise linear

approximation. It follows that the nearest-neighbour classifier has low bias, but also high variance: move any of the exemplars spanning part of the decision boundary, and you will also change the boundary. This suggests a risk of overfitting if the training data is limited, noisy or unrepresentative.

From an algorithmic point of view, training the nearest-neighbour classifier is very fast, taking only $O(n)$ time for storing n exemplars. The downside is that classifying a single instance also takes $O(n)$ time, as the instance will need to be compared with every exemplar to determine which one is the nearest. It is possible to reduce classification time at the expense of increased training time by storing the exemplars in a more elaborate data structure, but this tends not to scale well to large numbers of features.

In fact, high-dimensional instance spaces can be problematic for another reason: the infamous *curse of dimensionality*. High-dimensional spaces tend to be extremely sparse, which means that every point is far away from virtually every other point, and hence pairwise distances tend to be uninformative. However, whether or not you are hit by the curse of dimensionality is not simply a matter of counting the number of features, as there are several reasons why the effective dimensionality of the instance space may be much smaller than the number of features. For example, some of the features may be irrelevant and drown out the relevant features' signal in the distance calculations. In such a case it would be a good idea, before building a distance-based model, to reduce dimensionality by performing *feature selection*, as will be discussed in Chapter 10. Alternatively, the data may live on a *manifold* of lower dimension than the instance space (e.g., the surface of a sphere is a two-dimensional manifold wrapped around a three-dimensional object), which allows other dimensionality-reduction techniques such as *principal component analysis*, which will be explained in the same chapter. In any case, before applying nearest-neighbour classification it is a good idea to plot a histogram of pairwise distances of a sample to see if they are sufficiently varied.

Notice that the nearest-neighbour method can easily be applied to regression problems with a real-valued target variable. In fact, the method is completely oblivious to the type of target variable and can be used to output text documents, images and videos. It is also possible to output the exemplar itself instead of a separate target, in which case we usually speak of *nearest-neighbour retrieval*. Of course we can only output targets (or exemplars) stored in the exemplar database, but if we have a way of aggregating these we can go beyond this restriction by applying the *k-nearest neighbour* method. In its simplest form, the k -nearest neighbour classifier takes a vote between the $k \geq 1$ nearest exemplars of the instance to be classified, and predicts the majority class. We can easily turn this into a probability estimator by returning the normalised class counts as a probability distribution over classes.

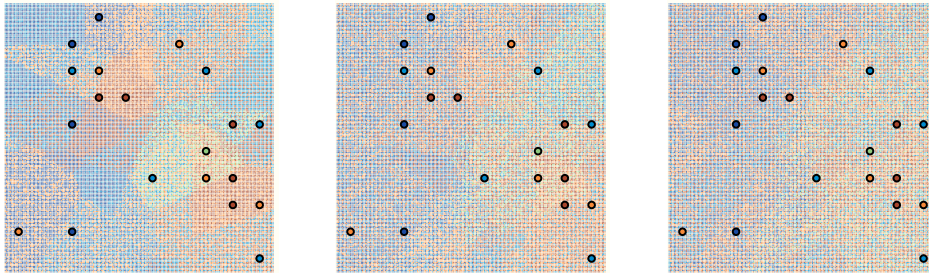


Figure 8.9. (left) Decision regions of a 3-nearest neighbour classifier; the shading represents the predicted probability distribution over the five classes. **(middle)** 5-nearest neighbour. **(right)** 7-nearest neighbour.

Figure 8.9 illustrates this on a small data set of 20 exemplars from five different classes, for $k = 3, 5, 7$. The class distribution is visualised by assigning each test point the class of a uniformly sampled neighbour: so, in a region where two of $k = 3$ neighbours are red and one is orange, the shading is a mix of two-thirds red and one-third orange. While for $k = 3$ the decision regions are still mostly discernible, this is much less so for $k = 5$ and $k = 7$. This may seem at odds with our earlier demonstration of the increase in the number of decision regions with increasing k in Example 8.2. However, this increase is countered by the fact that the probability vectors become more similar to each other. To take an extreme example: if k is equal to the number of exemplars n , every test instance will have the same number of neighbours and will receive the same probability vector which is equal to the prior distribution over the exemplars. If $k = n - 1$ we can reduce one of the class counts by 1, which can be done in c ways: the same number of possibilities as with $k = 1$!

We conclude that the refinement of k -nearest neighbour – the number of different predictions it can make – initially increases with increasing k , then decreases again. Furthermore, we can say that the bias increases and the variance decreases with increasing k . There is no easy recipe to decide what value of k is appropriate for a given data set. However, it is possible to sidestep this question to some extent by applying *distance weighting* to the votes: that is, the closer an exemplar is to the instance to be classified, the more its vote counts. Figure 8.10 demonstrates this, using the reciprocal of the distance to an exemplar as the weight of its vote. This blurs the decision boundaries, as the model now applies a combination of grouping by means of the Voronoi boundaries, and grading by means of distance weighting. Furthermore, since the weights decrease quickly for larger distances, the effect of increasing k is much smaller than with unweighted voting. In fact, with distance weighting we can simply put $k = n$ and still obtain a model that makes different predictions in different parts of the instance space. One could say that distance weighting makes k -nearest neighbour

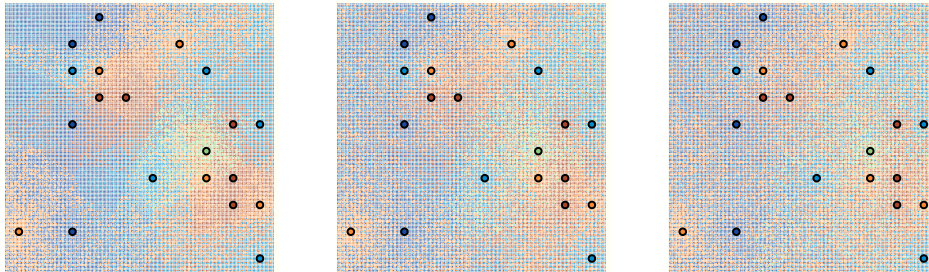


Figure 8.10. (left) 3-nearest neighbour with distance weighting on the data from Figure 8.9. (middle) 5-nearest neighbour. (right) 7-nearest neighbour.

more of a global model, while without it (and for small k) it is more like an aggregation of local models.

If k -nearest neighbour is used for regression problems, the obvious way to aggregate the predictions from the k neighbours is by taking the mean value, which can again be distance-weighted. This would lend the model additional predictive power by predicting values that aren't observed among the stored exemplars. More generally, we can apply k -means to any learning problem where we have an appropriate 'aggregator' for multiple target values.

8.4 Distance-based clustering

In a distance-based context, unsupervised learning is usually taken to refer to clustering, and we will now review a number of distance-based clustering methods. The ones considered in this section are all exemplar-based and hence predictive: they naturally generalise to unseen instances (see Section 3.3 for the distinction between predictive and descriptive clustering). In the next section we consider a clustering method that is not exemplar-based and hence descriptive.

Predictive distance-based clustering methods use the same ingredients as distance-based classifiers: a distance metric, a way to construct exemplars and a distance-based decision rule. In the absence of an explicit target variable, the assumption is that the distance metric indirectly encodes the learning target, so that we aim to find clusters that are *compact* with respect to the distance metric. This requires a notion of cluster compactness that can serve as our optimisation criterion. To that end, we refer back to the scatter matrix introduced in Background 7.2 on p.200.

Definition 8.3 (Scatter). Given a data matrix $\mathbf{X}$, the **scatter matrix** is the matrix

$$\mathbf{S} = (\mathbf{X} - \mathbf{1}\mu)^T (\mathbf{X} - \mathbf{1}\mu) = \sum_{i=1}^n (\mathbf{X}_{i\cdot} - \mu)^T (\mathbf{X}_{i\cdot} - \mu)$$

where μ is a row vector containing all column means of $\mathbf{X}$. The **scatter** of $\mathbf{X}$ is defined as $\text{Scat}(\mathbf{X}) = \sum_{i=1}^n \|\mathbf{X}_i - \mu\|^2$, which is equal to the trace of the scatter matrix (i.e., the sum of its diagonal elements). 

Imagine now that we partition D into K subsets $D_1 \uplus \dots \uplus D_K = D$, and let μ_j denote the mean of D_j . Let $\mathbf{S}$ be the scatter matrix of D , and $\mathbf{S}_j$ be the scatter matrices of D_j . These scatter matrices then have the following relationship:

$$\mathbf{S} = \sum_{j=1}^K \mathbf{S}_j + \mathbf{B} \quad (8.2)$$

Here, $\mathbf{B}$ is the scatter matrix that results by replacing each point in D with the corresponding μ_j . Each $\mathbf{S}_j$ is called a **within-cluster scatter matrix** and describes the compactness of the j -th cluster. $\mathbf{B}$ is the **between-cluster scatter matrix** and describes the spread of the cluster centroids. It follows that the traces of these matrices can be decomposed similarly, which gives

$$\text{Scat}(D) = \sum_{j=1}^K \text{Scat}(D_j) + \sum_{j=1}^K |D_j| \|\mu_j - \mu\|^2 \quad (8.3)$$

What this tells us is that minimising the total scatter over all clusters is equivalent to maximising the (weighted) scatter of the centroids. The **K -means problem** is to find a partition that minimises the total within-cluster scatter.

Example 8.3 (Reducing scatter by partitioning data). Consider the following five points: $(0, 3)$, $(3, 3)$, $(3, 0)$, $(-2, -4)$ and $(-4, -2)$. These points are, conveniently, centred around $(0, 0)$. The scatter matrix is

$$\mathbf{S} = \begin{pmatrix} 0 & 3 & 3 & -2 & -4 \\ 3 & 3 & 0 & -4 & -2 \end{pmatrix} \begin{pmatrix} 0 & 3 \\ 3 & 3 \\ 3 & 0 \\ -2 & -4 \\ -4 & -2 \end{pmatrix} = \begin{pmatrix} 38 & 25 \\ 25 & 38 \end{pmatrix}$$

with trace $\text{Scat}(D) = 76$. If we cluster the first two points together in one cluster and the remaining three in another, then we obtain cluster means $\mu_1 = (1.5, 3)$ and $\mu_2 = (-1, -2)$ and within-cluster scatter matrices

$$\mathbf{S}_1 = \begin{pmatrix} 0-1.5 & 3-1.5 \\ 3-3 & 3-3 \end{pmatrix} \begin{pmatrix} 0-1.5 & 3-3 \\ 3-1.5 & 3-3 \end{pmatrix} = \begin{pmatrix} 4.5 & 0 \\ 0 & 0 \end{pmatrix}$$

$$\mathbf{S}_2 = \begin{pmatrix} 3-(-1) & -2-(-1) & -4-(-1) \\ 0-(-2) & -4-(-2) & -2-(-2) \end{pmatrix} \begin{pmatrix} 3-(-1) & 0-(-2) \\ -2-(-1) & -4-(-2) \\ -4-(-1) & -2-(-2) \end{pmatrix} = \begin{pmatrix} 26 & 10 \\ 10 & 8 \end{pmatrix}$$

with traces $\text{Scat}(D_1) = 4.5$ and $\text{Scat}(D_2) = 34$. Two copies of μ_1 and three copies of μ_2 have, by definition, the same centre of gravity as the complete data set: $(0, 0)$ in this case. We thus calculate the between-cluster scatter matrix as

$$\mathbf{B} = \begin{pmatrix} 1.5 & 1.5 & -1 & -1 & -1 \\ 3 & 3 & -2 & -2 & -2 \end{pmatrix} \begin{pmatrix} 1.5 & 3 \\ 1.5 & 3 \\ -1 & -2 \\ -1 & -2 \\ -1 & -2 \end{pmatrix} = \begin{pmatrix} 7.5 & 15 \\ 15 & 30 \end{pmatrix}$$

with trace 37.5.

Alternatively, if we treat the first three points as a cluster and put the other two in a second cluster, then we obtain cluster means $\mu'_1 = (2, 2)$ and $\mu'_2 = (-3, -3)$, and within-cluster scatter matrices

$$\mathbf{S}'_1 = \begin{pmatrix} 0-2 & 3-2 & 3-2 \\ 3-2 & 3-2 & 0-2 \end{pmatrix} \begin{pmatrix} 0-2 & 3-2 \\ 3-2 & 3-2 \\ 3-2 & 0-2 \end{pmatrix} = \begin{pmatrix} 6 & -3 \\ -3 & 6 \end{pmatrix}$$

$$\mathbf{S}'_2 = \begin{pmatrix} -2-(-3) & -4-(-3) \\ -4-(-3) & -2-(-3) \end{pmatrix} \begin{pmatrix} -2-(-3) & -4-(-3) \\ -4-(-3) & -2-(-3) \end{pmatrix} = \begin{pmatrix} 2 & -2 \\ -2 & 2 \end{pmatrix}$$

with traces $\text{Scat}(D'_1) = 12$ and $\text{Scat}(D'_2) = 4$. The between-cluster scatter matrix is

$$\mathbf{B}' = \begin{pmatrix} 2 & 2 & 2 & -3 & -3 \\ 2 & 2 & 2 & -3 & -3 \end{pmatrix} \begin{pmatrix} 2 & 2 \\ 2 & 2 \\ 2 & 2 \\ -3 & -3 \\ -3 & -3 \end{pmatrix} = \begin{pmatrix} 30 & 30 \\ 30 & 30 \end{pmatrix}$$

with trace 60. Clearly, the second clustering produces tighter clusters whose centroids are further apart.

K-means algorithm

The K -means problem is NP-complete, which means that there is no efficient solution to find the global minimum and we need to resort to a heuristic algorithm. The best-known algorithm is usually also called K -means, although the name ‘Lloyd’s algorithm’ is also used. The outline of the algorithm is given in [Algorithm 8.1](#). The algorithm iterates between partitioning the data using the nearest-centroid decision rule, and

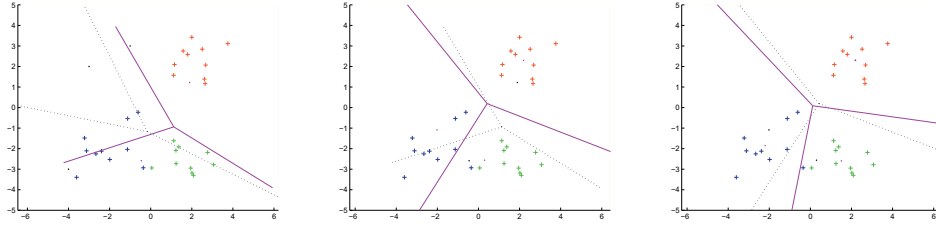


Figure 8.11. (left) First iteration of 3-means on Gaussian mixture data. The dotted lines are the Voronoi boundaries resulting from randomly initialised centroids; the violet solid lines are the result of the recalculated means. (middle) Second iteration, taking the previous partition as starting point (dotted line). (right) Third iteration with stable clustering.

recalculating centroids from a partition. Figure 8.11 demonstrates the algorithm on a small data set with three clusters, and Example 8.4 gives the result on our example data set describing properties of different machine learning methods.

Example 8.4 (Clustering MLM data). Refer back to the MLM data set in Table 1.4 on p.39 (it is also helpful to look at its two-dimensional approximation in Figure 1.7 on p.37). When we run K -means on this data with $K = 3$, we obtain the clusters {Associations, Trees, Rules}, {GMM, naive Bayes}, and a larger cluster with the remaining data points. When we run it with $K = 4$, we get that the large cluster splits into two: {kNN, Linear Classifier, Linear Regression} and

Algorithm 8.1: $KMeans(D, K)$ – K -means clustering using Euclidean distance Dis_2 .

Input : data $D \subseteq \mathbb{R}^d$; number of clusters $K \in \mathbb{N}$.
Output : K cluster means $\mu_1, \dots, \mu_K \in \mathbb{R}^d$.
1 randomly initialise K vectors $\mu_1, \dots, \mu_K \in \mathbb{R}^d$;
2 **repeat**
3 assign each $\mathbf{x} \in D$ to $\arg \min_j Dis_2(\mathbf{x}, \mu_j)$;
4 **for** $j = 1$ to K **do**
5 $D_j \leftarrow \{\mathbf{x} \in D \mid \mathbf{x} \text{ assigned to cluster } j\}$;
6 $\mu_j = \frac{1}{|D_j|} \sum_{\mathbf{x} \in D_j} \mathbf{x}$;
7 **end**
8 **until** no change in $\mu_1, \dots, \mu_K$;
9 **return** $\mu_1, \dots, \mu_K$;

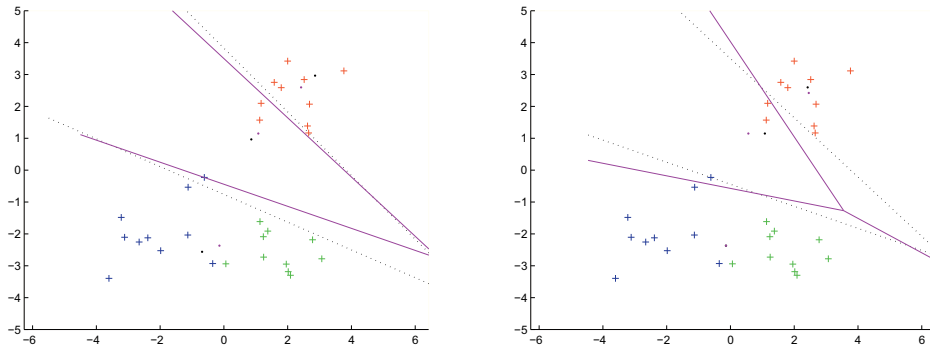


Figure 8.12. (left) First iteration of 3-means on the same data as Figure 8.11 with differently initialised centroids. (right) 3-means has converged to a sub-optimal clustering.

{Kmeans, Logistic Regression, SVM}; but also that GMM gets reallocated to the latter cluster, and naive Bayes ends up as a singleton.

It can be shown that one iteration of K -means can never increase the within-cluster scatter, from which it follows that the algorithm will reach a *stationary point*: a point where no further improvement is possible. It is worth noting that even the simplest data set will have many stationary points.

Example 8.5 (Stationary points in clustering). Consider the task of dividing the set of numbers {8, 44, 50, 58, 84} into two clusters. There are four possible partitions that 2-means can find: {8}, {44, 50, 58, 84}; {8, 44}, {50, 58, 84}; {8, 44, 50}, {58, 84}; and {8, 44, 50, 58}, {84}. It is easy to verify that each of these establishes a stationary point for 2-means, and hence will be found with a suitable initialisation. Only the first clustering is optimal; i.e., it minimises the total within-cluster scatter.

In general, while K -means converges to a stationary point in finite time, no guarantees can be given about whether the convergence point is in fact the global minimum, or if not, how far we are from it. Figure 8.12 shows how an unfortunate initialisation of the centroids can lead to a sub-optimal solution. In practice it is advisable to run the algorithm a number of times and select the solution with the smallest within-cluster scatter.

Clustering around medoids

It is straightforward to adapt the K -means algorithm to use a different distance metric; note that this will also change the objective function being minimised. Algorithm 8.2 gives the *K -medoids* algorithm, which additionally requires the exemplars to be data points. Notice that calculating the medoid of a cluster requires examining all pairs of points – whereas calculating the mean requires just a single pass through the points – which can be prohibitive for large data sets. Algorithm 8.3 gives an alternative algorithm called *partitioning around medoids (PAM)* that tries to improve a clustering locally by swapping medoids with other data points. The quality of a clustering Q is calculated as the total distance over all points to their nearest medoid. Notice that there are $k(n - k)$ pairs of one medoid and one non-medoid, and evaluating Q requires iterating over $n - k$ data points, so the computational cost of one iteration is quadratic in the number of data points. For large data sets one can run PAM on a small sample but evaluate Q on the whole data set, and repeat this a number of times for different samples.

An important limitation of the clustering methods discussed in this section is that they represent clusters only by means of exemplars. This disregards the shape of the clusters, and sometimes leads to counter-intuitive results. The two data sets in Figure 8.13 are identical, except for a rescaling of the y -axis. Nevertheless, K -means finds entirely different clusterings. This is not actually a shortcoming of the K -means algorithm as such, as in Figure 8.13 (right) the two centroids are further away than in the intended solution, and hence this represents a better solution in terms of Equation

Algorithm 8.2: $KMedoids(D, K, Dis)$ – K -medoids clustering using arbitrary distance metric Dis .

Input : data $D \subseteq \mathcal{X}$; number of clusters $K \in \mathbb{N}$;
distance metric $Dis: \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$.

Output : K medoids $\mu_1, \dots, \mu_K \in D$, representing a predictive clustering of $\mathcal{X}$.

```

1 randomly pick  $K$  data points  $\mu_1, \dots, \mu_K \in D$ ;
2 repeat
3   assign each  $\mathbf{x} \in D$  to  $\arg \min_j Dis(\mathbf{x}, \mu_j)$ ;
4   for  $j = 1$  to  $k$  do
5      $D_j \leftarrow \{\mathbf{x} \in D \mid \mathbf{x} \text{ assigned to cluster } j\}$ ;
6      $\mu_j = \arg \min_{\mathbf{x} \in D_j} \sum_{\mathbf{x}' \in D_j} Dis(\mathbf{x}, \mathbf{x}')$ ;
7   end
8 until no change in  $\mu_1, \dots, \mu_K$ ;
9 return  $\mu_1, \dots, \mu_K$ ;
```

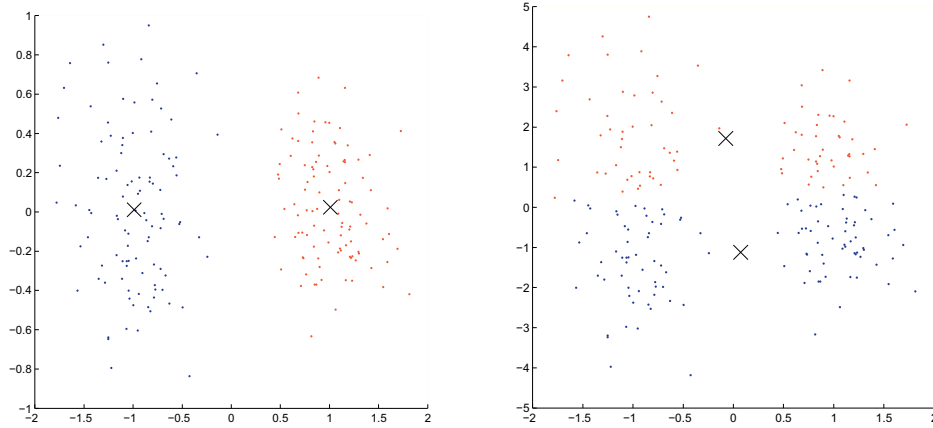


Figure 8.13. (left) On this data 2-means detects the right clusters. (right) After rescaling the y -axis, this configuration has a higher between-cluster scatter than the intended one.

8.3. The real issue is that in this case we want to estimate the ‘shape’ of the clusters as well as the cluster centroids, and hence take account of more than just the trace of the scatter matrices. We will discuss this further in the next chapter.

Algorithm 8.3: $\text{PAM}(D, K, \text{Dis})$ – Partitioning around medoids clustering using arbitrary distance metric Dis .

Input : data $D \subseteq \mathcal{X}$; number of clusters $K \in \mathbb{N}$;
distance metric $\text{Dis} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$.

Output : K medoids $\mu_1, \dots, \mu_K \in D$, representing a predictive clustering of $\mathcal{X}$.

```

1 randomly pick  $K$  data points  $\mu_1, \dots, \mu_K \in D$ ;
2 repeat
3   assign each  $\mathbf{x} \in D$  to  $\arg \min_j \text{Dis}(\mathbf{x}, \mu_j)$ ;
4   for  $j = 1$  to  $k$  do
5      $D_j \leftarrow \{\mathbf{x} \in D \mid \mathbf{x} \text{ assigned to cluster } j\}$ ;
6   end
7    $Q \leftarrow \sum_j \sum_{\mathbf{x} \in D_j} \text{Dis}(\mathbf{x}, \mu_j)$ ;
8   for each medoid  $\mathbf{m}$  and each non-medoid  $\mathbf{o}$  do
9     calculate the improvement in  $Q$  resulting from swapping  $\mathbf{m}$  with  $\mathbf{o}$ ;
10  end
11  select the pair with maximum improvement and swap;
12 until no further improvement possible;
13 return  $\mu_1, \dots, \mu_K$ ;
```

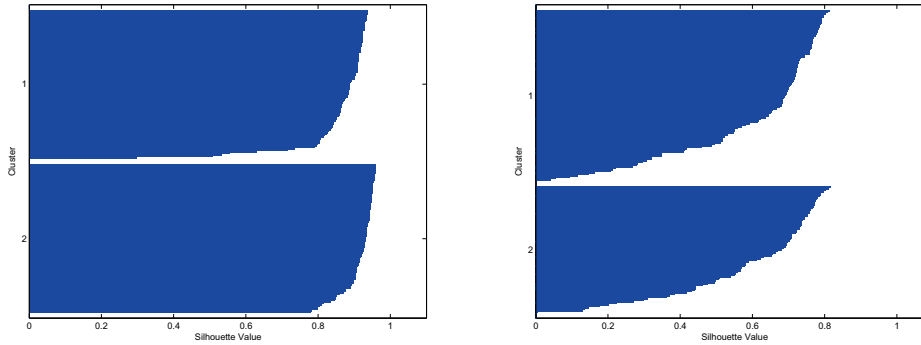


Figure 8.14. (left) Silhouette for the clustering in Figure 8.13 (left), using squared Euclidean distance. Almost all points have a high $s(\mathbf{x})$, which means that they are much closer, on average, to the other members of their cluster than to the members of the neighbouring cluster. (right) The silhouette for the clustering in Figure 8.13 (right) is much less convincing.

Silhouettes

How could we detect the poor quality of the clustering in Figure 8.13 (right)? An interesting technique is the use of silhouettes. For any data point $\mathbf{x}_i$, let $d(\mathbf{x}_i, D_j)$ denote the average distance of $\mathbf{x}_i$ to the data points in cluster D_j , and let $j(i)$ denote the index of the cluster that $\mathbf{x}_i$ belongs to. Furthermore, let $a(\mathbf{x}_i) = d(\mathbf{x}_i, D_{j(i)})$ be the average distance of $\mathbf{x}_i$ to the points in its own cluster $D_{j(i)}$, and let $b(\mathbf{x}_i) = \min_{k \neq j(i)} d(\mathbf{x}_i, D_k)$ be the average distance to the points in its neighbouring cluster. We would expect $a(\mathbf{x}_i)$ to be considerably smaller than $b(\mathbf{x}_i)$, but this cannot be guaranteed. So we can take the difference $b(\mathbf{x}_i) - a(\mathbf{x}_i)$ as an indication of how ‘well-clustered’ $\mathbf{x}_i$ is, and divide this by $b(\mathbf{x}_i)$ to obtain a number less than or equal to 1.

It is, however, conceivable that $a(\mathbf{x}_i) > b(\mathbf{x}_i)$, in which case the difference $b(\mathbf{x}_i) - a(\mathbf{x}_i)$ is negative. This describes the situation that, on average, the members of the neighbouring cluster are closer to $\mathbf{x}_i$ than the members of its own cluster. In order to get a normalised value we divide by $a(\mathbf{x}_i)$ in this case. This leads to the following definition:

$$s(\mathbf{x}_i) = \frac{b(\mathbf{x}_i) - a(\mathbf{x}_i)}{\max(a(\mathbf{x}_i), b(\mathbf{x}_i))} \quad (8.4)$$

A *silhouette* then sorts and plots $s(\mathbf{x})$ for each instance, grouped by cluster. Examples are shown in Figure 8.14 for the two clusterings in Figure 8.13. In this particular case we have used squared Euclidean distance in the construction of the silhouette, but the method can be applied to other distance metrics. We can clearly see that the first clustering is much better than the second. In addition to the graphical representation, we can compute average silhouette values per cluster and over the whole data set.

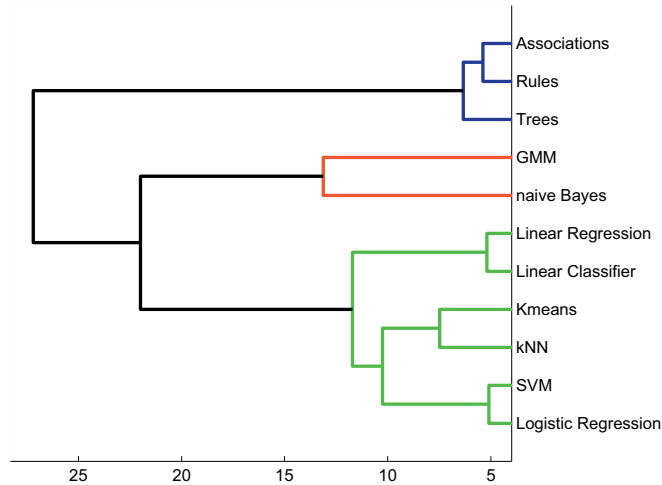


Figure 8.15. A dendrogram (printed left to right to improve readability) constructed by hierarchical clustering from the data in Table 1.4 on p.39.

8.5 Hierarchical clustering

The clustering methods discussed in the previous section use exemplars to represent a predictive clustering: a partition of the entire instance space. In this section we take a look at methods that represent clusters using trees. We previously encountered *clustering trees* in Section 5.3: those trees use features to navigate the instance space, similar to decision trees, and aren't distance-based as such. Here we consider trees called dendrograms, which are purely defined in terms of a distance measure. Because dendrograms use features only indirectly, as the basis on which the distance measure is calculated, they partition the given data rather than the entire instance space, and hence represent a descriptive clustering rather than a predictive one.

Example 8.6 (Hierarchical clustering of MLM data). We continue Example 8.4 on p.248. A hierarchical clustering of the MLM data is given in Figure 8.15. The tree shows that the three logical methods at the top form a strong cluster. If we wanted three clusters, we get the logical cluster, a second small cluster {GMM, naive Bayes}, and the remainder. If we wanted four clusters, we would separate GMM and naive Bayes, as the tree indicates this cluster is the least tight of the three (notice that this is slightly different from the solution found by 4-means). If we wanted five clusters, we would construct {Linear Regression, LinearClassifier} as a separate cluster. This illustrates the key

advantage of hierarchical clustering: it doesn't require fixing the number of clusters in advance.

A precise definition of a dendrogram is as follows.

Definition 8.4 (Dendrogram). *Given a data set D , a **dendrogram** is a binary tree with the elements of D at its leaves. An internal node of the tree represents the subset of elements in the leaves of the subtree rooted at that node. The level of a node is the distance between the two clusters represented by the children of the node. Leaves have level 0.*

For this definition to work, we need a way to measure how close two clusters are. You might think that this is straightforward: just calculate the distance between the two cluster means. However, this occasionally leads to problems, as discussed later in this section. Furthermore, taking cluster means as exemplars assumes Euclidean distance, and we may want to use one of the other distance metrics discussed earlier. This has led to the introduction of the so-called linkage function, which is a general way to turn pairwise point distances into pairwise cluster distances.

Definition 8.5 (Linkage function). *A **linkage function** $L : 2^{\mathcal{X}} \times 2^{\mathcal{X}} \rightarrow \mathbb{R}$ calculates the distance between arbitrary subsets of the instance space, given a distance metric $\text{Dis} : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$.*

The most common linkage functions are as follows:

- Single linkage** defines the distance between two clusters as the *smallest* pairwise distance between elements from each cluster.
- Complete linkage** defines the distance between two clusters as the *largest* pointwise distance.
- Average linkage** defines the cluster distance as the *average* pointwise distance.
- Centroid linkage** defines the cluster distance as the point distance between the cluster means.

These linkage functions can be defined mathematically as follows:

$$\begin{aligned}
 L_{\text{single}}(A, B) &= \min_{x \in A, y \in B} \text{Dis}(x, y) \\
 L_{\text{complete}}(A, B) &= \max_{x \in A, y \in B} \text{Dis}(x, y) \\
 L_{\text{average}}(A, B) &= \frac{\sum_{x \in A, y \in B} \text{Dis}(x, y)}{|A| \cdot |B|} \\
 L_{\text{centroid}}(A, B) &= \text{Dis}\left(\frac{\sum_{x \in A} x}{|A|}, \frac{\sum_{y \in B} y}{|B|}\right)
 \end{aligned}$$

Clearly, all these linkage functions coincide for singleton clusters: $L(\{x\}, \{y\}) = \text{Dis}(x, y)$. However, for larger clusters they start to diverge. For example, suppose $\text{Dis}(x, y) < \text{Dis}(x, z)$, then the linkage between $\{x\}$ and $\{y, z\}$ is different in all four cases:

$$\begin{aligned} L_{\text{single}}(\{x\}, \{y, z\}) &= \text{Dis}(x, y) \\ L_{\text{complete}}(\{x\}, \{y, z\}) &= \text{Dis}(x, z) \\ L_{\text{average}}(\{x\}, \{y, z\}) &= (\text{Dis}(x, y) + \text{Dis}(x, z)) / 2 \\ L_{\text{centroid}}(\{x\}, \{y, z\}) &= \text{Dis}(x, (y + z) / 2) \end{aligned}$$

The general algorithm to build a dendrogram is given in [Algorithm 8.4](#). The tree is built from the data points upwards and is hence a bottom-up or *agglomerative* algorithm. At each iteration the algorithm constructs a new partition of the data by merging the two nearest clusters together. In general, the *HAC* algorithm gives different results when different linkage functions are used. Single linkage is the easiest case to understand, as it effectively builds a graph by adding increasingly longer links between points, one at a time, such that ultimately there is a path between any pair of points (hence the term ‘linkage’). At any point during this process, the connected components are the clusters found at that iteration, and the linkage of the most recently found cluster is the length of the most recently added link. Hierarchical clustering using single linkage can essentially be done by calculating and sorting all pairwise distances between data points, which requires $O(n^2)$ time for n points. The other linkage functions require at least $O(n^2 \log n)$. Notice that the unoptimised algorithm in [Algorithm 8.4](#) has time complexity $O(n^3)$.

Algorithm 8.4: *HAC*(D, L) – Hierarchical agglomerative clustering.

Input : data $D \subseteq \mathcal{X}$; linkage function $L : 2^{\mathcal{X}} \times 2^{\mathcal{X}} \rightarrow \mathbb{R}$ defined in terms of distance metric.

Output : a dendrogram representing a descriptive clustering of D .

- 1 initialise clusters to singleton data points;
 - 2 create a leaf at level 0 for every singleton cluster;
 - 3 **repeat**
 - 4 | find the pair of clusters X, Y with lowest linkage l , and merge;
 - 5 | create a parent of X, Y at level l ;
 - 6 **until** all data points are in one cluster;
 - 7 **return** the constructed binary tree with linkage levels;
-

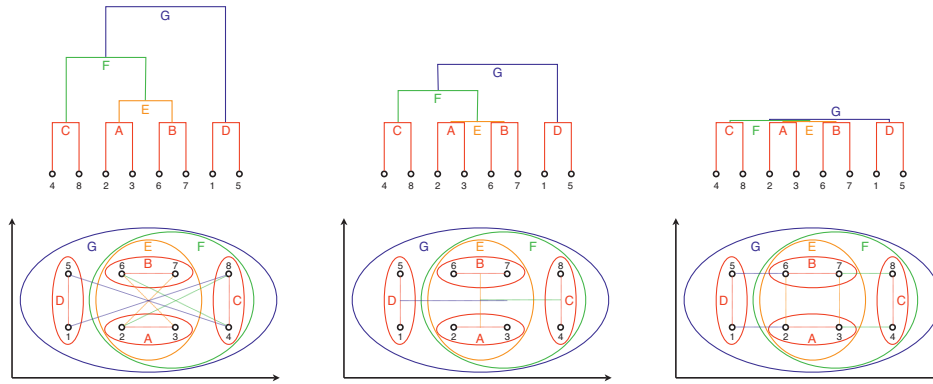


Figure 8.16. (left) Complete linkage defines cluster distance as the largest pairwise distance between elements from each cluster, indicated by the coloured lines between data points. The clustering found can be represented as nested partitions (bottom) or a dendrogram (top); the level of a horizontal connection between clusters in the dendrogram corresponds to the length of a linkage line. The example assumes that ties are broken by small irregularities in the grid. (middle) Centroid linkage defines the distance between clusters as the distance between their means. Notice that E obtains the same linkage as A and B, and so the latter clusters effectively disappear. (right) Single linkage defines the distance between clusters as the smallest pairwise distance. The dendrogram all but collapses, which means that no meaningful clusters are found in the given grid configuration.

Example 8.7 (Linkage matters). We consider a regular grid of 8 points in two rows of four (Figure 8.16). We assume that ties are broken by small irregularities. Each linkage function merges the same clusters in the same order, but the linkages are quite different in each case. Complete linkage gives the impression that D is far removed from the rest, whereas by moving D very slightly to the right it would have been added to E before C. With centroid linkage we see that E has in fact the same linkage as A and B, which means that A and B are not really discernible as separate clusters, even though they are found first. Single linkage seems preferable in this case, as it most clearly demonstrates that there is no meaningful cluster structure in this set of points.

Single and complete linkage both define the distance between clusters in terms of a particular pair of points. Consequently, they cannot take the shape of the cluster into account, which is why average and centroid linkage can offer an advantage. However, centroid linkage can lead to non-intuitive dendrograms, as illustrated in Figure

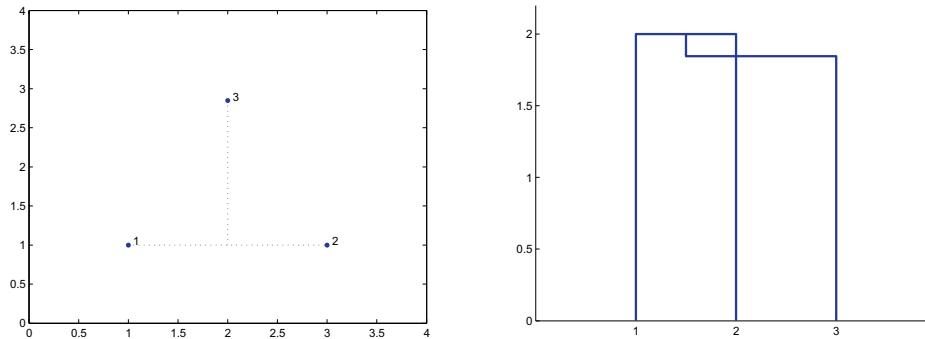


Figure 8.17. (left) Points 1 and 2 are closer to each other than to point 3. However, the distance between point 3 to the centroid of the other two points is less than any of the pairwise distances. **(right)** This results in a decrease in linkage when adding point 3 to cluster $\{1, 2\}$, and hence a non-monotonic dendrogram.

8.17. The issue here is that we have $L(\{1\}, \{2\}) < L(\{1\}, \{3\})$ and $L(\{1\}, \{2\}) < L(\{2\}, \{3\})$ but $L(\{1\}, \{2\}) > L(\{1, 2\}, \{3\})$. The first two inequalities mean that 1 and 2 are the first to be merged into a cluster; but the second inequality means that the level of cluster $\{1, 2, 3\}$ in the dendrogram drops below the level of $\{1, 2\}$. Centroid linkage violates the requirement of *monotonicity*, which stipulates that $L(A, B) < L(A, C)$ and $L(A, B) < L(B, C)$ implies $L(A, B) < L(A \cup B, C)$ for any clusters A, B and C . The other three linkage functions are monotonic (the example also serves as an illustration why average linkage and centroid linkage are not the same).

Another thing to keep in mind when constructing dendrograms is that the hierarchical clustering method is deterministic and will always construct a clustering. Consider Figure 8.18, which shows a data set of 20 uniformly randomly sampled points. One would be hard-pressed to find any cluster structure in this data; yet a dendrogram constructed with complete linkage and Euclidean distance appears to indicate that there are three or four clearly discernible clusters. But if we look closer, we see that the linkage levels are very close together in the bottom of the tree, and the fact that linkages are higher towards the top comes primarily from the use of complete linkage, which concentrates on maximal pairwise distances. The silhouette in Figure 8.18 (right) confirms that the cluster structure is not very strong. Effectively, we are witnessing here a particular, clustering-related kind of overfitting, already familiar from other tree-based models discussed in Chapter 5. Furthermore, dendrograms – like other tree models – have high variance in that small changes in the data points can lead to large changes in the dendrogram.

In conclusion, hierarchical clustering methods have the distinct advantage that the number of clusters does not need to be fixed in advance. However, this advantage comes at considerable computational cost. Furthermore, we now need to choose not

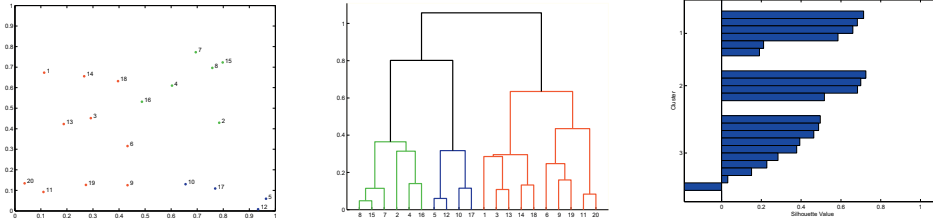


Figure 8.18. (left) 20 data points, generated by uniform random sampling. (middle) The dendrogram generated from complete linkage. The three clusters suggested by the dendrogram are spurious as they cannot be observed in the data. (right) The rapidly decreasing silhouette values in each cluster confirm the absence of a strong cluster structure. Point 18 has a negative silhouette value as it is on average closer to the green points than to the other red points.

just the distance measure used, but also the linkage function.

8.6 From kernels to distances

In Section 7.5 we discussed how kernels can be used to extend the power of linear models considerably. Recall that a kernel is a function $\kappa(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j)$ that calculates a dot product in some feature space, but without constructing the feature vectors $\phi(\mathbf{x})$ explicitly. Any learning method that can be defined purely in terms of dot products of data points is amenable to such ‘kernelisation’. Because of the close connection between Euclidean distance and dot products we can apply the same ‘kernel trick’ to many distance-based learning methods.

The key insight is that Euclidean distance can be rewritten in terms of dot products:

$$\text{Dis}_2(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|_2 = \sqrt{(\mathbf{x} - \mathbf{y}) \cdot (\mathbf{x} - \mathbf{y})} = \sqrt{\mathbf{x} \cdot \mathbf{x} - 2\mathbf{x} \cdot \mathbf{y} + \mathbf{y} \cdot \mathbf{y}}$$

This formula clearly shows that the distance between $\mathbf{x}$ and $\mathbf{y}$ decreases whenever the dot product $\mathbf{x} \cdot \mathbf{y}$ increases, which suggests that the dot product itself is a kind of similarity measure. However, it is not translation-invariant, because it depends on the location of the origin. The two terms $\mathbf{x} \cdot \mathbf{x}$ and $\mathbf{y} \cdot \mathbf{y}$ have the effect of making the overall expression translation-invariant. Replacing the dot product with a kernel function κ , we can construct the following kernelised distance:

$$\text{Dis}_\kappa(\mathbf{x}, \mathbf{y}) = \sqrt{\kappa(\mathbf{x}, \mathbf{x}) - 2\kappa(\mathbf{x}, \mathbf{y}) + \kappa(\mathbf{y}, \mathbf{y})} \quad (8.5)$$

It turns out that Dis_κ defines a pseudo-metric (see Definition 8.2 on p.235) whenever κ is a positive semi-definite kernel.²

²It is only a metric if the feature mapping ϕ is injective: suppose not, then some distinct $\mathbf{x}$ and $\mathbf{y}$ are mapped to the same feature vector $\phi(\mathbf{x}) = \phi(\mathbf{y})$, from which we derive $\kappa(\mathbf{x}, \mathbf{x}) - 2\kappa(\mathbf{x}, \mathbf{y}) + \kappa(\mathbf{y}, \mathbf{y}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{x}) - 2\phi(\mathbf{x}) \cdot \phi(\mathbf{y}) + \phi(\mathbf{y}) \cdot \phi(\mathbf{y}) = 0$.

As an illustration, [Algorithm 8.5](#) adapts the K -means algorithm ([Algorithm 8.1](#) on [p.248](#)) to use a kernelised distance. So, the algorithm clusters according to a non-linear distance in instance space, corresponding to Euclidean distance in an implicit feature space. However, one complication arises, which is that [Theorem 8.1](#) doesn't apply to non-linear distances, and so we cannot construct cluster means in instance space. For this reason [Algorithm 8.5](#) treats the clustering as a partition rather than a set of exemplars. Consequently, assigning each data point $\mathbf{x}$ to its nearest cluster (step 3) is now of quadratic complexity, since for each cluster we need to sum up the distances of all its members to $\mathbf{x}$. In contrast, this step is linear in $|D|$ for the K -means algorithm.

There is an alternative way to turn dot products into distances. Since the dot product can be written as $\|\mathbf{x}\| \cdot \|\mathbf{y}\| \cos \theta$, where θ is the angle between the vectors $\mathbf{x}$ and $\mathbf{y}$, we define the *cosine similarity* as

$$\cos \theta = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \cdot \|\mathbf{y}\|} = \frac{\mathbf{x} \cdot \mathbf{y}}{\sqrt{(\mathbf{x} \cdot \mathbf{x})(\mathbf{y} \cdot \mathbf{y})}} \quad (8.6)$$

Cosine similarity differs from Euclidean distance in that it doesn't depend on the length of the vectors $\mathbf{x}$ and $\mathbf{y}$. On the other hand, it is not translation-independent, but assigns special status to the origin: one way to think of it is as a projection onto a unit sphere around the origin, and measuring distance on that sphere. Cosine similarity is usually turned into a distance metric by taking $1 - \cos \theta$. Being defined entirely in terms of dot products, it is as easily kernelised as Euclidean distance.

Algorithm 8.5: $\text{Kernel-KMeans}(D, K)$ – K -means clustering using kernelised distance Dis_κ .

Input : data $D \subseteq \mathcal{X}$; number of clusters $K \in \mathbb{N}$.
Output : K -fold partition $D_1 \uplus \dots \uplus D_K = D$.
1 randomly initialise K clusters $D_1, \dots, D_K$;
2 **repeat**
3 assign each $\mathbf{x} \in D$ to $\arg \min_j \frac{1}{|D_j|} \sum_{\mathbf{y} \in D_j} \text{Dis}_\kappa(\mathbf{x}, \mathbf{y})$;
4 **for** $j = 1$ to K **do**
5 $D_j \leftarrow \{\mathbf{x} \in D \mid \mathbf{x} \text{ assigned to cluster } j\}$;
6 **end**
7 **until** no change in $D_1, \dots, D_K$;
8 **return** $D_1, \dots, D_K$;

8.7 Distance-based models: Summary and further reading

Along with linear models, distance-based models are the second group of models with strong geometric intuitions. The literature on distance-based models is rich and diverse; in this chapter I've concentrated on getting the main intuitions across.

- ☞ In [Section 8.1](#) we reviewed the most commonly used distance metrics: the Minkowski distance or p -norm with special cases Euclidean distance ($p = 2$) and Manhattan distance ($p = 1$); the Hamming distance, which counts the number of bits or literals that are different; and the Mahalanobis distance, which decorrelates and normalises the features ([Mahalanobis, 1936](#)). Other distances can be taken into account, as long as they satisfy the requirements of a distance metric listed in [Definition 8.2](#).
- ☞ [Section 8.2](#) investigated the key concepts of neighbours and exemplars. Exemplars are either centroids that find a centre of mass according to a chosen distance metric, or medoids that find the most centrally located data point. The most commonly used centroid is the arithmetic mean, which minimises squared Euclidean distance to all other points. Other definitions of centroids are possible but harder to compute: e.g., the geometric median is the point minimising Euclidean distance, but does not admit a closed-form solution. The complexity of finding a medoid is always quadratic regardless of the distance metric. We then considered nearest-neighbour decision rules, and looked in particular at the difference between 2-norm and 1-norm nearest-exemplar decision boundaries, and how these get refined by switching to a 2-nearest-exemplars decision rule.
- ☞ In [Section 8.3](#) we discussed nearest-neighbour models which simply use the training data as exemplars. This is a very widely used model for classification, the origins of which can be traced back to [Fix and Hodges \(1951\)](#). Despite its simplicity, it can be shown that with sufficient training data the error rate is at most twice the optimal error rate ([Cover and Hart, 1967](#)). The 1-nearest neighbour classifier has low bias but high variance; by increasing the number of neighbours over which we aggregate we can reduce the variance but at the same time increase the bias. The nearest-neighbour decision rule can also be applied to real-valued target variables, and more generally to any task where we have an appropriate aggregator for multiple target values.
- ☞ [Section 8.4](#) considered a number of algorithms for distance-based clustering using either arithmetic means or medoids. The K -means algorithm is a simple heuristic approach to solve the K -means problem that was originally proposed

in 1957 and is sometimes referred to as Lloyd's algorithm (Lloyd, 1982). It is dependent on the initial configuration and can easily converge to the wrong stationary point. We also looked at the K -medoids and partitioning around medoids algorithms, the latter due to Kaufman and Rousseeuw (1990). These are computationally more expensive due to the use of medoids. Silhouettes (Rousseeuw, 1987) are a useful technique to check whether points are on average closer to the other members of their cluster than they are to the members of the neighbouring cluster. Much more detail about these and other clustering methods is provided by Jain, Murty and Flynn (1999).

- ☞ Whereas the previous clustering methods all result in a partition of the instance space and are therefore predictive, hierarchical clustering discussed in Section 8.5 applies only to the given data and is hence descriptive. A distinct advantage is that the clustering is constructed in the form of a dendrogram, which means that the number of clusters does not need to be specified in advance and can be chosen by inspecting the dendrogram. However, the method is computationally expensive and infeasible for large data sets. Furthermore, it is not always obvious which of the possible linkage functions to choose.
- ☞ Finally, in Section 8.6 we briefly considered how distances can be 'kernelised', and we gave one example in the form of kernel K -means. The use of a non-Euclidean distance metric leads to quadratic complexity of recalculating the clusters in each iteration.

