CHAPTER 3

Beyond binary classification

THE PREVIOUS CHAPTER introduced binary classification and associated tasks such as ranking and class probability estimation. In this chapter we will go beyond these basic tasks in a number of ways. Section 3.1 discusses how to handle more than two classes. In Section 3.2 we consider the case of a real-valued target variable. Section 3.3 is devoted to various forms of learning that are either unsupervised or aimed at learning descriptive models.

3.1 Handling more than two classes

Certain concepts are fundamentally binary. For instance, the notion of a coverage curve does not easily generalise to more than two classes. We will now consider general issues related to having more than two classes in classification, scoring and class probability estimation. The discussion will address two issues: how to evaluate multi-class performance, and how to build multi-class models out of binary models. The latter is necessary for some models, such as linear classifiers, that are primarily designed to separate two classes. Other models, including decision trees, handle any number of classes quite naturally.

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

Multi-class classification

Classification tasks with more than two classes are very common. For instance, once a patient has been diagnosed as suffering from a rheumatic disease, the doctor will want to classify him or her further into one of several variants. If we have k classes, performance of a classifier can be assessed using a k-by-k contingency table. Assessing performance is easy if we are interested in the classifier's accuracy, which is still the sum of the descending diagonal of the contingency table, divided by the number of test instances. However, as before, this can obscure differences in performance on different classes, and other quantities may be more meaningful.

Example 3.1 (Performance of multi-class classifiers). Consider the following three-class confusion matrix (plus marginals):

	Predicted			
	15	**2**	**3**	20
Actual	**7**	**15**	**8**	30
	2	**3**	**45**	50
	24	20	56	100

The accuracy of this classifier is $(15+15+45)/100 = 0.75$. We can calculate per-class precision and recall: for the first class this is $15/24 = 0.63$ and $15/20 = 0.75$ respectively, for the second class $15/20 = 0.75$ and $15/30 = 0.50$, and for the third class $45/56 = 0.80$ and $45/50 = 0.90$. We could average these numbers to obtain single precision and recall numbers for the whole classifier, or we could take a weighted average taking the proportion of each class into account. For instance, the weighted average precision is $0.20 \cdot 0.63 + 0.30 \cdot 0.75 + 0.50 \cdot 0.80 = 0.75$. Notice that we still have that accuracy is weighted average per-class recall, as in the two-class case (see Example 2.1 on p.56).

Another possibility is to perform a more detailed analysis by looking at precision and recall numbers for each pair of classes: for instance, when distinguishing the first class from the third precision is $15/17 = 0.88$ and recall is $15/18 = 0.83$, while distinguishing the third class from the first these numbers are $45/48 = 0.94$ and $45/47 = 0.96$ (can you explain why these numbers are much higher in the latter direction?).

Imagine now that we want to construct a multi-class classifier, but we only have the ability to train two-class models – say linear classifiers. There are various ways to

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

combine several of them into a single k-class classifier. The *one-versus-rest* scheme is to train k binary classifiers, the first of which separates class C_1 from $C_2, \ldots, C_n$, the second of which separates C_2 from all other classes, and so on. When training the i-th classifier we treat all instances of class C_i as positive examples, and the remaining instances as negative examples. Sometimes the classes are learned in a fixed order, in which case we learn $k-1$ models, the i-th one separating C_i from $C_{i+1}, \ldots, C_n$ with $1 \leq i < n$. An alternative to one-versus-rest is *one-versus-one*. In this scheme, we train $k(k-1)/2$ binary classifiers, one for each pair of different classes. If a binary classifier treats the classes asymmetrically, as happens with certain models, it makes more sense to train two classifiers for each pair, leading to a total of $k(k-1)$ classifiers.

A convenient way to describe all these and other schemes to decompose a k-class task into l binary classification tasks is by means of a so-called *output code* matrix. This is a k-by-l matrix whose entries are +1, 0 or −1. The following are output codes describing the two ways to transform a three-class task by means of one-versus-one:

$$\begin{pmatrix} +1 & +1 & 0 \\ -1 & 0 & +1 \\ 0 & -1 & -1 \end{pmatrix} \qquad \begin{pmatrix} +1 & -1 & +1 & -1 & 0 & 0 \\ -1 & +1 & 0 & 0 & +1 & -1 \\ 0 & 0 & -1 & +1 & -1 & +1 \end{pmatrix}$$

Each column of these matrices describes a binary classification task, using the class corresponding to the row with the +1 entry as positive class and the class with the −1 entry as the negative class. So, in the symmetric scheme on the left, we train three classifiers: one to distinguish between C_1 (positive) and C_2 (negative), one to distinguish between C_1 (positive) and C_3 (negative), and the remaining one to distinguish between C_2 (positive) and C_3 (negative). The asymmetric scheme on the right learns three more classifiers with the roles of positives and negatives swapped. The code matrices for the unordered and ordered version of the one-versus-rest scheme are as follows:

$$\begin{pmatrix} +1 & -1 & -1 \\ -1 & +1 & -1 \\ -1 & -1 & +1 \end{pmatrix} \qquad \begin{pmatrix} +1 & 0 \\ -1 & +1 \\ -1 & -1 \end{pmatrix}$$

On the left, we learn one classifier to distinguish C_1 (positive) from C_2 and C_3 (negative), another one to distinguish C_2 (positive) from C_1 and C_3 (negative), and the third one to distinguish C_3 (positive) from C_1 and C_2 (negative). On the right, we have ordered the classes in the order $C_1 - C_2 - C_3$, and thus only two classifiers are needed.

In order to decide the class for a new test instance, we collect predictions from all binary classifiers which can again be +1 for positive, −1 for negative and 0 for no prediction or *reject* (the latter is possible, for instance, with a rule-based classifier). Together, these predictions form a 'word' that can be looked up in the code matrix, a process also known as *decoding*. Suppose the word is −1 +1 −1 and the scheme is un-

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

ordered one-versus-rest, then we know the decision should be class C_2. The question is: what should we with words that do not appear in the code matrix? For instance, suppose the word is $0\ +1\ 0$, and the scheme is symmetric one-versus-one (the first of the above four code matrices). In this case we could argue that the nearest code word is the first row in the matrix, and so we should predict C_1. To make this a little bit more precise, we define the distance between a word w and a code word c as $d(w,c) = \sum_i (1 - w_i c_i)/2$, where i ranges over the 'bits' of the words (the columns in the code matrix). That is, bits where the two words agree do not contribute to the distance; each bit where one word has $+1$ and the other -1 contributes 1; and if one of the bits is 0 the contribution is 1/2, regardless of the other bit.[1] The predicted class for word w is then $\arg\min_j d(w,c_j)$, where c_j is the j-th row of the code matrix. So, if $w = 0\ +1\ 0$ then $d(w,c_1) = 1$ and $d(w,c_2) = d(w,c_3) = 1.5$, which means that we predict C_1.

However, the nearest code word is not always unique. For instance, suppose we use a four-class one-versus-rest scheme, and two of the binary classifiers predict positive and the other two negative, then this word is equidistant to two code words, and so we can't resolve which of the two classes corresponding to the two nearest code words to predict. We can improve the situation by adding more columns to our code matrix:

$$\begin{pmatrix} +1 & -1 & -1 & -1 \\ -1 & +1 & -1 & -1 \\ -1 & -1 & +1 & -1 \\ -1 & -1 & -1 & +1 \end{pmatrix} \qquad \begin{pmatrix} +1 & -1 & -1 & -1 & +1 & +1 & +1 \\ -1 & +1 & -1 & -1 & +1 & -1 & -1 \\ -1 & -1 & +1 & -1 & -1 & +1 & -1 \\ -1 & -1 & -1 & +1 & -1 & -1 & +1 \end{pmatrix}$$

On the left we see a standard four-class one-versus-rest code matrix, which has been extended with three extra columns (i.e., binary learning problems) on the right. As a result, the distance between any two code words has now increased from 2 to 4, increasing the likelihood that we can decode words that are not contained in the code matrix. The resulting scheme can be seen as a mix between one-versus-rest and one-versus-one classification. However, notice that the additional binary learning problems may be hard. For instance, if our four classes are spam e-mails, work e-mails, household e-mails (e.g., utility bills or credit card statements) and private e-mails, then each one-versus-rest binary classification task may be much easier than, say, distinguishing between spam and work e-mails on the one hand and household and private e-mails on the other.

The one-versus-rest and one-versus-one schemes are the most commonly used ways to turn binary classifiers into multi-class classifiers. In order to force a decision in the one-versus-rest scenario we can settle on a class ordering prior to or after learning. In the one-versus-one scheme we can use voting to arrive at a decision, which is actu-

[1] This is a slight generalisation of the *Hamming distance* for binary strings, which counts the number of positions in which the two strings differ.

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

ally equivalent to distance-based decoding as demonstrated by the following example.

Example 3.2 (One-versus-one voting). A one-versus-one code matrix for $k = 4$ classes is as follows:

$$\begin{pmatrix} +1 & +1 & +1 & 0 & 0 & 0 \\ -1 & 0 & 0 & +1 & +1 & 0 \\ 0 & -1 & 0 & -1 & 0 & +1 \\ 0 & 0 & -1 & 0 & -1 & -1 \end{pmatrix}$$

Suppose our six pairwise classifiers predict $w = +1\ -1\ +1\ -1\ +1\ +1$. We can interpret this as votes for $C_1 - C_3 - C_1 - C_3 - C_2 - C_3$; i.e., three votes for C_3, two votes for C_1 and one vote for C_2. More generally, the i-th classifier's vote for the j-th class can be expressed as $(1 + w_i c_{ji})/2$, where c_{ji} is the entry in the j-th row and i-th column of the code matrix. However, this overcounts the 0 entries in the code matrix; since every class participates in $k-1$ pairwise binary tasks, and there are $l = k(k-1)/2$ tasks, the number of zeros in every row is $k(k-1)/2 - (k-1) = (k-1)(k-2)/2 = l(k-2)/k$ (3 in our case). For each zero we need to subtract half a vote, so the number of votes for C_j is

$$\begin{aligned} v_j &= \left(\sum_{i=1}^{l} \frac{1 + w_i c_{ji}}{2} \right) - l\frac{k-2}{2k} = \left(\sum_{i=1}^{l} \frac{w_i c_{ji} - 1}{2} \right) + l - l\frac{k-2}{2k} \\ &= -d_j + l\frac{2k-k+2}{2k} = \frac{(k-1)(k+2)}{4} - d_j \end{aligned}$$

where $d_j = \sum_i (1 - w_i c_{ji})/2$ is the bit-wise distance we used earlier. In other words, the distance and number of votes for each class sum to a constant depending only on the number of classes; with three classes this is 4.5. This can be checked by noting that the distance between w and the first code word is 2.5 (two votes for C_1); with the second code word, 3.5 (one vote for C_2); with the third code word, 1.5 (three votes for C_3); and 4.5 with the fourth code word (no votes).

If our binary classifiers output scores, we can take these into account as follows. As before we assume that the sign of the scores s_i indicates the class. We can then use the appropriate entry in the code matrix c_{ji} to calculate a margin $z_i = s_i c_{ji}$, which we feed into a loss function L (margins and loss functions were discussed in Section 2.2). We thus define the distance between a vector of scores s and the j-th code word c_j as $d(s, c_j) = \sum_i L(s_i c_{ji})$, and we assign the class which minimises this distance. This way of arriving at a multi-class decision from binary scores is called *loss-based decoding*.

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

Example 3.3 (Loss-based decoding). Continuing the previous example, suppose the scores of the six pairwise classifiers are $(+5, -0.5, +4, -0.5, +4, +0.5)$. This leads to the following margins, in matrix form:

$$\begin{pmatrix} +5 & -0.5 & +4 & 0 & 0 & 0 \\ -5 & 0 & 0 & -0.5 & +4 & 0 \\ 0 & +0.5 & 0 & +0.5 & 0 & +0.5 \\ 0 & 0 & -4 & 0 & -4 & -0.5 \end{pmatrix}$$

Using 0–1 loss we ignore the magnitude of the margins and thus predict C_3 as in the voting-based scheme of Example 3.2. Using exponential loss $L(z) = \exp(-z)$, we obtain the distances $(4.67, 153.08, 4.82, 113.85)$. Loss-based decoding would therefore (just) favour C_1, by virtue of its strong wins against C_2 and C_4; in contrast, all three wins of C_3 are with small margin.

It should be noted that loss-based decoding assumes that each binary classifier scores on the same scale.

Multi-class scores and probabilities

If we want to calculate multi-class scores and probabilities from binary classifiers, we have a number of different options.

- We can use the distances obtained by loss-based decoding and turn them into scores by means of some appropriate transformation, just as we turned bit-wise distances into votes in Example 3.2. This method is applicable if the binary classifiers output calibrated scores on a single scale.

- Alternatively, we can use the output of each binary classifier as features (real-valued if we use the scores, binary if we only use the predicted class) and train a model that can produce multi-class scores, such as naive Bayes or tree models. This method is generally applicable but requires additional training.

- A simple alternative that is also generally applicable and often produces satisfactory results is to derive scores from *coverage counts*: the number of examples of each class that are classified as positive by the binary classifer. Example 3.4 illustrates this.

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

Example 3.4 (Coverage counts as scores). Suppose we have three classes and three binary classifiers which either predict positive or negative (there is no reject option). The first classifier classifies 8 examples of the first class as positive, no examples of the second class, and 2 examples of the third class. For the second classifier these counts are 2, 17 and 1, and for the third they are 4, 2 and 8. Suppose a test instance is predicted as positive by the first and third classifiers. We can add the coverage counts of these two classifiers to obtain a score vector of $(12,2,10)$. Likewise, if all three classifiers 'fire' for a particular test instance (i.e., predict positive), the score vector is $(14,19,11)$.

We can describe this scheme conveniently using matrix notation:

$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 8 & 0 & 2 \\ 2 & 17 & 1 \\ 4 & 2 & 8 \end{pmatrix} = \begin{pmatrix} 12 & 2 & 10 \\ 14 & 19 & 11 \end{pmatrix} \tag{3.1}$$

The middle matrix contains the class counts (one row for each classifier). The left 2-by-3 matrix contains, for each example, a row indicating which classifiers fire for that example. The right-hand side then gives the combined counts for each example.

With l binary classifiers, this scheme divides the instance space into up to 2^l regions. Each of these regions is assigned its own score vector, so in order to obtain diverse scores l should be reasonably large.

Once we have multi-class scores, we can ask the familiar question of how good these are. As we have seen in Section 2.1, an important performance index of a binary scoring classifier is the area under the ROC curve or AUC, which is the proportion of correctly ranked positive–negative pairs. Unfortunately ranking does not have a direct multi-class analogue, and so the most obvious thing to do is to calculate the average AUC over binary classification tasks, either in a one-versus-rest or one-versus-one fashion. For instance, the one-versus-rest average AUC estimates the probability that, taking a uniformly drawn class as positive, a uniformly drawn example from that class gets a higher score than a uniformly drawn example over all other classes. Notice that the 'negative' is more likely to come from the more prevalent classes; for that reason the positive class is sometimes also drawn from a non-uniform distribution in which each class is weighted with its prevalence in the test set.

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

Example 3.5 (Multi-class AUC). Assume we have a multi-class scoring classifier that produces a k-vector of scores $\hat{\mathbf{s}}(x) = (\hat{s}_1(x), \ldots, \hat{s}_k(x))$ for each test instance x. By restricting attention to $\hat{s}_i(x)$ we obtain a scoring classifier for class C_i against the other classes, and we can calculate the one-versus-rest AUC for C_i in the normal way.

By way of example, suppose we have three classes, and the one-versus-rest AUCs come out as 1 for the first class, 0.8 for the second class and 0.6 for the third class. Thus, for instance, all instances of class 1 receive a higher first entry in their score vectors than any of the instances of the other two classes. The average of these three AUCs is 0.8, which reflects the fact that, if we uniformly choose an index i, and we select an instance x uniformly among class C_i and another instance x' uniformly among all instances not from C_i, then the expectation that $\hat{s}_i(x) > \hat{s}_i(x')$ is 0.8.

Suppose now C_1 has 10 instances, C_2 has 20 and C_3 70. The weighted average of the one-versus-rest AUCs is then 0.68: that is, if we uniformly choose x *without reference to the class*, and then choose x' uniformly from among all instances not of the same class as x', the expectation that $\hat{s}_i(x) > \hat{s}_i(x')$ is 0.68. This is lower than before, because it is now more likely that a random x comes from class C_3, whose scores do a worse ranking job.

We can obtain similar averages from one-versus-one AUCs. For instance, we can define AUC_{ij} as the AUC obtained using scores $\hat{s}_i$ to rank instances from classes C_i and C_j. Notice that $\hat{s}_j$ may rank these instances differently, and so $\text{AUC}_{ji} \neq \text{AUC}_{ij}$. Taking an unweighted average over all $i \neq j$ estimates the probability that, for uniformly chosen classes i and $j \neq i$, and uniformly chosen $x \in C_i$ and $x' \in C_j$, we have $\hat{s}_i(x) > \hat{s}_i(x')$. The weighted version of this estimates the probability that the instances are correctly ranked if we don't pre-select the class.

The simplest way to turn multi-class scores into classifications is by assigning the class that achieves the maximum score – that is, if $\hat{\mathbf{s}}(x) = (\hat{s}_1(x), \ldots, \hat{s}_k(x))$ is the score vector assigned to instance x and $m = \arg\max_i \hat{s}_i(x)$, then the class assigned to x is $\hat{c}(x) = C_m$. However, just as in the two-class case such a fixed decision rule can be suboptimal, and instead we may want to learn it from data. What this means is that we want to learn a weight vector $\mathbf{w} = (w_1, \ldots, w_k)$ to adjust the scores and assign $\hat{c}(x) = C_{m'}$ with $m' = \arg\max_i w_i \hat{s}_i(x)$ instead.[2] Since the weight vector can be multiplied with a constant without affecting m', we can fix one of the degrees of freedom by setting

[2] Notice that with two classes such a weighted decision rule assigns class C_1 if $w_1 \hat{s}_1(x) > w_2 \hat{s}_2(x)$, or equivalently, $\hat{s}_1(x)/\hat{s}_2(x) > w_2/w_1$. This can be interpreted as a threshold on suitably transformed scores,

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

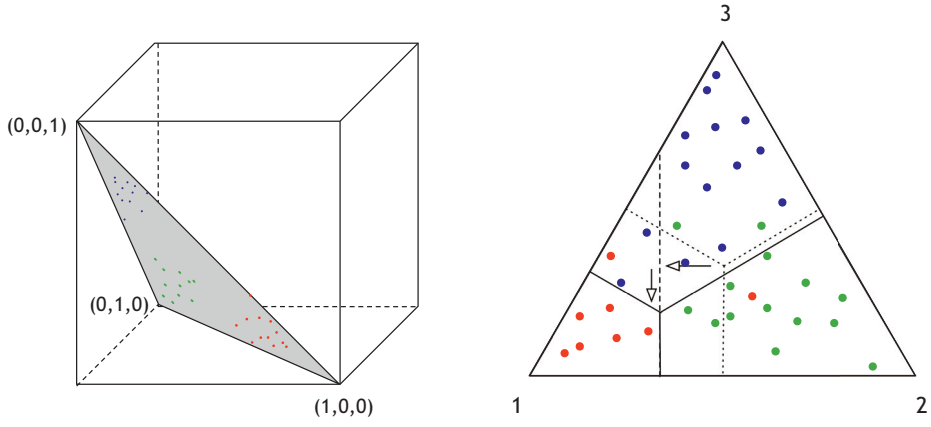

Figure 3.1. (left) Triples of probabilistic scores represented as points in an equilateral triangle connecting three corners of the unit cube. **(right)** The arrows show how the weights are adjusted from the initial equal weights (dotted lines), first by optimising the separation of C_2 against C_1 (dashed line), then by optimising the separation of C_3 against the other two classes (solid lines). The end result is that the weight of C_1 is considerably decreased, to the benefit of the other two classes.

$w_1 = 1$. Unfortunately, finding a globally optimal weight vector is computationally intractable. A heuristic approach that works well in practice is to first learn w_2 to optimally separate C_2 from C_1 as in the two-class case; then learn w_3 to separate C_3 from $C_1 \cup C_2$, and so on.

Example 3.6 (Reweighting multi-class scores). We illustrate the procedure for a three-class probabilistic classifier. The probability vectors $\hat{\mathbf{p}}(x) = \big(\hat{p}_1(x), \hat{p}_2(x), \hat{p}_3(x)\big)$ can be thought of as points inside the unit cube. Since the probabilities add up to 1, the points lie in an equilateral triangle connecting three corners of the cube (Figure 3.1 (left)). Each corner of this triangle represents one of the classes; the probability assigned to a particular class in a given point is proportional to the distance to the opposite side.

Any decision rule of the form $\arg\max_i w_i \hat{s}_i(x)$ cuts the triangle in three areas using lines perpendicular to the sides. For the unweighted decision rule these lines intersect in the triangle's centre of mass (Figure 3.1 (right)). Optimising the separation between C_2 against C_1 means moving this point along a line parallel to the base of the triangle, moving away from the class that receives greater weight. Once the optimal point on this line is found, we optimise the separation

so the weighted decision rule indeed generalises the two-class decision threshold.

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

of C_3 against the first two classes by moving in a direction perpendicular to the previous line.

Finally, we briefly look at the issue of obtaining calibrated multi-class probabilities. This is not a solved problem and several approaches have been suggested in the literature. One of the simplest and most robust of these calculates normalised coverage counts. Specifically, we take the summed or averaged coverage counts of all classifiers that fire, and normalise these to obtain probability vectors whose components sum to one. Equivalently, we can obtain probability vectors for each classifier separately, and take a weighted average of these with weights determined by the relative coverage of each classifier.

Example 3.7 (Multi-class probabilities from coverage counts). In Example 3.4 on p.87 we can divide the class counts by the total number of positive predictions. This results in the following class distributions: $(0.80, 0, 0.20)$ for the first classifier, $(0.10, 0.85, 0.05)$ for the second classifier, and $(0.29, 0.14, 0.57)$ for the third. The probability distribution associated with the combination of the first and third classifiers is

$$\frac{10}{24}(0.80, 0, 0.20) + \frac{14}{24}(0.29, 0.14, 0.57) = (0.50, 0.08, 0.42)$$

which is the same distribution as obtained by normalising the combined counts $(12, 2, 10)$. Similarly, the distribution associated with all three classifiers is

$$\frac{10}{44}(0.80, 0, 0.20) + \frac{20}{44}(0.10, 0.85, 0.05) + \frac{14}{44}(0.29, 0.14, 0.57) = (0.32, 0.43, 0.25)$$

Matrix notation describes this very succinctly as

$$\begin{pmatrix} 10/24 & 0 & 14/24 \\ 10/44 & 20/44 & 14/44 \end{pmatrix} \begin{pmatrix} 0.80 & 0.00 & 0.20 \\ 0.10 & 0.85 & 0.05 \\ 0.29 & 0.14 & 0.57 \end{pmatrix} = \begin{pmatrix} 0.50 & 0.08 & 0.42 \\ 0.32 & 0.43 & 0.25 \end{pmatrix}$$

The middle matrix is a row-normalised version of the middle matrix in Equation 3.1. *Row normalisation* works by dividing each entry by the sum of the entries in the row in which it occurs. As a result the entries in each row sum to one, which means that each row can be interpreted as a probability distribution. The left matrix combines two pieces of information: (*i*) which classifiers fire for each example (for instance, the second classifier doesn't fire for the first example); and

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

(*ii*) the coverage of each classifier. The right-hand side then gives the class distribution for each example. Notice that the product of row-normalised matrices again gives a row-normalised matrix.

In this section we have seen that many interesting issues arise, once we have more than two classes. The general way of addressing a k-class learning problem with binary classifiers is to (*i*) break the problem up into l binary learning problems; (*ii*) train l binary classifiers on two-class versions of the original data; and (*iii*) combine the predictions from these l classifiers into a single k-class prediction. The most common ways to do the first and third step is one-versus-one or one-versus-rest, but the use of code matrices gives the opportunity of implementing other schemes. We have also looked at ways of obtaining multi-class scores and probabilities from the binary classifiers, and discussed a heuristic method to calibrate the multi-class decision rule by reweighting.

This concludes our discussion of classification, arguably the most common task in machine learning. In the remainder of this chapter we will look at one more supervised predictive task in the next section, before we turn our attention to unsupervised and descriptive learning in Section 3.3.

3.2 Regression

In all the tasks considered so far – classification, scoring, ranking and probability estimation – the label space was a discrete set of classes. In this section we will consider the case of a real-valued target variable. A *function estimator*, also called a *regressor*, is a mapping $\hat{f} : \mathcal{X} \to \mathbb{R}$. The regression learning problem is to learn a function estimator from examples $(x_i, f(x_i))$. For instance, we might want to learn an estimator for the Dow Jones index or the FTSE 100 based on selected economic indicators.

While this may seem a natural and innocuous generalisation of discrete classification, it is not without its consequences. For one thing, we switched from a relatively low-resolution target variable to one with infinite resolution. Trying to match this precision in the function estimator will almost certainly lead to overfitting – besides, it is highly likely that some part of the target values in the examples is due to fluctuations that the model is unable to capture. It is therefore entirely reasonable to assume that the examples are noisy, and that the estimator is only intended to capture the general trend or shape of the function.

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

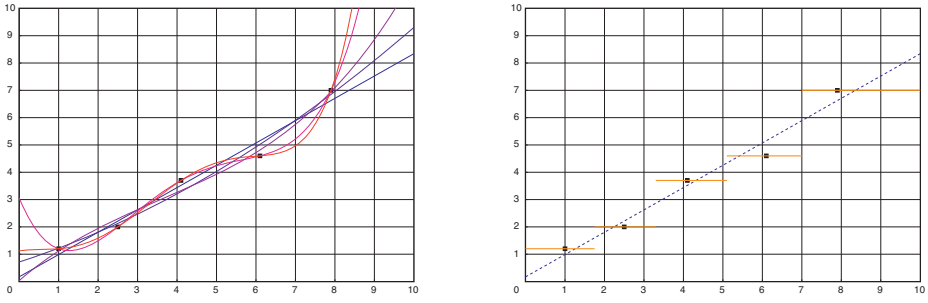

Figure 3.2. (left) Polynomials of different degree fitted to a set of five points. From bottom to top in the top right-hand corner: degree 1 (straight line), degree 2 (parabola), degree 3, degree 4 (which is the lowest degree able to fit the points exactly), degree 5. **(right)** A piecewise constant function learned by a grouping model; the dotted reference line is the linear function from the left figure.

Example 3.8 (Line fitting example). Consider the following set of five points:

x	y
1.0	1.2
2.5	2.0
4.1	3.7
6.1	4.6
7.9	7.0

We want to estimate y by means of a polynomial in x. Figure 3.2 (left) shows the result for degrees of 1 to 5 using ☞*linear regression*, which will be explained in Chapter 7. The top two degrees fit the given points exactly (in general, any set of n points can be fitted by a polynomial of degree no more than $n-1$), but they differ considerably at the extreme ends: e.g., the polynomial of degree 4 leads to a decreasing trend from $x = 0$ to $x = 1$, which is not really justified by the data.

To avoid overfitting the kind of data exemplified in Example 3.8 it is advisable to choose the degree of the polynomial as low as possible – often a simple linear relationship is assumed.

Regression is a task where the distinction between grouping and grading models comes to the fore. The philosophy of grouping models is to cleverly divide the instance space into segments and learn a local model in each segment that is as simple as possible. For instance, in decision trees the local model is a majority class classifier. In the

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

same spirit, to obtain a regression tree we could predict a constant value in each leaf. In the univariate problem of Example 3.8 this would result in the piecewise constant curve of Figure 3.2 (right). Notice that such a grouping model is able to fit the given points exactly, just as a polynomial of sufficiently high degree, and the same caveat regarding overfitting applies.

We can understand the phenomenon of overfitting a bit better by looking at the number of parameters that each model has. An n-degree polynomial has $n+1$ parameters: e.g., a straight line $y = a \cdot x + b$ has two parameters, and the polynomial of degree 4 that fits the five points exactly has five parameters. A piecewise constant model with n segments has $2n-1$ parameters: n y-values and $n-1$ x-values where the 'jumps' occur. So the models that are able to fit the points exactly are the models with more parameters. A rule of thumb is that, *to avoid overfitting, the number of parameters estimated from the data must be considerably less than the number of data points*.

We have seen that classification models can be evaluated by applying a loss function to the margins, penalising negative margins (misclassifications) and rewarding positive margins (correct classifications). Regression models are evaluated by applying a loss function to the *residuals* $f(x) - \hat{f}(x)$. Unlike classification loss functions a regression loss function will typically be symmetric around 0 (although it is conceivable that positive and negative residuals have different weights). The most common choice here is to take the squared residual as the loss function. This has the advantage of mathematical convenience, and can also be justified by the assumption that the observed function values are the true values contaminated by additive, normally distributed noise. However, it is well-known that squared loss is sensitive to outliers: you can see an example of this in Figure 7.2 on p.199.

If we underestimate the number of parameters of the model, we will not be able to decrease the loss to zero, regardless of how much training data we have. On the other hand, with a larger number of parameters the model will be more dependent on the training sample, and small variations in the training sample can result in a considerably different model. This is sometimes called the *bias–variance dilemma*: a low-complexity model suffers less from variability due to random variations in the training data, but may introduce a systematic bias that even large amounts of training data can't resolve; on the other hand, a high-complexity model eliminates such bias but can suffer non-systematic errors due to variance.

We can make this a bit more precise by noting that expected squared loss on a training example x can be decomposed as follows:[3]

$$\mathbb{E}\left[\left(f(x) - \hat{f}(x)\right)^2\right] = \left(f(x) - \mathbb{E}\left[\hat{f}(x)\right]\right)^2 + \mathbb{E}\left[\left(\hat{f}(x) - \mathbb{E}\left[\hat{f}(x)\right]\right)^2\right] \tag{3.2}$$

[3] The derivation expands the squared difference term, making use of the linearity of $\mathbb{E}[\cdot]$ and that $\mathbb{E}\left[f(x)\right] = f(x)$, after which terms can be rearranged to yield Equation 3.2.

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

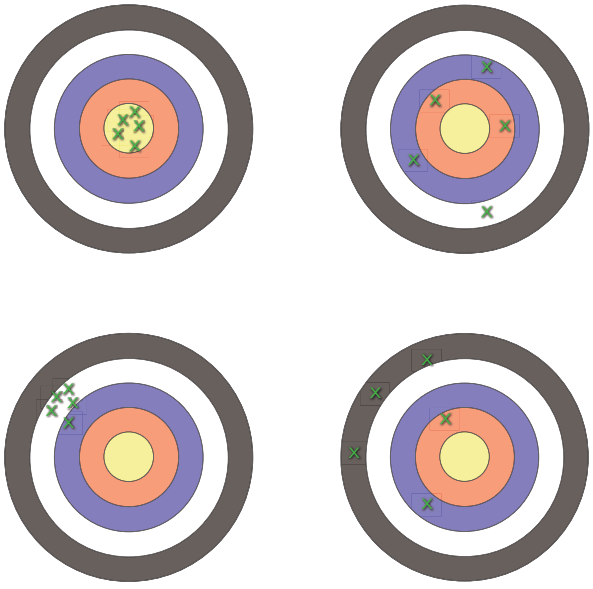

Figure 3.3. A dartboard metaphor illustrating the concepts of bias and variance. Each dartboard corresponds to a different learning algorithm, and each dart signifies a different training sample. The top row learning algorithms exhibit low bias, staying close to the bull's eye (the true function value for a particular x) on average, while the ones on the bottom row have high bias. The left column shows low variance and the right column high variance.

It is important to note that the expectation is taken over different training sets and hence different function estimators, but the learning algorithm and the example are fixed. The first term on the right-hand side in Equation 3.2 is zero if these function estimators get it right on average; otherwise the learning algorithm exhibits a systematic *bias* of some kind. The second term quantifies the *variance* in the function estimates $\hat{f}(x)$ as a result of variations in the training set. Figure 3.3 illustrates this graphically using a dartboard metaphor. The best situation is clearly achieved in the top left-hand corner of the figure, but in practice this is rarely achievable and we need to settle either for a low bias and a high variance (e.g., approximating the target function by a high-degree polynomial) or for a high bias and a low variance (e.g., using a linear approximation). We will return to the bias–variance dilemma at several places in the book: although the decomposition is not unique for most loss functions other than squared loss, it serves as a useful conceptual tool for understanding over- and underfitting.

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

3.3 Unsupervised and descriptive learning

So far, we have concerned ourselves exclusively with supervised learning of predictive models. That is, we learn a mapping from instance space $\mathscr{X}$ to output space $\mathscr{Y}$ using labelled examples $(x, l(x)) \in \mathscr{X} \times \mathscr{L}$ (or a noisy version thereof). This kind of learning is called 'supervised' because of the presence of the target variable $l(x)$ in the training data, which has to be supplied by a 'supervisor' or 'teacher' with some knowledge about the true labelling function l. Furthermore, the models are called 'predictive' because the outputs produced by the models are either direct estimates of the target variable or provide us with further information about its most likely value. Thus, we have only paid attention to the top-left entry in Table 3.1. In the remainder of this chapter we will briefly introduce the other three learning settings by means of selected examples:

- ☞ unsupervised learning of a predictive model in the form of predictive clustering;
- ☞ unsupervised learning of a descriptive model, exemplified by descriptive clustering and association rule discovery;
- ☞ supervised learning of a descriptive model, with subgroup discovery as practical realisation.

	Predictive model	*Descriptive model*
Supervised learning	classification, regression	**subgroup discovery**
Unsupervised learning	**predictive clustering**	**descriptive clustering**, **association rule discovery**

Table 3.1. The learning settings indicated in **bold** are introduced in the remainder of this chapter.

It is worthwhile reflecting for a moment on the nature of descriptive learning. The task here is to come up with a description of the data – to produce a descriptive model. It follows that the task output, being a model, is of the same kind as the learning output. Furthermore, it makes no sense to employ a separate training set to produce the descriptive model, as we want the model to describe our actual data rather than some hold-out set. In other words, *in descriptive learning the task and learning problem coincide* (Figure 3.4). This makes some things harder: for example, it is unlikely that a 'ground truth' or 'gold standard' is available to test the descriptive models against, and hence evaluating descriptive learning algorithms is much less straightforward than evaluating predictive ones. On the other hand, one could say that descriptive learning leads to the *discovery* of genuinely new knowledge, and it is often situated at the intersection of machine learning and data mining.

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

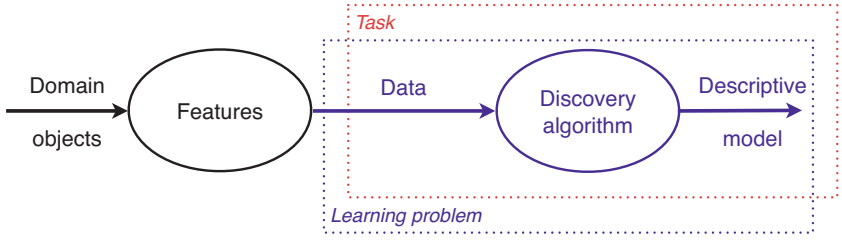

Figure 3.4. In descriptive learning the task and learning problem coincide: we do not have a separate training set, and the task is to produce a descriptive model of the data.

Predictive and descriptive clustering

The distinction between predictive and descriptive models can be clearly observed in clustering tasks. One way to understand clustering is as learning a new labelling function from unlabelled data. So we could define a 'clusterer' in the same way as a classifier, namely as a mapping $\hat{q} : \mathscr{X} \to \mathscr{C}$, where $\mathscr{C} = \{C_1, C_2, \ldots, C_k\}$ is a set of new labels. This corresponds to a *predictive* view of clustering, as the domain of the mapping is the entire instance space, and hence it generalises to unseen instances. A *descriptive* clustering model learned from given data $D \subseteq \mathscr{X}$ would be a mapping $\hat{q} : D \to \mathscr{C}$ whose domain is D rather than $\mathscr{X}$. In either case the labels have no intrinsic meaning, other than to express whether two instances belong to the same cluster. So an alternative way to define a clusterer is as an equivalence relation $\hat{q} \subseteq \mathscr{X} \times \mathscr{X}$ or $\hat{q} \subseteq D \times D$ (see Background 2.1 on p.51 for the definition of an equivalence relation), or, equivalently, as a partition of $\mathscr{X}$ or D.

The distinction between predictive and descriptive clustering is subtle and not always articulated clearly in the literature. Several well-known clustering algorithms including ☞*K-means* (discussed in more detail in Chapter 8) learn a predictive clustering. Thus, they learn a clustering model from training data that can subsequently be used to assign new data to clusters. This is in keeping with our distinction between the task (clustering arbitrary data) and the learning problem (learning a clustering model from training data). However, this distinction isn't really applicable to descriptive clustering methods: here, the clustering model learned from D can only be used to cluster D. In effect, the task becomes learning a suitable clustering model for the given data.

Without any further information, any clustering is as good as any other. What distinguishes a good clustering is that the data is partitioned into *coherent* groups or clusters. 'Coherence' here means that, on average, two instances from the same cluster have more in common – are more similar – than two instances from different clusters. This assumes some way of assessing the similarity or, as is usually more convenient, the dissimilarity or distance of an arbitrary pair of instances. If our features are numerical, i.e., $\mathscr{X} = \mathbb{R}^d$, the most obvious distance measure is Euclidean distance, but

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

other choices are possible, some of which generalise to non-numerical features. Most distance-based clustering methods depend on the possibility of defining a 'centre of mass' or *exemplar* for an arbitrary set of instances, such that the exemplar minimises some distance-related quantity over all instances in the set, called its *scatter*. A good clustering is then one where the scatter summed over each cluster – the *within-cluster scatter* – is much smaller than the scatter of the entire data set.

This analysis suggests a definition of the clustering problem as finding a partition $D = D_1 \uplus \ldots \uplus D_K$ that minimises the within-cluster scatter. However, there are a few issues with this definition:

- ☞ the problem as stated has a trivial solution: set $K = |D|$ so that each 'cluster' contains a single instance from D and thus has zero scatter;
- ☞ if we fix the number of clusters K in advance, the problem cannot be solved efficiently for large data sets (it is NP-hard).

The first problem is the clustering equivalent of overfitting the training data. It could be dealt with by penalising large K. Most approaches, however, assume that an educated guess of K can be made. This leaves the second problem, which is that finding a globally optimal solution is intractable for larger problems. This is a well-known situation in computer science and can be dealt with in two ways:

- ☞ by applying a heuristic approach, which finds a 'good enough' solution rather than the best possible one;
- ☞ by relaxing the problem into a 'soft' clustering problem, by allowing instances a degree of membership in more than one cluster.

Most clustering algorithms follow the heuristic route, including the K-means algorithm. The soft clustering approach can be addressed in various ways, including ☞*Expectation-Maximisation* (Section 9.4) and ☞*matrix decomposition* (Section 10.3). Figure 3.5 illustrates the heuristic and soft clustering approaches. Notice that a soft clustering generalises the notion of a partition, in the same way that a probability estimator generalises a classifier.

The representation of clustering models depends on whether they are predictive, descriptive or soft. A descriptive clustering of n data points into c clusters could be represented by a *partition matrix*: an n-by-c binary matrix with exactly one 1 in each row (and at least one 1 in each column, otherwise there would be empty clusters). A soft clustering corresponds to a row-normalised n-by-c matrix. A predictive clustering partitions the whole instance space and is therefore not suitable for a matrix representation. Typically, predictive clustering methods represent a cluster by their *centroid* or *exemplar*: in that case, the cluster boundaries are a set of straight lines called a *Voronoi*

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

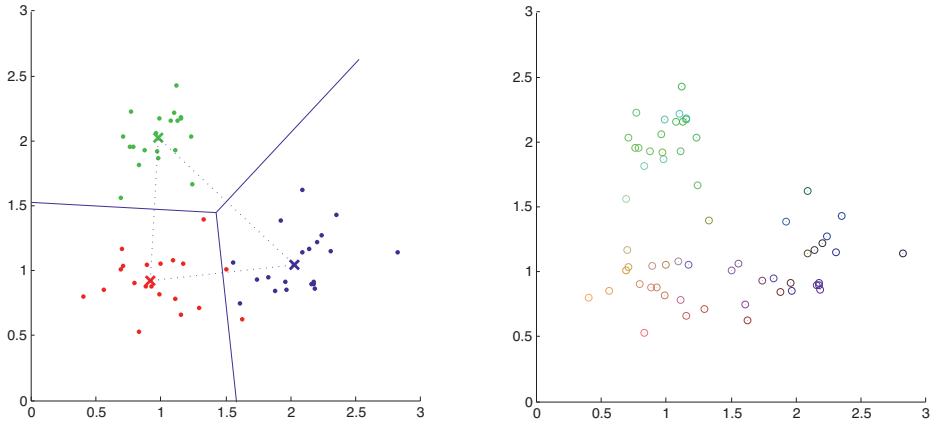

Figure 3.5. (left) An example of a predictive clustering. The coloured dots were sampled from three bivariate Gaussians centred at $(\mathbf{1},\mathbf{1})$, $(\mathbf{1},\mathbf{2})$ and $(\mathbf{2},\mathbf{1})$. The crosses and solid lines are the cluster exemplars and cluster boundaries found by 3-means. **(right)** A soft clustering of the same data found by matrix decomposition.

diagram (Figure 3.5 (left)). More generally, each cluster could be represented by a probability density, with the boundaries occurring where densities of neighbouring clusters are equal; this would allow non-linear cluster boundaries.

Example 3.9 (Representing clusterings). The cluster exemplars in Figure 3.5 (left) can be given as a c-by-2 matrix:

$$\begin{pmatrix} 0.92 & 0.93 \\ 0.98 & 2.02 \\ 2.03 & 1.04 \end{pmatrix}$$

The following n-by-c matrices represent a descriptive clustering (left) and a soft clustering (right) of given data points:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ \cdots & \cdots & \cdots \end{pmatrix} \qquad \begin{pmatrix} 0.40 & 0.30 & 0.30 \\ 0.40 & 0.51 & 0.09 \\ 0.44 & 0.29 & 0.27 \\ 0.35 & 0.08 & 0.57 \\ \cdots & \cdots & \cdots \end{pmatrix}$$

---

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

An interesting question is how clustering models should be evaluated. In the absence of labelled data we cannot use a test set in the same way as we would in classification or regression. We can use within-cluster scatter as a measure of the quality of a clustering. For a predictive clustering it is possible to evaluate within-cluster scatter on hold-out data that wasn't used to build the clusters in the first place. An alternative way of evaluating a clustering arises if we have some knowledge about instances that should, or should not, be clustered together.

Example 3.10 (Evaluating clusterings). Suppose we have five test instances that we think should be clustered as $\{e1, e2\}, \{e3, e4, e5\}$. So out of the $5 \cdot 4 = 20$ possible pairs, 4 are considered 'must-link' pairs and the other 16 as 'must-not-link' pairs. The clustering to be evaluated clusters these as $\{e1, e2, e3\}, \{e4, e5\}$ – so two of the must-link pairs are indeed clustered together ($e1$–$e2$, $e4$–$e5$), the other two are not ($e3$–$e4$, $e3$–$e5$), and so on.

We can tabulate this as follows:

	Are together	*Are not together*	
Should be together	**2**	**2**	4
Should not be together	**2**	**14**	16
	4	16	20

We can now treat this as a two-by-two contingency table, and evaluate it accordingly. For instance, we can take the proportion of pairs on the 'good' diagonal, which is $16/20 = 0.8$. In classification we would call this accuracy, but in the clustering context this is known as the *Rand index*.

Note that there are usually many more must-not-link pairs than must-link pairs, and it is a good idea to compensate for this. One way to do that is to calculate the harmonic mean of precision and recall (the latter the same as true positive rate, see Table 2.3 on p.57), which in the information retrieval literature is known as the *F-measure*.[4] Precision is calculated on the left column of the contingency table and recall on the top row; as a result the bottom right-hand cell (the must-not-link pairs that are correctly not clustered together) are ignored, which is precisely what we want. In the example both precision and recall are $2/4 = 0.5$, and so is the F-measure. This shows that the relatively good Rand index is mostly accounted for by the must-not-link pairs that end up in different clusters.

[4] The harmonic mean of precision and recall is $\frac{2}{1/prec + 1/rec} = \frac{2 prec \cdot rec}{prec + rec}$. The harmonic mean is appropriate for averaging ratios; see Background 10.1 on p.300.

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

Other descriptive models

To wrap up our catalogue of machine learning tasks we will briefly look at two other descriptive models, one learned in a supervised fashion from labelled data and the other entirely unsupervised.

Subgroup models don't try to approximate the labelling function, but rather aim at identifying subsets of the data exhibiting a class distribution that is significantly different from the overall population. Formally, a *subgroup* is a mapping $\hat{g} : D \to \{\mathsf{true}, \mathsf{false}\}$ and is learned from a set of labelled examples $(x_i, l(x_i))$, where $l : \mathscr{X} \to \mathscr{C}$ is the true labelling function. Note that $\hat{g}$ is the characteristic function of the set $G = \{x \in D | \hat{g}(x) = \mathsf{true}\}$, which is called the *extension* of the subgroup. Note also that we used the given data D rather than the whole instance space $\mathscr{X}$ for the domain of a subgroup, since it is a descriptive model.

Example 3.11 (Subgroup discovery). Imagine you want to market the new version of a successful product. You have a database of people who have been sent information about the previous version, containing all kinds of demographic, economic and social information about those people, as well as whether or not they purchased the product. If you were to build a classifier or ranker to find the most likely customers for your product, it is unlikely to outperform the majority class classifier (typically, relatively few people will have bought the product). However, what you are really interested in is finding reasonably sized subsets of people with a proportion of customers that is significantly higher than in the overall population. You can then target those people in your marketing campaign, ignoring the rest of your database.

A subgroup is essentially a binary classifier, and so one way to develop a subgroup discovery system is to adapt an existing classifier training algorithm. This may not involve much more than adapting the search heuristic to reflect the specific objective of a subgroup (to identify subsets of the data with a significantly different class distribution). However, this would only give us a single subgroup. Rule learners are particularly appropriate for subgroup discovery since every rule can be interpreted as a separate subgroup.

How do we distinguish interesting subgroups from uninteresting ones? This can be determined by constructing a contingency table similar to the ones we use in binary classification. For three classes such a table looks as follows:

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

	In subgroup	*Not in subgroup*	
Labelled C_1	g_1	$C_1 - g_1$	C_1
Labelled C_2	g_2	$C_2 - g_2$	C_2
Labelled C_3	g_3	$C_3 - g_3$	C_3
	$\lvert G\rvert$	$\lvert D\rvert - \lvert G\rvert$	$\lvert D\rvert$

where $g_i = |\{x \in D | \hat{g}(x) = \mathsf{true} \wedge l(x) = C_i\}|$ and C_i is shorthand for $|\{x \in D | l(x) = C_i\}|$. From here there are a number of possibilities. One idea is to measure the extent to which the class distribution in the left column is different from the class distribution in the row marginals (the right-most column). As we shall see later (Example 6.6 on p.180), this boils down to using an adaptation of average recall as evaluation measure. Another idea is to treat the subgroup as a decision tree split and borrow splitting criteria from ☞*decision tree* learning (Section 5.1). It is also possible to use the χ^2 statistic to evaluate the extent to which each g_i differs from what would be expected on the basis of the marginals C_i and $|G|$. What these evaluation measures have in common is that they prefer different class distributions in the subgroup and its complement from the overall distribution in D, and also larger subgroups over smaller ones. Most of these measures are actually symmetric in that they assign the same evaluation to a subgroup and its complement, from which it follows that they also prefer larger complements over smaller ones – in other words, they prefer subgroups that are about half the size of the data (other things being equal).

I will now give an example of unsupervised learning of descriptive models. Associations are things that usually occur together. For example, in market basket analysis we are interested in items frequently bought together. An example of an association rule is ·**if** beer **then** crisps·, stating that customers who buy beer tend to also buy crisps. Association rule discovery starts with identifying feature values that often occur together. There is some superficial similarity with subgroups here, but these so-called frequent item sets are identified in a purely unsupervised manner, without need for labelled training data. Item sets then give rise to rules describing co-occurrences between feature values. These association rules are if-then rules similar to classification rules, except that the then-part isn't restricted to a particular class variable and can contain any feature (or even several features). Rather than adapting a given learning algorithm we need a new algorithm that first finds frequent item sets and then turns them into association rules. The process needs to take into account a mix of statistics in order to avoid generating trivial rules.

Example 3.12 (Association rule discovery). In a motorway service station most clients will buy petrol. This means that there will be many frequent item sets

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

involving petrol, such as {newspaper, petrol}. This might suggest the construction of an association rule ·**if** newspaper **then** petrol· – however, this is predictable given that {petrol} is already a frequent item set (and clearly at least as frequent as {newspaper, petrol}). Of more interest would be the converse rule ·**if** petrol **then** newspaper· which expresses that a considerable proportion of the people buying petrol also buy a newspaper.

We clearly see a relationship with subgroup discovery in that association rules also identify subsets that have a different distribution when compared with the full data set, namely with respect to the then-part of the rule. The difference is that the then-part is not a fixed target variable but it is found as part of the discovery process. Both subgroup discovery and association rule discovery will be discussed in the context of rule learning in Section 6.3.

3.4 Beyond binary classification: Summary and further reading

While binary classification is an important task in machine learning, there are many other relevant tasks and in this chapter we looked at a number of them.

☞ In Section 3.1 we considered classification tasks with more than two classes. We shall see in the coming chapters that some models handle this situation very naturally, but if our models are essentially two-class (such as linear models) we have to approach it via a combination of binary classification tasks. One key idea is the use of a code matrix to combine the results of several binary classifiers, as proposed by Dietterich and Bakiri (1995) under the name 'error-correcting output codes' and developed by Allwein *et al.* (2000). We also looked at ways to obtain scores for more than two classes and to evaluate those scores using multi-class adaptations of the area under the ROC curve. One of these multi-class extensions of AUC was proposed and analysed by Hand and Till (2001). The heuristic procedure for reweighting multi-class scores in Example 3.6 on p.89 was proposed by Lachiche and Flach (2003); Bourke *et al.* (2008) demonstrated that it achieves good performance in comparison with a number of alternative approaches.

☞ Section 3.2 was devoted to regression: predicting a real-valued target value. This is a classical data analysis problem that was already studied by Carl Friedrich Gauss in the late eighteenth century. It is natural to use a quadratic loss function on the residuals, although this carries with it a certain sensitivity to outliers. Grading models are most common here, although it is also possible to

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press

learn a grouping model that divides the instance space into segments that admit a simple local model. Since it is often possible to fit a set of points exactly (e.g., with a high-degree polynomial), care must be taken to avoid overfitting. Finding the right balance between over- and underfitting is sometimes called the bias–variance dilemma; an extensive discussion (including the dartboard metaphor) can be found in Rajnarayan and Wolpert (2010).

☞ In Section 3.3 we considered unsupervised and descriptive learning tasks. We saw that in descriptive learning the task and learning problem coincide. A clustering model can be either predictive or descriptive: in the former case it is meant to construct classes in a wholly unsupervised manner, after which the learned model can be applied to unseen data in the usual way. Descriptive clustering, on the other hand, only applies to the data at hand. It should be noted that the distinction between predictive and descriptive clustering is not universally recognised in the literature; sometimes the term 'predictive clustering' is used in the slightly different sense of clustering simultaneously on the target variable and the features (Blockeel *et al.*, 1998).

☞ Like descriptive clustering, association rule discovery is another descriptive task which is wholly unsupervised. It was introduced by Agrawal, Imielinski and Swami (1993) and has given rise to a very large body of work in the data mining literature. Subgroup discovery is a form of supervised learning of descriptive models aimed at finding subsets of the data with a significantly different distribution of the target variable. It was first studied by Klösgen (1996) and extended to the more general notion of exceptional model mining in order to deal with, e.g., real-valued target variables by Leman *et al.* (2008). More generally, unsupervised learning of descriptive models is a large subject that was pioneered by Tukey (1977).

https://doi.org/10.1017/CBO9780511973000.005 Published online by Cambridge University Press