

---

# Beyond binary classification

---

**T**HE 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.

### 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):

		<i>Predicted</i>			
		15	2	3	20
<i>Actual</i>	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

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  $C_2, \dots, C_n$ , 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  $C_{i+1}, \dots, C_n$  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-

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.<sup>1</sup> The predicted class for word  $w$  is then  $\operatorname{argmin}_j d(w, c_j)$ , 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-

<sup>1</sup>This is a slight generalisation of the *Hamming distance* for binary strings, which counts the number of positions in which the two strings differ.

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*.

**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 classifier. [Example 3.4](#) illustrates this.

**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:

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

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.

**Example 3.5 (Multi-class AUC).** Assume we have a multi-class scoring classifier that produces a  $k$ -vector of scores  $\hat{\mathbf{s}}(x) = (\hat{s}_1(x), \dots, \hat{s}_k(x))$  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.

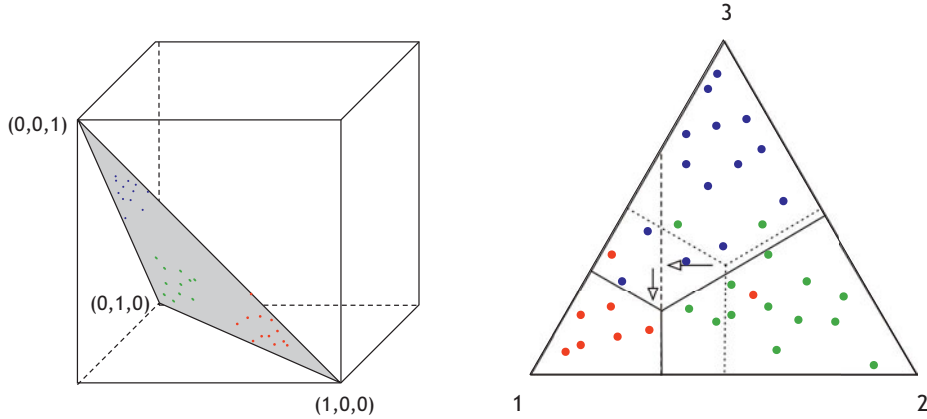
Suppose now  $C_1$  has 10 instances,  $C_2$  has 20 and  $C_3$  70. The weighted average of the one-versus-rest **AUCs** is then 0.68: that is, if we uniformly choose  $x$  *without reference to the class*, and then choose  $x'$  uniformly from among all instances not of the same class as  $x$ , the 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 **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  $\hat{\mathbf{s}}(x) = (\hat{s}_1(x), \dots, \hat{s}_k(x))$  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 sub-optimal, and instead we may want to learn it from data. What this means is that we want to learn a weight vector  $\mathbf{w} = (w_1, \dots, w_k)$  to adjust the scores and assign  $\hat{c}(x) = C_{m'}$  with  $m' = \arg\max_i w_i \hat{s}_i(x)$  instead.<sup>2</sup> Since the weight vector can be multiplied with a constant without affecting  $m'$ , we can fix one of the degrees of freedom by setting

<sup>2</sup>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,





**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  $\hat{\mathbf{p}}(x) = (\hat{p}_1(x), \hat{p}_2(x), \hat{p}_3(x))$  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.

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

(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.

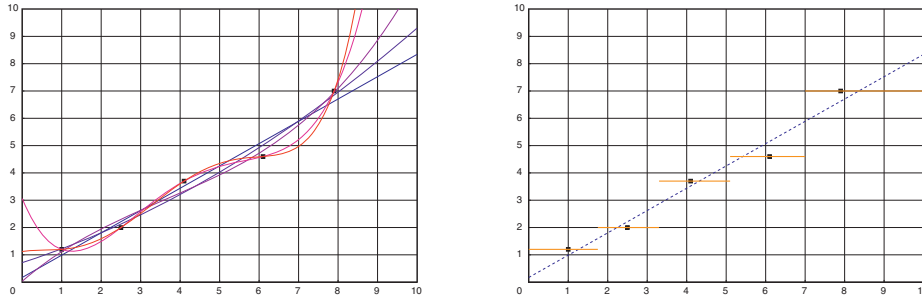
In this section we have seen that many interesting issues arise, once we have more than two classes. The general way of addressing a  $k$ -class learning problem with binary classifiers is to (i) break the problem up into  $l$  binary learning problems; (ii) train  $l$  binary classifiers on two-class 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  $\hat{f} : \mathcal{X} \rightarrow \mathbb{R}$ . 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.



**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

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*.

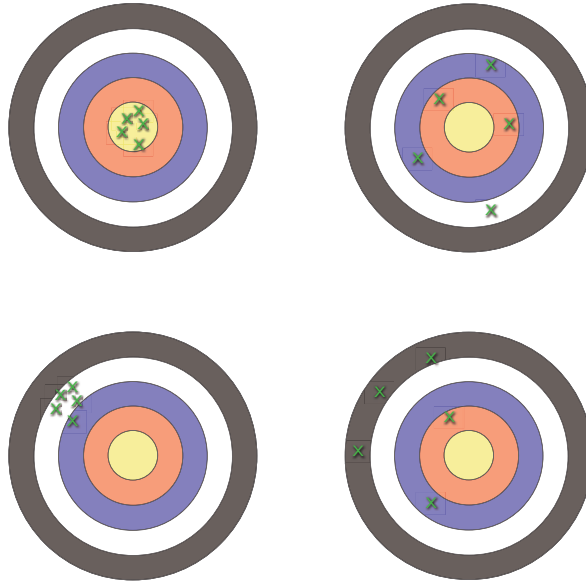
We have seen that classification models can be evaluated by applying a loss function to the margins, penalising negative margins (misclassifications) and rewarding positive margins (correct classifications). Regression models are evaluated by applying a loss function to the *residuals*  $f(x) - \hat{f}(x)$ . 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:<sup>3</sup>

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

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



**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.

### 3.3 Unsupervised and descriptive learning

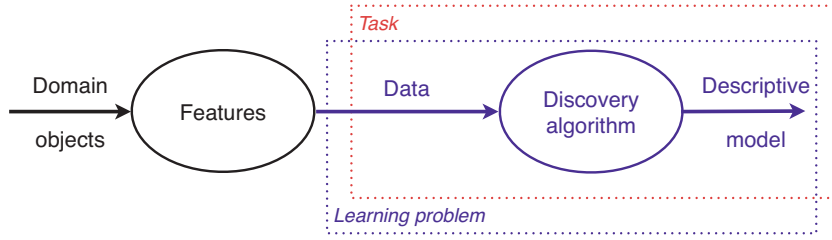
So far, we have concerned ourselves exclusively with supervised learning of predictive models. That is, we learn a mapping from instance space  $\mathcal{X}$  to output space  $\mathcal{Y}$  using labelled examples  $(x, l(x)) \in \mathcal{X} \times \mathcal{L}$  (or a noisy version thereof). This kind 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.

	<i>Predictive model</i>	<i>Descriptive model</i>
<i>Supervised learning</i>	classification, regression	<b>subgroup discovery</b>
<i>Unsupervised learning</i>	<b>predictive clustering</b>	<b>descriptive clustering,</b> <b>association rule discovery</b>

**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.



**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  $\hat{q} : \mathcal{X} \rightarrow \mathcal{C}$ , where  $\mathcal{C} = \{C_1, C_2, \dots, C_k\}$  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  $D \subseteq \mathcal{X}$  would be a mapping  $\hat{q} : D \rightarrow \mathcal{C}$  whose domain is  $D$  rather than  $\mathcal{X}$ . 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  $\hat{q} \subseteq \mathcal{X} \times \mathcal{X}$  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  $\mathcal{X}$  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.,  $\mathcal{X} = \mathbb{R}^d$ , the most obvious distance measure is Euclidean distance, but



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  $D = D_1 \uplus \dots \uplus D_K$  that minimises the within-cluster scatter. However, there are a few issues with this definition:

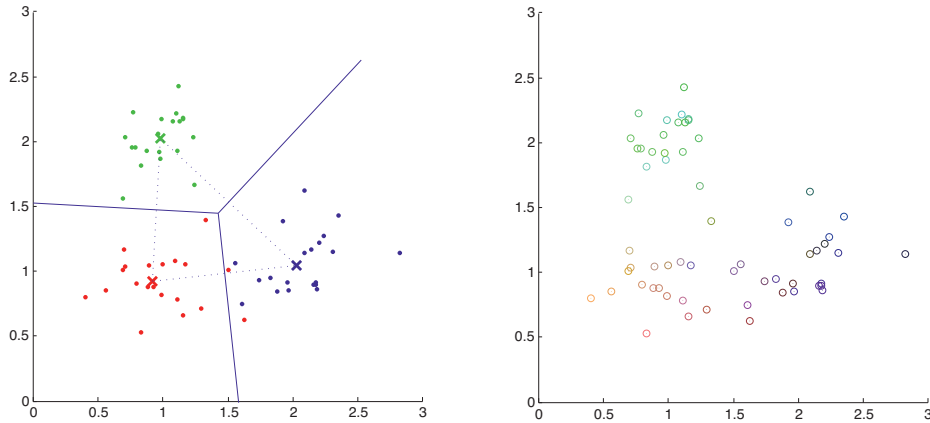
- ☞ the problem as stated has a trivial solution: set  $K = |D|$  so that each ‘cluster’ contains a single instance from  $D$  and thus has zero scatter;
- ☞ if we fix the number of clusters  $K$  in advance, the problem cannot be solved efficiently for large data sets (it is NP-hard).

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*



**Figure 3.5. (left)** An example of a predictive clustering. The coloured dots were sampled from three bivariate Gaussians centred at  $(1, 1)$ ,  $(1, 2)$  and  $(2, 1)$ . 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:

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

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:

	<i>Are together</i>	<i>Are not together</i>	
<i>Should be together</i>	<b>2</b>	<b>2</b>	4
<i>Should not be together</i>	<b>2</b>	<b>14</b>	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*.<sup>4</sup> 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.

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

### 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  $\hat{g} : D \rightarrow \{\text{true}, \text{false}\}$  and is learned from a set of labelled examples  $(x_i, l(x_i))$ , where  $l : \mathcal{X} \rightarrow \mathcal{C}$  is the true labelling function. Note that  $\hat{g}$  is the characteristic function of the set  $G = \{x \in D \mid \hat{g}(x) = \text{true}\}$ , which is called the *extension* of the subgroup. Note also that we used the given data  $D$  rather than the whole instance space  $\mathcal{X}$  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:

	<i>In subgroup</i>	<i>Not in subgroup</i>	
<i>Labelled</i> $C_1$	$g_1$	$C_1 - g_1$	$C_1$
<i>Labelled</i> $C_2$	$g_2$	$C_2 - g_2$	$C_2$
<i>Labelled</i> $C_3$	$g_3$	$C_3 - g_3$	$C_3$
	$ G $	$ D  -  G $	$ D $

where  $g_i = |\{x \in D | \hat{g}(x) = \text{true} \wedge l(x) = C_i\}|$  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  $\chi^2$  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

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

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 \(1977\)](#).



---

# Concept learning

---

**H**AVING DISCUSSED A VARIETY of tasks in the preceding two chapters, we are now in an excellent position to start discussing machine learning models and algorithms for learning them. This chapter and the next two are devoted to logical models, the hallmark of which is that they use logical expressions to divide the instance space into segments and hence construct grouping models. The goal is to find a segmentation such that the data in each segment is more homogeneous, with respect to the task to be solved. For instance, in classification we aim to find a segmentation such that the instances in each segment are predominantly of one class, while in regression a good segmentation is such that the target variable is a simple function of a small number of predictor variables. There are essentially two kinds of logical models: tree models and rule models. Rule models consist of a collection of implications or if-then rules, where the if-part defines a segment, and the then-part defines the behaviour of the model in this segment. Tree models are a restricted kind of rule model where the if-parts of the rules are organised in a tree structure.

In this chapter we consider methods for learning logical expressions or *concepts* from examples, which lies at the basis of both tree models and rule models. In concept learning we only learn a description for the positive class, and label everything that doesn't satisfy that description as negative. We will pay particular attention to the generality ordering that plays an important role in logical models. In the next two chapters



The simplest logical expressions are equalities of the form **Feature = Value** and, for numerical features, inequalities of the form **Feature < Value**; these are called *literals*. Complex Boolean expressions can be built using logical connectives: *conjunction*  $\wedge$ , *disjunction*  $\vee$ , *negation*  $\neg$  and *implication*  $\rightarrow$ . The following equivalences hold (the left two are called the *De Morgan laws*):

$$\begin{aligned}\neg(A \wedge B) &\equiv \neg A \vee \neg B & \neg\neg A &\equiv A \\ \neg(A \vee B) &\equiv \neg A \wedge \neg B & A \rightarrow B &\equiv \neg A \vee B\end{aligned}$$

If Boolean expression  $A$  is true of instance  $x$ , we say that  $A$  *covers*  $x$ . The set of instances covered by expression  $A$  is called its *extension* and denoted  $\mathcal{X}_A = \{x \in \mathcal{X} \mid A \text{ covers } x\}$ , where  $\mathcal{X}$  denotes the instance space which acts as the universe of discourse (see [Background 2.1](#) on p.51). There is a direct correspondence between logical connectives and operations on sets: e.g.,  $\mathcal{X}_{A \wedge B} = \mathcal{X}_A \cap \mathcal{X}_B$ ,  $\mathcal{X}_{A \vee B} = \mathcal{X}_A \cup \mathcal{X}_B$  and  $\mathcal{X}_{\neg A} = \mathcal{X} \setminus \mathcal{X}_A$ . If  $\mathcal{X}_A \supseteq \mathcal{X}_{A'}$ , we say that  $A$  is *at least as general as*  $A'$ , and if in addition  $\mathcal{X}_A \not\subseteq \mathcal{X}_{A'}$  we say that  $A$  is *more general than*  $A'$ . This *generality ordering* is a partial order on logical expressions as defined in [Background 2.1](#). (More precisely: it is a partial order on the equivalence classes of the relation of logical equivalence  $\equiv$ .)

A *clause* is an implication  $P \rightarrow Q$  such that  $P$  is a conjunction of literals and  $Q$  is a disjunction of literals. Using the equivalences above we can rewrite such an implication as

$$(A \wedge B) \rightarrow (C \vee D) \equiv \neg(A \wedge B) \vee (C \vee D) \equiv \neg A \vee \neg B \vee C \vee D$$

and hence a clause can equivalently be seen as a disjunction of literals or their negations. Any logical expression can be rewritten as a conjunction of clauses; this is referred to as *conjunctive normal form* (CNF). Alternatively, any logical expression can be written as a disjunction of conjunctions of literals or their negation; this is called *disjunctive normal form* (DNF). A *rule* is a clause  $A \rightarrow B$  where  $B$  is a single literal; this is also often referred to as a *Horn clause*, after the American logician Alfred Horn.

---

**Background 4.1.** Some logical concepts and notation.

---

we consider tree and rule models, which go considerably beyond concept learning as they can handle multiple classes, probability estimation, regression, as well as clustering tasks.

## 4.1 The hypothesis space

The simplest concept learning setting is where we restrict the logical expressions describing concepts to conjunctions of literals (see [Background 4.1](#) for a review of important definitions and notation from logic). The following example illustrates this.<sup>1</sup>

---

**Example 4.1 (Learning conjunctive concepts).** Suppose you come across a number of sea animals that you suspect belong to the same species. You observe their length in metres, whether they have gills, whether they have a prominent beak, and whether they have few or many teeth. Using these features, the first animal can be described by the following conjunction:

$$\text{Length} = 3 \wedge \text{Gills} = \text{no} \wedge \text{Beak} = \text{yes} \wedge \text{Teeth} = \text{many}$$

The next one has the same characteristics but is a metre longer, so you drop the length condition and generalise the conjunction to

$$\text{Gills} = \text{no} \wedge \text{Beak} = \text{yes} \wedge \text{Teeth} = \text{many}$$

The third animal is again 3 metres long, has a beak, no gills and few teeth, so your description becomes

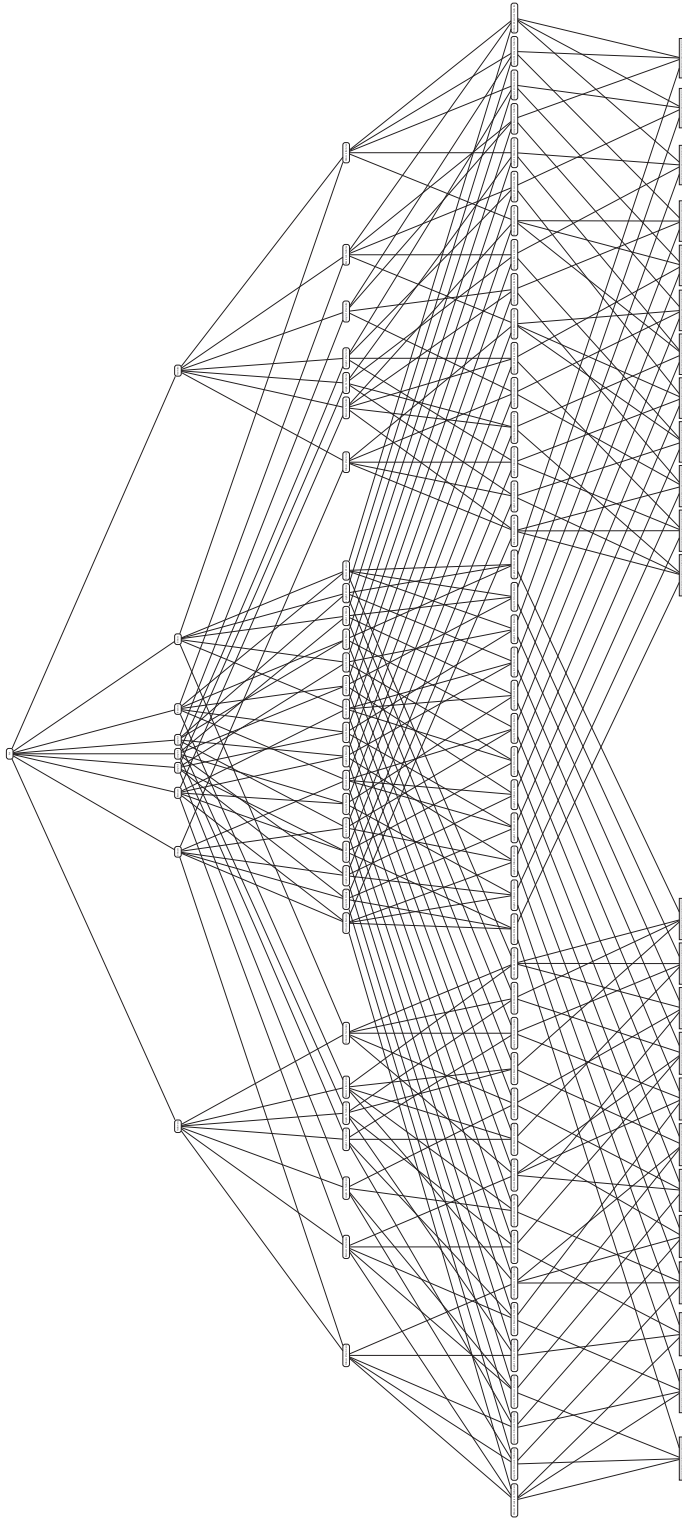
$$\text{Gills} = \text{no} \wedge \text{Beak} = \text{yes}$$

All remaining animals satisfy this conjunction, and you finally decide they are some kind of dolphin.

---

Despite the simplicity of this example, the space of possible concepts – usually called the *hypothesis space* – is already fairly large. Let's assume we have three different lengths: 3, 4 and 5 metres, while the other three features have two values each. We then have  $3 \cdot 2 \cdot 2 \cdot 2 = 24$  possible instances. How many conjunctive concepts are there using these same features? We can answer this question if we treat the absence of a feature as an additional 'value'. This gives a total of  $4 \cdot 3 \cdot 3 \cdot 3 = 108$  different concepts. While this seems quite a lot, you should realise that the number of possible extensions – sets of instances – is much larger:  $2^{24}$ , which is more than 16 million! That is, if you pick a random set of instances, the odds that you can't find a conjunctive concept that exactly describes those instances are well over 100 000 to 1. This is actually a good thing, as it forces the learner to generalise beyond the training data and cover instances that it hasn't seen before. [Figure 4.1](#) depicts this hypothesis space, making use of the general-

<sup>1</sup>Inspired by [www.cwtstrandings.org](http://www.cwtstrandings.org).



**Figure 4.1.** The hypothesis space corresponding to Example 4.1. The bottom row corresponds to the 24 possible instances, which are complete conjunctions with four literals each. The next row up are all 44 concepts with three literals each; then 30 concepts with two literals each; 9 concepts consisting of a single literal; and the top concept is the empty conjunction which is always true and hence covers all possible instances. A connecting line is drawn between concepts on consecutive layers if the higher one is more general than the lower one (i.e., the higher concept's extension is a superset of the lower's).

ity ordering (i.e., the subset relationship between concept extensions; see [Background 4.1](#)).

### Least general generalisation

If we rule out all concepts that don't cover at least one of the instances in [Example 4.1](#), the hypothesis space is reduced to 32 conjunctive concepts ([Figure 4.2](#)). Insisting that any hypothesis cover all three instances reduces this further to only four concepts, the least general one of which is the one found in the example – it is called their *least general generalisation* (LGG). [Algorithm 4.1](#) formalises the procedure, which is simply to repeatedly apply a pairwise LGG operation ([Algorithm 4.2](#)) to an instance and the current hypothesis, as they both have the same logical form. The structure of the hypothesis space ensures that the result is independent of the order in which the instances are processed.

Intuitively, the LGG of two instances is the nearest concept in the hypothesis space where paths upward from both instances intersect. The fact that this point is unique is a special property of many logical hypothesis spaces, and can be put to good use in learning. More precisely, such a hypothesis space forms a *lattice*: a partial order in which each two elements have a *least upper bound* (*lub*) and a *greatest lower bound* (*glb*). So, the LGG of a set of instances is exactly the least upper bound of the instances in that lattice. Furthermore, it is the greatest lower bound of the set of all generalisations of the instances: all possible generalisations are at least as general as the LGG. In this very precise sense, *the LGG is the most conservative generalisation that we can learn from the data*.

If we want to be a bit more adventurous, we could choose one of the more general hypotheses, such as *Gills = no* or *Beak = yes*. However, we probably don't want to choose the most general hypothesis, which is simply that every animal is a dolphin,

---

**Algorithm 4.1:** LGG-Set( $D$ ) – find least general generalisation of a set of instances.

---

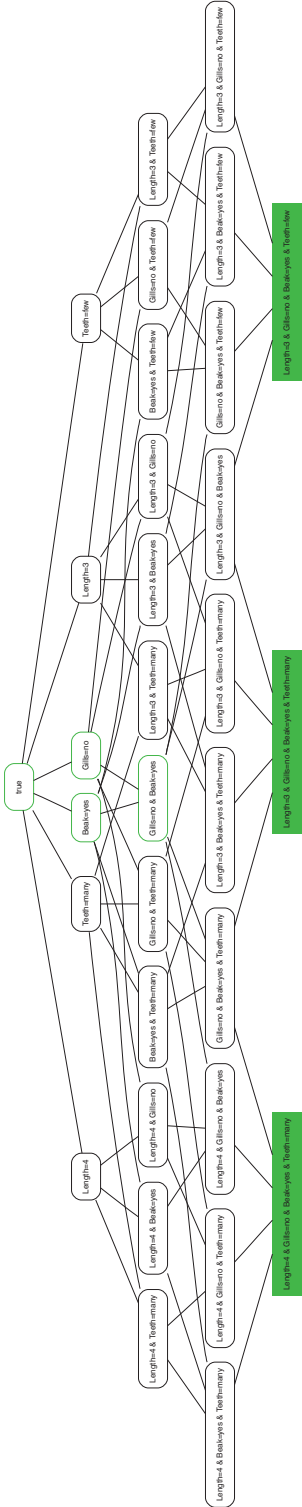
**Input** : data  $D$ .

**Output** : logical expression  $H$ .

```

1  $x \leftarrow$  first instance from  $D$ ;
2  $H \leftarrow x$ ;
3 while instances left do
4    $x \leftarrow$  next instance from  $D$ ;
5    $H \leftarrow \text{LGG}(H, x)$ ;           // e.g., LGG-Conj (Alg. 4.2) or LGG-Conj-ID (Alg. 4.3)
6 end
7 return  $H$ 
```

---



**Figure 4.2.** Part of the hypothesis space in Figure 4.1 containing only concepts that are more general than at least one of the three given instances on the bottom row. Only four conjunctions, indicated in green at the top, are more general than all three instances; the least general of these is  $Gills = no \wedge Beak = yes$ . It can be observed that the two left-most and right-most instances would be sufficient to learn that concept.

as this would clearly be an over-generalisation. Negative examples are very useful to prevent over-generalisation.

---

**Example 4.2 (Negative examples).** In Example 4.1 we observed the following dolphins:

p1: Length = 3  $\wedge$  Gills = no  $\wedge$  Beak = yes  $\wedge$  Teeth = many

p2: Length = 4  $\wedge$  Gills = no  $\wedge$  Beak = yes  $\wedge$  Teeth = many

p3: Length = 3  $\wedge$  Gills = no  $\wedge$  Beak = yes  $\wedge$  Teeth = few

Suppose you next observe an animal that clearly doesn't belong to the species – a negative example. It is described by the following conjunction:

n1: Length = 5  $\wedge$  Gills = yes  $\wedge$  Beak = yes  $\wedge$  Teeth = many

This negative example rules out some of the generalisations that were hitherto still possible: in particular, it rules out the concept Beak = yes, as well as the empty concept which postulates that everything is a dolphin.

---

The process is illustrated in Figure 4.3. We now have two hypotheses left, one which is least general and the other most general.

### Internal disjunction

You might be tempted to conclude from this and the previous example that we always have a unique most general hypothesis, but that is not the case in general. To demonstrate that, we are going to make our logical language slightly richer, by allowing a restricted form of disjunction called *internal disjunction*. The idea is very simple: if you observe one dolphin that is 3 metres long and another one of 4 metres, you may want to add the condition 'length is 3 or 4 metres' to your concept. We will write this as Length = [3,4], which logically means Length = 3  $\vee$  Length = 4. This of course only makes sense for features that have more than two values: for instance, the internal disjunction Teeth = [many, few] is always true and can be dropped.

---

**Algorithm 4.2: LGG-Conj( $x, y$ )** – find least general conjunctive generalisation of two conjunctions.

---

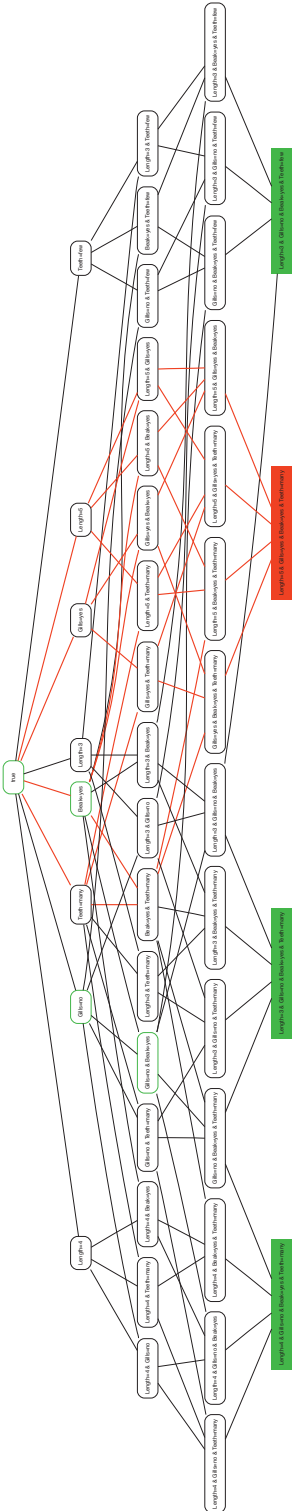
**Input** : conjunctions  $x, y$ .

**Output** : conjunction  $z$ .

1  $z \leftarrow$  conjunction of all literals common to  $x$  and  $y$ ;

2 **return**  $z$

---



**Figure 4.3.** A negative example can rule out some of the generalisations of the LGG of the positive examples. Every concept which is connected by a red path to a negative example covers that negative and is therefore ruled out as a hypothesis. Only two conjunctions cover all positives and no negatives:  $Gills = no \wedge Beak = yes$  and  $Gills = no$ .

**Example 4.3 (Internal disjunction).** Using the same three positive examples as in Example 4.1, the second and third hypothesis are now

$$\text{Length} = [3, 4] \wedge \text{Gills} = \text{no} \wedge \text{Beak} = \text{yes} \wedge \text{Teeth} = \text{many}$$

and

$$\text{Length} = [3, 4] \wedge \text{Gills} = \text{no} \wedge \text{Beak} = \text{yes}$$

We can drop any of the three conditions in the latter LGG without covering the negative example from Example 4.2. Generalising further to single conditions, we see that  $\text{Length} = [3, 4]$  and  $\text{Gills} = \text{no}$  are still OK but  $\text{Beak} = \text{yes}$  is not, as it covers the negative example.

Algorithm 4.3 details how we can calculate the LGG of two conjunctions employing internal disjunction. The function  $\text{Combine-ID}(v_x, v_y)$  returns  $[v_x, v_y]$  if  $v_x$  and  $v_y$  are constants, and their union if  $v_x$  or  $v_y$  are already sets of values: e.g.,  $\text{Combine-ID}([3, 4], [4, 5]) = [3, 4, 5]$ .

## 4.2 Paths through the hypothesis space

As we can clearly see in Figure 4.4, in this example we have not one but two most general hypotheses. What we can also notice is that *every concept between the least general one and one of the most general ones is also a possible hypothesis*, i.e., covers all the positives and none of the negatives. Mathematically speaking we say that the set of

---

**Algorithm 4.3:**  $\text{LGG-Conj-ID}(x, y)$  – find least general conjunctive generalisation of two conjunctions, employing internal disjunction.

---

**Input** : conjunctions  $x, y$ .

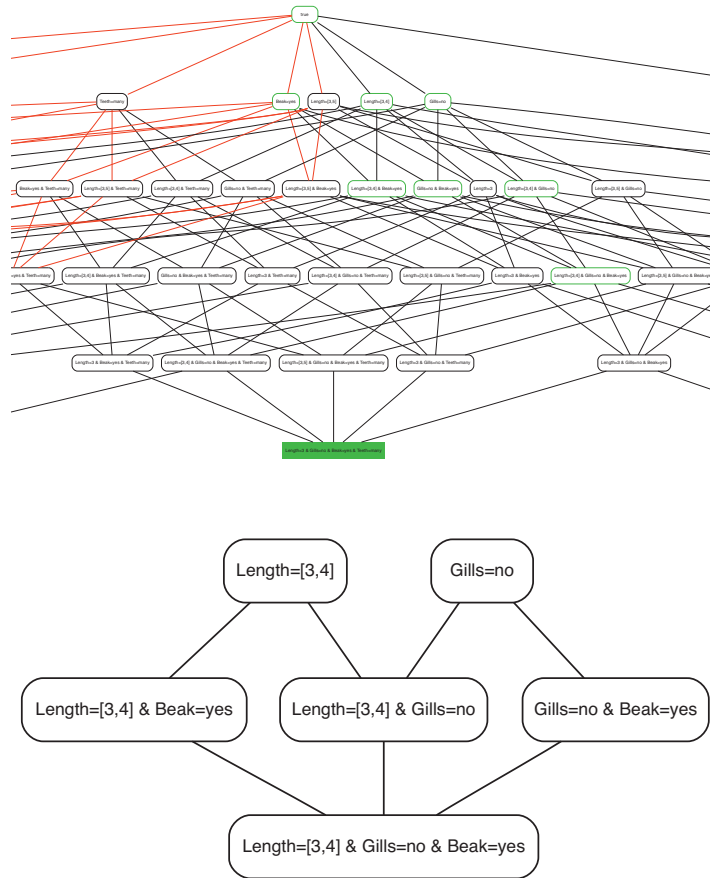
**Output** : conjunction  $z$ .

```

1  $z \leftarrow \text{true}$ ;
2 for each feature  $f$  do
3   if  $f = v_x$  is a conjunct in  $x$  and  $f = v_y$  is a conjunct in  $y$  then
4     | add  $f = \text{Combine-ID}(v_x, v_y)$  to  $z$ ;           //  $\text{Combine-ID}$ : see text
5   end
6 end
7 return  $z$ 
```

---



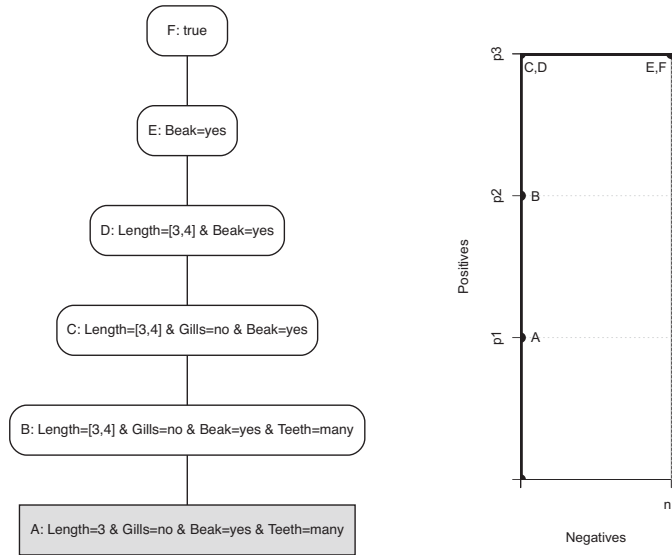


**Figure 4.4. (top)** A snapshot of the expanded hypothesis space that arises when internal disjunction is used for the ‘Length’ feature. We now need one more generalisation step to travel upwards from a completely specified example to the empty conjunction. **(bottom)** The version space consists of one least general hypothesis, two most general hypotheses, and three in between.

hypotheses that agree with the data is a *convex set*, which basically means that we can interpolate between any two members of the set, and if we find a concept that is less general than one and more general than the other then that concept is also a member of the set. This in turn means that we can describe the set of all possible hypotheses by its least and most general members. This is summed up in the following definition.

**Definition 4.1 (Version space).** A concept is *complete* if it covers all positive examples. A concept is *consistent* if it covers none of the negative examples. The *version space* is the set of all complete and consistent concepts. This set is convex and is fully defined by its least and most general elements.





**Figure 4.5. (left)** A path in the hypothesis space of Figure 4.3 from one of the positive examples (p1, see Example 4.2 on p.110) all the way up to the empty concept. Concept A covers a single example; B covers one additional example; C and D are in the version space, and so cover all three positives; E and F also cover the negative. **(right)** The corresponding coverage curve, with ranking  $p1 - p2 - p3 - n1$ .

We can draw a useful connection between logical hypothesis spaces and the coverage plots introduced in Chapter 2. Suppose you were to follow a path in the hypothesis space from a positive example, through a selection of its generalisations, all the way up to the empty concept. The latter, by construction, covers all positives and all negatives, and hence occupies the top-right point (*Neg, Pos*) in the coverage plot. The starting point, being a single positive example, occupies the point (0, 1) in the coverage plot. In fact, it is customary to extend the hypothesis space with a bottom element which doesn't cover any examples and hence is less general than any other concept. Taking that point as the starting point of the path means that we start in the bottom-left point (0, 0) in the coverage plot.

Moving upwards in the hypothesis space by generalisation means that the numbers of covered positives and negatives can stay the same or increase, but never decrease. In other words, *an upward path through the hypothesis space corresponds to a coverage curve* and hence to a ranking. Figure 4.5 illustrates this for the running example. The chosen path is but one among many possible paths; however, notice that if a path, like this one, includes elements of the version space, the corresponding coverage curve passes through 'ROC heaven' (0, *Pos*) and  $AUC = 1$ . In other words, such paths are optimal. Concept learning can be seen as the search for an optimal path through the

hypothesis space.

What happens, you may ask, if the LGG of the positive examples covers one or more negatives? In that case, any generalisation of the LGG will be inconsistent as well. Conversely, any consistent hypothesis will be incomplete. It follows that the version space is empty in this case; we will say that the data is not *conjunctively separable*. The following example illustrates this.

**Example 4.4 (Data that is not conjunctively separable).** Suppose we have the following five positive examples (the first three are the same as in Example 4.1):

p1: Length = 3  $\wedge$  Gills = no  $\wedge$  Beak = yes  $\wedge$  Teeth = many  
 p2: Length = 4  $\wedge$  Gills = no  $\wedge$  Beak = yes  $\wedge$  Teeth = many  
 p3: Length = 3  $\wedge$  Gills = no  $\wedge$  Beak = yes  $\wedge$  Teeth = few  
 p4: Length = 5  $\wedge$  Gills = no  $\wedge$  Beak = yes  $\wedge$  Teeth = many  
 p5: Length = 5  $\wedge$  Gills = no  $\wedge$  Beak = yes  $\wedge$  Teeth = few

and the following negatives (the first one is the same as in Example 4.2):

n1: Length = 5  $\wedge$  Gills = yes  $\wedge$  Beak = yes  $\wedge$  Teeth = many  
 n2: Length = 4  $\wedge$  Gills = yes  $\wedge$  Beak = yes  $\wedge$  Teeth = many  
 n3: Length = 5  $\wedge$  Gills = yes  $\wedge$  Beak = no  $\wedge$  Teeth = many  
 n4: Length = 4  $\wedge$  Gills = yes  $\wedge$  Beak = no  $\wedge$  Teeth = many  
 n5: Length = 4  $\wedge$  Gills = no  $\wedge$  Beak = yes  $\wedge$  Teeth = few

The least general complete hypothesis is Gills = no  $\wedge$  Beak = yes as before, but this covers n5 and hence is inconsistent. There are seven most general consistent hypotheses, none of which are complete:

Length = 3 (covers p1 and p3)  
 Length = [3,5]  $\wedge$  Gills = no (covers all positives except p2)  
 Length = [3,5]  $\wedge$  Teeth = few (covers p3 and p5)  
 Gills = no  $\wedge$  Teeth = many (covers p1, p2 and p4)  
 Gills = no  $\wedge$  Beak = no  
 Gills = yes  $\wedge$  Teeth = few  
 Beak = no  $\wedge$  Teeth = few

The last three of these do not cover any positive examples.

### Most general consistent hypotheses

As this example suggests, finding most general consistent hypotheses is considerably more involved than finding least general complete ones. Essentially, the process is one of enumeration. [Algorithm 4.4](#) gives an algorithm which returns all most general consistent specialisations of a given concept, where a minimal specialisation of a concept is one that can be reached in one downward step in the hypothesis lattice (e.g., by adding a conjunct, or removing a value from an internal disjunction). Calling the algorithm with  $C = \text{true}$  returns the most general consistent hypotheses.

[Figure 4.6](#) shows a path through the hypothesis space of [Example 4.4](#), and the corresponding coverage curve. We see that the path goes through three consistent hypotheses, which are consequently plotted on the  $y$ -axis of the coverage plot. The other three hypotheses are complete, and therefore end up on the top of the graph; one of these is, in fact, the LGG of the positives (D). The ranking corresponding to this coverage curve is  $p3 - p5 - [p1, p4] - [p2, n5] - [n1-4]$ . This ranking commits half a ranking error out of 25, and so  $\text{AUC} = 0.98$ . We can choose one concept from the ranking by applying the techniques discussed in [Section 2.2](#). For instance, suppose that classification accuracy is the criterion we want to optimise. In coverage space, accuracy isometrics have slope 1, and so we see immediately that concepts C and D (or E) both achieve the best accuracy in [Figure 4.6](#). If performance on the positives is more important we prefer the complete but inconsistent concept D; if performance on the negatives is valued more we choose the incomplete but consistent concept C.

### Closed concepts

It is worthwhile to reflect on the fact that concepts D and E occupy the same point in coverage space. What this means is that generalising D into E by dropping  $\text{Beak} = \text{yes}$  does not change the coverage in terms of positive and negative examples. One could

---

**Algorithm 4.4:**  $\text{MGConsistent}(C, N)$  – find most general consistent specialisations of a concept.

---

**Input** : concept  $C$ ; negative examples  $N$ .

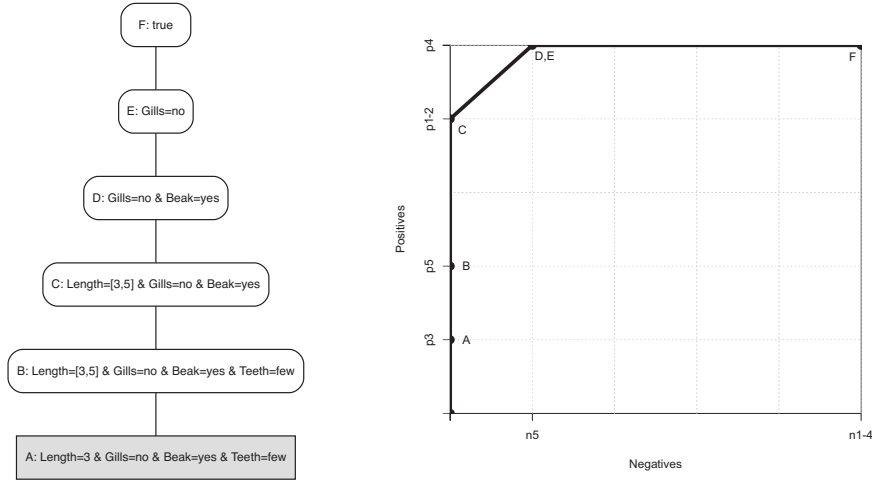
**Output** : set of concepts  $S$ .

```

1 if  $C$  doesn't cover any element from  $N$  then return  $\{C\}$ ;
2  $S \leftarrow \emptyset$ ;
3 for each minimal specialisation  $C'$  of  $C$  do
4   |  $S \leftarrow S \cup \text{MGConsistent}(C', N)$ ;
5 end
6 return  $S$ 

```

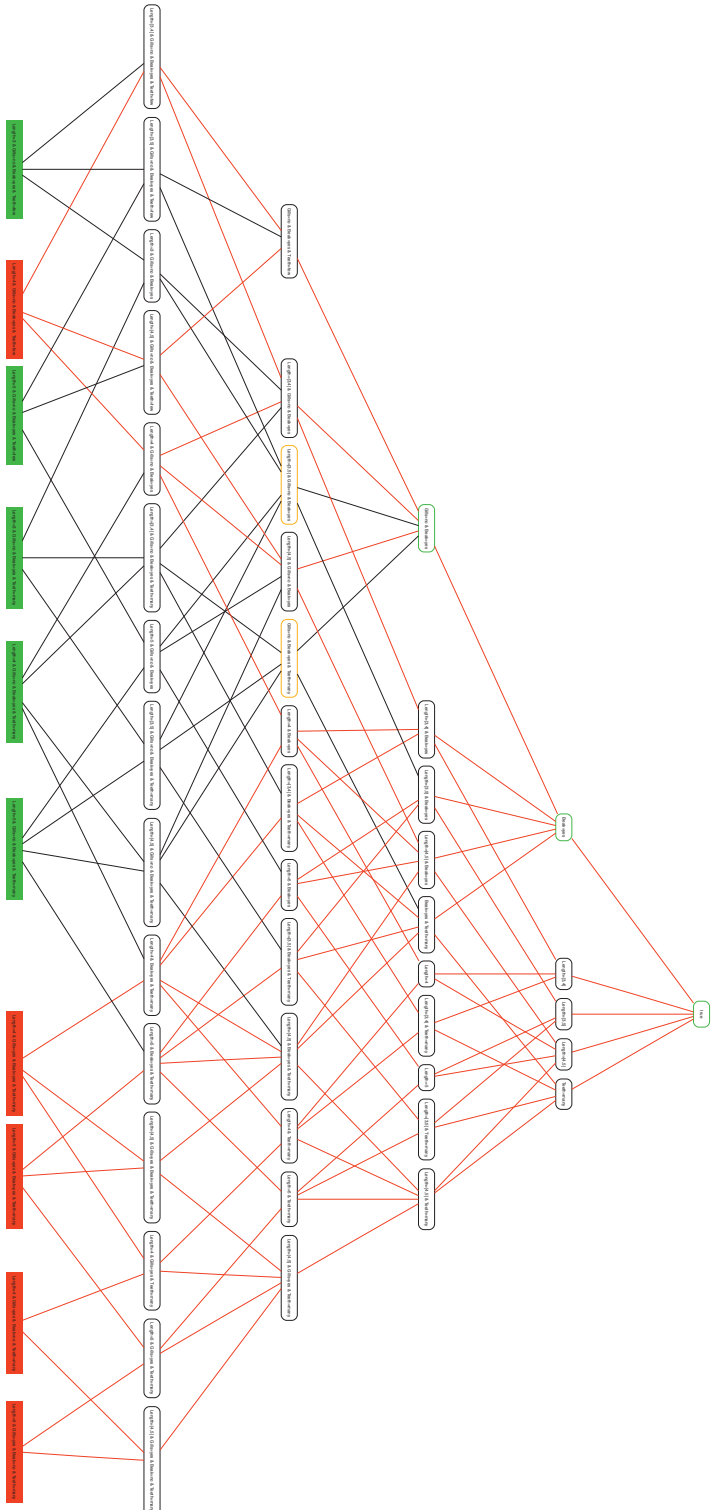
---



**Figure 4.6.** (left) A path in the hypothesis space of Example 4.4. Concept A covers a single positive (p3); B covers one additional positive (p5); C covers all positives except p4; D is the LGG of all five positive examples, but also covers a negative (n5), as does E. (right) The corresponding coverage curve.

say that the data suggests that, in the context of concept E, the condition **Beak = yes** is implicitly understood. A concept that includes all implicitly understood conditions is called a *closed concept*. Essentially, a closed concept is the LGG of all examples that it covers. For instance, D and E both cover all positives and n5; the LGG of those six examples is **Gills = no  $\wedge$  Beak = yes**, which is D. Mathematically speaking we say that the closure of E is D, which is also its own closure – hence the term ‘closed concept’. This doesn’t mean that D and E are logically equivalent: on the contrary, since  $\mathcal{X}_D \subset \mathcal{X}_E$  – the extension of D is a proper subset of the extension of E – there exist instances in  $\mathcal{X}$  that are covered by E but not by D. However, none of these ‘witnesses’ are present in the data, and thus, as far as the data is concerned, D and E are indistinguishable. As can be seen in Figure 4.7, limiting attention to closed concepts can considerably reduce the hypothesis space.

In this section we have looked at the problem of learning a single logical expression that covers most or all positive examples and few or no negative examples. We have seen that such concepts live in a hypothesis space ordered by generality, and learning a concept can be understood as finding a good path through that hypothesis space. Such a path has a natural interpretation as a ranker, which allows a connection with coverage curves and ROC curves. On the other hand, insisting on a single conjunction of feature-value literals is a strong limitation; in the next section we look at ways to relax it.



**Figure 4.7.** The hypothesis space is reduced considerably if we restrict attention to closed concepts. There are three, rather than four, complete concepts (in green), and two, rather than seven, most general consistent closed concepts (in orange). Notice that the latter are both specialisations of the LGG of the positives, and hence it is possible to select a path that includes both the LGG and a most general consistent hypothesis.

## 4.3 Beyond conjunctive concepts

Recall from [Background 4.1](#) that a conjunctive normal form expression (CNF) is a conjunction of disjunctions of literals, or equivalently, a conjunction of clauses. The conjunctions of literals we have looked at until now are trivially in CNF where each disjunction consists of a single literal. CNF expressions are much more expressive, particularly since literals can occur in several clauses. We will look at an algorithm for learning Horn theories, where each clause  $A \rightarrow B$  is a Horn clause, i.e.,  $A$  is a conjunction of literals and  $B$  is a single literal. For ease of notation we will restrict attention to Boolean features, and write  $F$  for  $F = \text{true}$  and  $\neg F$  for  $F = \text{false}$ . In the example below we adapt the dolphins example to Boolean variables **ManyTeeth** (standing for **Teeth = many**), **Gills, Short** (standing for **Length = 3**) and **Beak**.

When we looked at learning conjunctive concepts, the main intuition was that uncovered positive examples led us to generalise by dropping literals from the conjunction, while covered negative examples require specialisation by adding literals. This intuition still holds if we are learning Horn theories, but now we need to think ‘clauses’ rather than ‘literals’. Thus, if a Horn theory doesn’t cover a positive we need to drop all clauses that violate the positive, where a clause  $A \rightarrow B$  violates a positive if all literals in the conjunction  $A$  are true in the example, and  $B$  is false.

Things get more interesting if we consider covered negatives, since then we need to find one or more clauses to add to the theory in order to exclude the negative. For example, suppose that our current hypothesis covers the negative

$$\text{ManyTeeth} \wedge \text{Gills} \wedge \text{Short} \wedge \neg \text{Beak}$$

To exclude it, we can add the following Horn clause to our theory:

$$\text{ManyTeeth} \wedge \text{Gills} \wedge \text{Short} \rightarrow \text{Beak}$$

While there are other clauses that can exclude the negative (e.g.,  $\text{ManyTeeth} \rightarrow \text{Beak}$ ) this is the most specific one, and hence least at risk of also excluding covered positives. However, the most specific clause excluding a negative is only unique if the negative has exactly one literal set to **false**. For example, if our covered negative is

$$\text{ManyTeeth} \wedge \text{Gills} \wedge \neg \text{Short} \wedge \neg \text{Beak}$$

then we have a choice between the following two Horn clauses:

$$\text{ManyTeeth} \wedge \text{Gills} \rightarrow \text{Short}$$

$$\text{ManyTeeth} \wedge \text{Gills} \rightarrow \text{Beak}$$

Notice that, the fewer literals are set to **true** in the negative example, the more general the clauses excluding the negative are.

The approach of [Algorithm 4.5](#) is to add *all* of these clauses to the hypothesis. However, the algorithm applies two clever tricks. The first is that it maintains a list  $S$  of negative examples, from which it periodically rebuilds the hypothesis. The second is that, rather than simply adding new negative examples to the list, it tries to find negatives with fewer literals set to **true**, since this will result in more general clauses. This is possible if we assume we have access to a *membership oracle*  $Mb$  which can tell us whether a particular example is a member of the concept we're learning or not. So in line 7 of the algorithm we form the *intersection* of a new negative  $x$  and an existing one  $s \in S$  – i.e., an example with only those literals set to **true** which are **true** in both  $x$  and  $s$  – and pass the result  $z$  to the membership oracle to check whether it belongs to the target concept. The algorithm also assumes access to an *equivalence oracle*  $Eq$  which either tells us that our current hypothesis  $h$  is logically equivalent to the target formula  $f$ , or else produces a *counter-example* that can be either a false positive (it is covered by  $h$  but not by  $f$ ) or a false negative (it is covered by  $f$  but not by  $h$ ).

---

**Algorithm 4.5:**  $\text{Horn}(Mb, Eq)$  – learn a conjunction of Horn clauses from membership and equivalence oracles.

---

```

Input   : equivalence oracle  $Eq$ ; membership oracle  $Mb$ .
Output  : Horn theory  $h$  equivalent to target formula  $f$ .

1  $h \leftarrow \text{true};$                                 // conjunction of Horn clauses, initially empty
2  $S \leftarrow \emptyset;$                              // a list of negative examples, initially empty
3 while  $Eq(h)$  returns counter-example  $x$  do
4   if  $x$  violates at least one clause of  $h$  then           //  $x$  is a false negative
5     | specialise  $h$  by removing every clause that  $x$  violates
6   else                                           //  $x$  is a false positive
7     | find the first negative example  $s \in S$  such that (i)  $z = s \cap x$  has fewer true
      | literals than  $s$ , and (ii)  $Mb(z)$  labels it as a negative;
8     | if such an example exists then replace  $s$  in  $S$  with  $z$ , else append  $x$  to the
      | end of  $S$ ;
9     |  $h \leftarrow \text{true};$ 
10    for all  $s \in S$  do                                // rebuild  $h$  from  $S$ 
11      |  $p \leftarrow$  the conjunction of literals true in  $s$ ;
12      |  $Q \leftarrow$  the set of literals false in  $s$ ;
13      | for all  $q \in Q$  do  $h \leftarrow h \wedge (p \rightarrow q);$ 
14    end
15  end
16 end
17 return  $h$ 

```

---



**Example 4.5 (Learning a Horn theory).** Suppose the target theory  $f$  is

$$(\text{ManyTeeth} \wedge \text{Short} \rightarrow \text{Beak}) \wedge (\text{ManyTeeth} \wedge \text{Gills} \rightarrow \text{Short})$$

This theory has 12 positive examples: eight in which **ManyTeeth** is **false**; another two in which **ManyTeeth** is **true** but both **Gills** and **Short** are **false**; and two more in which **ManyTeeth**, **Short** and **Beak** are **true**. The negative examples, then, are

- n1:  $\text{ManyTeeth} \wedge \text{Gills} \wedge \text{Short} \wedge \neg \text{Beak}$
- n2:  $\text{ManyTeeth} \wedge \text{Gills} \wedge \neg \text{Short} \wedge \text{Beak}$
- n3:  $\text{ManyTeeth} \wedge \text{Gills} \wedge \neg \text{Short} \wedge \neg \text{Beak}$
- n4:  $\text{ManyTeeth} \wedge \neg \text{Gills} \wedge \text{Short} \wedge \neg \text{Beak}$

$S$  is initialised to the empty list and  $h$  to the empty conjunction. We call the equivalence oracle which returns a counter-example which has to be a false positive (since every example satisfies our initial hypothesis), say n1 which violates the first clause in  $f$ . There are no negative examples in  $S$  yet, so we add n1 to  $S$  (step 8 of Algorithm 4.5). We then generate a new hypothesis from  $S$  (steps 9–13):  $p$  is  $\text{ManyTeeth} \wedge \text{Gills} \wedge \text{Short}$  and  $Q$  is  $\{\text{Beak}\}$ , so  $h$  becomes  $(\text{ManyTeeth} \wedge \text{Gills} \wedge \text{Short} \rightarrow \text{Beak})$ . Notice that this clause is implied by our target theory: if **ManyTeeth** and **Gills** are **true** then so is **Short** by the second clause of  $f$ ; but then so is **Beak** by  $f$ 's first clause. But we need more clauses to exclude all the negatives.

Now, suppose the next counter-example is the false positive n2. We form the intersection with n1 which was already in  $S$  to see if we can get a negative example with fewer literals set to **true** (step 7). The result is equal to n3 so the membership oracle will confirm this as a negative, and we replace n1 in  $S$  with n3. We then rebuild  $h$  from  $S$  which gives ( $p$  is  $\text{ManyTeeth} \wedge \text{Gills}$  and  $Q$  is  $\{\text{Short}, \text{Beak}\}$ )

$$(\text{ManyTeeth} \wedge \text{Gills} \rightarrow \text{Short}) \wedge (\text{ManyTeeth} \wedge \text{Gills} \rightarrow \text{Beak})$$

Finally, assume that n4 is the next false positive returned by the equivalence oracle. The intersection with n3 on  $S$  is actually a positive example, so instead of intersecting with n3 we append n4 to  $S$  and rebuild  $h$ . This gives the previous two clauses from n3 plus the following two from n4:

$$(\text{ManyTeeth} \wedge \text{Short} \rightarrow \text{Gills}) \wedge (\text{ManyTeeth} \wedge \text{Short} \rightarrow \text{Beak})$$

The first of this second pair will subsequently be removed by a false negative from

the equivalence oracle, leading to the final theory

$$\begin{aligned} &(\text{ManyTeeth} \wedge \text{Gills} \rightarrow \text{Short}) \wedge \\ &(\text{ManyTeeth} \wedge \text{Gills} \rightarrow \text{Beak}) \wedge \\ &(\text{ManyTeeth} \wedge \text{Short} \rightarrow \text{Beak}) \end{aligned}$$

which is logically equivalent (though not identical) to  $f$ .

---

The Horn algorithm combines a number of interesting new ideas. First, it is an *active learning* algorithm: rather than learning from a provided data set, it constructs its own training examples and asks the membership oracle to label them. Secondly, the core of the algorithm is the list of cleverly chosen negative examples, from which the hypothesis is periodically rebuilt. The intersection step is crucial here: if the algorithm just remembered negatives, the hypothesis would consist of many specific clauses. It can be shown that, in order to learn a theory consisting of  $m$  clauses and  $n$  Boolean variables, the algorithm requires  $O(mn)$  equivalence queries and  $O(m^2n)$  membership queries. In addition, the runtime of the algorithm is quadratic in both  $m$  and  $n$ . While this is probably prohibitive in practice, the Horn algorithm can be shown to always successfully learn a Horn theory that is equivalent to the target theory. Furthermore, if we don't have access to an equivalence oracle the algorithm is still guaranteed to 'almost always' learn a Horn theory that is 'mostly correct'. This will be made more precise in [Section 4.4](#).

### Using first-order logic

Another way to move beyond conjunctive concepts defined by simple features is to use a richer logical language. The languages we have been using so far are *propositional*: each literal is a proposition such as  $\text{Gills} = \text{yes}$  – standing for 'the dolphin has gills' – from which larger expressions are built using logical connectives. *First-order predicate logic*, or first-order logic for short, generalises this by building more complex literals from *predicates* and *terms*. For example, a first-order literal could be  $\text{BodyPart}(\text{Dolphin42}, \text{PairOf}(\text{Gill}))$ . Here,  $\text{Dolphin42}$  and  $\text{PairOf}(\text{Gill})$  are terms referring to objects:  $\text{Dolphin42}$  is a constant, and  $\text{PairOf}(\text{Gill})$  is a compound term consisting of the function symbol  $\text{PairOf}$  and the term  $\text{Gills}$ .  $\text{BodyPart}$  is a binary predicate forming a proposition (something that can be true or false) out of two terms. This richer language brings with it a number of advantages:

- ☞ we can use terms such as  $\text{Dolphin42}$  to refer to individual objects we're interested in;

- ☞ the structure of objects can be explicitly described; and
- ☞ we can introduce variables to refer to unspecified objects and quantify over them.

To illustrate the latter point, the first-order literal `BodyPart(x, PairOf(Gill))` can be used to refer to the set of all objects having a pair of gills; and the following expression applies universal quantification to state that everything with a pair of gills is a fish:

$$\forall x: \text{BodyPart}(x, \text{PairOf}(\text{Gill})) \rightarrow \text{Fish}(x)$$

Since we modified the structure of literals, we need to revisit notions such as generalisation and LGG. Remember that for propositional literals with internal disjunction we used the function `Combine-ID` for merging two internal disjunctions: thus, for example, `LGG-Conj-ID(Length = [3, 4], Length = [4, 5])` returns `Length = [3, 4, 5]`. In order to generalise first-order literals we use variables. Consider, for example, the two first-order literals `BodyPart(Dolphin42, PairOf(Gill))` and `BodyPart(Human123, PairOf(Leg))`: these generalise to `BodyPart(x, PairOf(y))`, signifying the set of objects that have a pair of some unspecified body part. There is a well-defined algorithm for computing LGGs of first-order literals called *anti-unification*, as it is the mathematical dual of the deductive operation of *unification*.

---

**Example 4.6 (Unification and anti-unification).** Consider the following terms:

<code>BodyPart(x, PairOf(Gill))</code>	describing the objects that have a pair of gills;
<code>BodyPart(Dolphin42, PairOf(y))</code>	describing the body parts that <code>Dolphin42</code> has a pair of.

The following two terms are their unification and anti-unification, respectively:

<code>BodyPart(Dolphin42, PairOf(Gill))</code>	describing <code>Dolphin42</code> as having a pair of gills;
<code>BodyPart(x, PairOf(y))</code>	describing the objects that have a pair of unspecified body parts.

---

So we see that in first-order logic literals already have quite a rich structure, owing to the use of variables. We will revisit this in [Section 6.4](#) when we discuss how to learn classification rules in first-order logic.

## 4.4 Learnability

In this chapter we have seen several hypothesis languages for concept learning, including conjunctions of literals (possibly with internal disjunction), conjunctions of Horn clauses, and clauses in first-order logic. It is intuitively clear that these languages differ in expressivity: for example, a conjunction of literals is also a conjunction of Horn clauses with empty if-part, so Horn theories are strictly more expressive than conjunctive concepts. The downside of a more expressive concept language is that it may be harder to learn. The field of computational learning theory studies exactly this question of *learnability*.

To kick things off we need a *learning model*: a clear statement of what we mean if we say that a concept language is learnable. One of the most common learning models is the model of *probably approximately correct* (PAC) learning. PAC-learnability means that there exists a learning algorithm that gets it mostly right, most of the time. The model makes an allowance for mistakes on non-typical examples: hence the ‘mostly right’ or ‘approximately correct’. The model also makes an allowance for sometimes getting it completely wrong, for example when the training data contains lots of non-typical examples: hence the ‘most of the time’ or ‘probably’. We assume that typicality of examples is determined by some unspecified probability distribution  $D$ , and we evaluate the error rate  $err_D$  of a hypothesis with respect to this distribution  $D$ . More formally, for arbitrary allowable error rate  $\epsilon < 1/2$  and failure rate  $\delta < 1/2$  we require a PAC-learning algorithm to output with probability at least  $1 - \delta$  a hypothesis  $h$  such that  $err_D < \epsilon$ .

Let’s assume for the moment that our data is noise-free, and that the target hypothesis is chosen from our hypothesis language. Furthermore, we assume our learner always outputs a hypothesis that is complete and consistent with the training sample. There is a possibility that this zero training error is misleading, and that the hypothesis is actually a ‘bad’ one, having a true error over the instance space that is larger than  $\epsilon$ . We just want to make sure that this happens with probability less than  $\delta$ . I will now show that this can be guaranteed by choosing the training sample large enough. Suppose our hypothesis space  $H$  contains a single bad hypothesis, then the probability it is complete and consistent on  $m$  independently sampled training examples is at most  $(1 - \epsilon)^m$ . Since  $1 - \epsilon \leq e^{-\epsilon}$  for any  $0 \leq \epsilon \leq 1$ , we have that this probability is at most  $e^{-m\epsilon}$ . We want this to be at most  $\delta$ , which can be achieved by setting  $m \geq \frac{1}{\epsilon} \ln \frac{1}{\delta}$ . Now,  $H$  may contain several bad hypotheses, say  $k \leq |H|$ ; then the probability that at least one of them is complete and consistent on  $m$  independently sampled training examples is at most  $k(1 - \epsilon)^m \leq |H|(1 - \epsilon)^m \leq |H|e^{-m\epsilon}$ , which is at most  $\delta$  if

$$m \geq \frac{1}{\epsilon} \left( \ln |H| + \ln \frac{1}{\delta} \right) \quad (4.1)$$

This is called the *sample complexity* of a complete and consistent learner. The good

news is that it is linear in  $1/\epsilon$  and logarithmic in  $1/\delta$ . Notice that this suggests that it is exponentially cheaper to reduce the failure rate than it is to reduce the error. Any learning algorithm that takes time polynomial in  $1/\epsilon$  and  $1/\delta$  to process a single training example will therefore also take polynomial training time, another requirement for PAC-learnability. However, finding a complete and consistent hypothesis is not tractable in many hypothesis languages.

Notice that the term  $\ln|H|$  arose because in the worst case almost all hypotheses in  $H$  are bad. However, in practice this means that the bound in Equation 4.1 is overly pessimistic. Still, it allows us to see that concept languages whose size is exponential in some parameter  $n$  are PAC-learnable. For example, the number of conjunctions over  $n$  Boolean variables is  $3^n$ , since each variable can occur unnegated, negated or not at all. Consequently, the sample complexity is  $(1/\epsilon)(n \ln 3 + \ln(1/\delta))$ . For example, if we set  $\delta = 0.05$  and  $\epsilon = 0.1$  then the sample complexity is approximately  $10(n \cdot 1.1 + 3) = 11n + 30$ . For our dolphin example with  $n = 4$  this is clearly pessimistic, since there are only  $2^4 = 16$  distinct examples! For larger  $n$  this is more realistic. Notice also that the PAC model is distribution-free: the learner is not given any information about the instance distribution  $D$ . This is another source for pessimism in the bound on the sample complexity.

We may not always be able to output a complete and consistent hypothesis: for instance, this may be computationally intractable, the target hypothesis may not be representable in our hypothesis language, or the examples may be noisy. A reasonable strategy would be to choose the hypothesis with lowest training error. A ‘bad’ hypothesis is then one whose true error exceeds the training error by at least  $\epsilon$ . Using some results from probability theory, we find that this probability is at most  $e^{-2m\epsilon^2}$ . As a result, the  $1/\epsilon$  factor in Equation 4.1 is replaced by  $1/2\epsilon^2$ : for  $\epsilon = 0.1$  we thus need 5 times as many training examples compared to the previous case.

It has already been mentioned that the  $|H|$  term is a weak point in the above analysis. What we need is a measure that doesn’t just count the size of the hypothesis space, but rather gives its expressivity or capacity in terms of classification. Such a measure does in fact exist and is called the *VC-dimension* after its inventors Vladimir Vapnik and Alexey Chervonenkis. We will illustrate the main idea by means of an example.

---

**Example 4.7 (Shattering a set of instances).** Consider the following instances:

$m = \text{ManyTeeth} \wedge \neg \text{Gills} \wedge \neg \text{Short} \wedge \neg \text{Beak}$

$g = \neg \text{ManyTeeth} \wedge \text{Gills} \wedge \neg \text{Short} \wedge \neg \text{Beak}$

$s = \neg \text{ManyTeeth} \wedge \neg \text{Gills} \wedge \text{Short} \wedge \neg \text{Beak}$

$$b = \neg \text{ManyTeeth} \wedge \neg \text{Gills} \wedge \neg \text{Short} \wedge \text{Beak}$$

There are 16 different subsets of the set  $\{m, g, s, b\}$ . Can each of them be represented by its own conjunctive concept? The answer is yes: for every instance we want to exclude, we add the corresponding negated literal to the conjunction. Thus,  $\{m, s\}$  is represented by  $\neg \text{Gills} \wedge \neg \text{Beak}$ ,  $\{g, s, b\}$  is represented by  $\neg \text{ManyTeeth}$ ,  $\{s\}$  is represented by  $\neg \text{ManyTeeth} \wedge \neg \text{Gills} \wedge \neg \text{Beak}$ , and so on. We say that this set of four instances is *shattered* by the hypothesis language of conjunctive concepts.

The VC-dimension is the size of the largest set of instances that can be shattered by a particular hypothesis language or model class. The previous example shows that the VC-dimension of conjunctive concepts over  $d$  Boolean literals is at least  $d$ . It is in fact equal to  $d$ , although this is harder to prove (since it involves showing that no set of  $d + 1$  instances can be shattered). This measures the capacity of the model class for representing concepts or binary classifiers. As another example, the VC-dimension of a linear classifier in  $d$  dimensions is  $d + 1$ : a threshold on the real line can shatter two points but not three (since the middle point cannot be separated from the other two by a single threshold); a straight line in a two-dimensional space can shatter three points but not four; and so on.

The VC-dimension can be used to bound the difference between sample error and true error of a hypothesis (which is the step where  $|H|$  appeared in our previous arguments). Consequently, it can also be used to derive a bound on the sample complexity of a complete and consistent learner in terms of the VC-dimension  $D$  rather than  $|H|$ :

$$m \geq \frac{1}{\epsilon} \max \left( 8D \log_2 \frac{13}{\epsilon}, 4 \log_2 \frac{2}{\delta} \right) \quad (4.2)$$

We see that the bound is linear in  $D$ , where previously it was logarithmic in  $|H|$ . This is natural, since to shatter  $D$  points we need at least  $2^D$  hypotheses, and so  $\log_2 |H| \geq D$ . Furthermore, it is still logarithmic in  $1/\delta$ , but linear times logarithmic in  $1/\epsilon$ . Plugging in our previous values of  $\delta = 0.05$  and  $\epsilon = 0.1$ , we obtain a sample complexity of  $\max(562 \cdot D, 213)$ .

We conclude that the VC-dimension allows us to derive the sample complexity of infinite concept classes, as long as they have finite VC-dimension. It is furthermore worth mentioning a classical result from computational learning theory which says that a concept class is PAC-learnable if and only if its VC-dimension is finite.

## 4.5 Concept learning: Summary and further reading


In this chapter we looked at methods for inductive concept learning: the process of constructing a logical expression defining a set of objects from examples. This problem was a focus of early work in artificial intelligence (Winston, 1970; Vere, 1975; Banerji, 1980), following the seminal work by psychologists Bruner, Goodnow and Austin (1956) and Hunt, Marin and Stone (1966).

☞ In Section 4.1 we considered the structure of the hypothesis space: the set of possible concepts. Every hypothesis has an extension (the set of instances it covers), and thus relationships between extensions such as subset relationships carry over to the hypothesis space. This gives the hypothesis space a lattice structure: a partial order with least upper bounds and greatest lower bounds. In particular, the LGG is the least upper bound of a set of instances, and is the most conservative generalisation that we can learn from the data. The concept was defined in the context of first-order logic by Plotkin (1971), who showed that it was the mathematical dual of the deductive operation of unification. We can extend the hypothesis language with internal disjunction among values of a feature, which creates a larger hypothesis space that still has a lattice structure. Internal disjunction is a common staple of attribute-value languages for learning following the work of Michalski (1973). For further pointers regarding hypothesis language and hypothesis space the reader is referred to (Blockeel, 2010a,b).

☞ Section 4.2 defined complete and consistent hypotheses as concepts that cover all positive examples and no negative examples. The set of complete and consistent concepts is called the version space, a notion introduced by Mitchell (1977). The version space can be summarised by its least general and most general members, since any concept between one least general hypothesis and another most general one is also complete and consistent. Alternatively, we can describe the version space by all paths from a least general to a most general hypothesis. Such upward paths give rise to a coverage curve which describes the extension of each concept on the path in terms of covered positives and negatives. Concept learning can then be seen as finding an upward path that goes through ROC heaven. Syntactically different concepts can have the same extension in a particular data set: a closed concept is the most specific one of these (technically, the LGG of the instances in its extension). The notion is studied in formal concept analysis (Ganter and Wille, 1999) and was introduced in a data mining context by Pasquier, Bastide, Taouil and Lakhal (1999); Garriga, Kralj and Lavrač (2008) investigate its usefulness for labelled data.

☞ In Section 4.3 we discussed the Horn algorithm for learning concepts described

by conjunctions of Horn rules, first published in [Angluin \*et al.\* \(1992\)](#). The algorithm makes use of a membership oracle, which can be seen as an early form of active learning ([Cohn, 2010](#); [Dasgupta, 2010](#)). Horn theories are superficially similar to classification rule models which will be studied in [Chapter 6](#). However, there is an important difference, since those classification rules have the target variable in the then-part of the rule, while the Horn clauses we are looking at here can have any literal in the then-part. In fact, in this chapter the target variable is not part of the logical language at all. This setting is sometimes called *learning from interpretations*, since examples are truth-value assignments to our theory. The classification rule setting is called *learning from entailment*, since in order to find out whether a particular rule covers an example we need to apply logical inference. [De Raedt \(1997\)](#) explains and explores the differences between these two settings. Further introductions to first-order logic and its use in learning are given by [Flach \(2010a\)](#) and [De Raedt \(2010\)](#).

 [Section 4.4](#) briefly reviewed some basic concepts and results in learnability theory. My account partly followed [Mitchell \(1997, Chapter 7\)](#); another excellent introduction is given by [Zeugmann \(2010\)](#). PAC-learnability, which allows an error rate of  $\epsilon$  and a failure rate of  $\delta$ , was introduced in a seminal paper by [Valiant \(1984\)](#). [Haussler \(1988\)](#) derived the sample complexity for complete and consistent learners ([Equation 4.1](#)), which is linear in  $1/\epsilon$  and logarithmic in  $1/\delta$  and the size of the hypothesis space. The VC-dimension as a measure of the capacity of a hypothesis language was introduced by [Vapnik and Chervonenkis \(1971\)](#) in order to quantify the difference between training error and true error. This allows a statement of the sample complexity in terms of the VC-dimension ([Equation 4.2](#)) which is due to [Blumer, Ehrenfeucht, Haussler and Warmuth \(1989\)](#). This same paper proved that a model class is PAC-learnable if and only if its VC-dimension is finite.

