

---

## The ingredients of machine learning

---

**M**ACHINE LEARNING IS ALL ABOUT using the right features to build the right models that achieve the right tasks – this is the slogan, visualised in Figure 3 on p.11, with which we ended the Prologue. In essence, *features* define a ‘language’ in which we describe the relevant objects in our domain, be they e-mails or complex organic molecules. We should not normally have to go back to the domain objects themselves once we have a suitable feature representation, which is why features play such an important role in machine learning. We will take a closer look at them in Section 1.3. A *task* is an abstract representation of a problem we want to solve regarding those domain objects: the most common form of these is classifying them into two or more classes, but we shall encounter other tasks throughout the book. Many of these tasks can be represented as a mapping from data points to outputs. This mapping or *model* is itself produced as the output of a machine learning algorithm applied to training data; there is a wide variety of models to choose from, as we shall see in Section 1.2.

We start this chapter by discussing tasks, the problems that can be solved with machine learning. No matter what variety of machine learning models you may encounter, you will find that they are designed to solve one of only a small number of tasks and use only a few different types of features. One could say that *models lend the machine learning field diversity, but tasks and features give it unity.*

## 1.1 Tasks: the problems that can be solved with machine learning

Spam e-mail recognition was described in the [Prologue](#). It constitutes a binary classification task, which is easily the most common task in machine learning which figures heavily throughout the book. One obvious variation is to consider classification problems with more than two classes. For instance, we may want to distinguish different kinds of ham e-mails, e.g., work-related e-mails and private messages. We could approach this as a combination of two binary classification tasks: the first task is to distinguish between spam and ham, and the second task is, among ham e-mails, to distinguish between work-related and private ones. However, some potentially useful information may get lost this way, as some spam e-mails tend to look like private rather than work-related messages. For this reason, it is often beneficial to view *multi-class classification* as a machine learning task in its own right. This may not seem a big deal: after all, we still need to learn a model to connect the class to the features. However, in this more general setting some concepts will need a bit of rethinking: for instance, the notion of a decision boundary is less obvious when there are more than two classes.

Sometimes it is more natural to abandon the notion of discrete classes altogether and instead predict a real number. Perhaps it might be useful to have an assessment of an incoming e-mail's urgency on a sliding scale. This task is called *regression*, and essentially involves learning a real-valued function from training examples labelled with true function values. For example, I might construct such a training set by randomly selecting a number of e-mails from my inbox and labelling them with an urgency score on a scale of 0 (ignore) to 10 (immediate action required). This typically works by choosing a class of functions (e.g., functions in which the function value depends linearly on some numerical features) and constructing a function which minimises the difference between the predicted and true function values. Notice that this is subtly different from SpamAssassin learning a real-valued spam score, where the training data are labelled with classes rather than 'true' spam scores. This means that SpamAssassin has less information to go on, but it also allows us to interpret SpamAssassin's score as an assessment of how far it thinks an e-mail is removed from the decision boundary, and therefore as a measure of confidence in its own prediction. In a regression task the notion of a decision boundary has no meaning, and so we have to find other ways to express a model's confidence in its real-valued predictions.

Both classification and regression assume the availability of a training set of examples labelled with true classes or function values. Providing the true labels for a data set is often labour-intensive and expensive. Can we learn to distinguish spam from ham, or work e-mails from private messages, without a labelled training set? The answer is: yes, up to a point. The task of grouping data without prior information on the groups is called *clustering*. Learning from unlabelled data is called *unsupervised learning* and is quite distinct from *supervised learning*, which requires labelled training data. A typical

clustering algorithm works by assessing the similarity between instances (the things we're trying to cluster, e.g., e-mails) and putting similar instances in the same cluster and 'dissimilar' instances in different clusters.

---

**Example 1.1 (Measuring similarity).** If our e-mails are described by word-occurrence features as in the text classification example, the similarity of e-mails would be measured in terms of the words they have in common. For instance, we could take the number of common words in two e-mails and divide it by the number of words occurring in either e-mail (this measure is called the *Jaccard coefficient*). Suppose that one e-mail contains 42 (different) words and another contains 112 words, and the two e-mails have 23 words in common, then their similarity would be  $\frac{23}{42+112-23} = \frac{23}{130} = 0.18$ . We can then cluster our e-mails into groups, such that the average similarity of an e-mail to the other e-mails in its group is much larger than the average similarity to e-mails from other groups. While it wouldn't be realistic to expect that this would result in two nicely separated clusters corresponding to spam and ham – there's no magic here – the clusters may reveal some interesting and useful structure in the data. It may be possible to identify a particular kind of spam in this way, if that subgroup uses a vocabulary, or language, not found in other messages.

---

There are many other patterns that can be learned from data in an unsupervised way. *Association rules* are a kind of pattern that are popular in marketing applications, and the result of such patterns can often be found on online shopping web sites. For instance, when I looked up the book *Kernel Methods for Pattern Analysis* by John Shawe-Taylor and Nello Cristianini on [www.amazon.co.uk](http://www.amazon.co.uk), I was told that 'Customers Who Bought This Item Also Bought' –

📖 *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods* by Nello Cristianini and John Shawe-Taylor;

📖 *Pattern Recognition and Machine Learning* by Christopher Bishop;

📖 *The Elements of Statistical Learning: Data Mining, Inference and Prediction* by Trevor Hastie, Robert Tibshirani and Jerome Friedman;

📖 *Pattern Classification* by Richard Duda, Peter Hart and David Stork;

and 34 more suggestions. Such associations are found by data mining algorithms that zoom in on items that frequently occur together. These algorithms typically work by

only considering items that occur a minimum number of times (because you wouldn't want your suggestions to be based on a single customer that happened to buy these 39 books together!). More interesting associations could be found by considering multiple items in your shopping basket. There exist many other types of associations that can be learned and exploited, such as correlations between real-valued variables.

### Looking for structure

Like all other machine learning models, patterns are a manifestation of underlying structure in the data. Sometimes this structure takes the form of a single *hidden* or *latent variable*, much like unobservable but nevertheless explanatory quantities in physics, such as energy. Consider the following matrix:

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 2 \\ 1 & 0 & 1 & 1 \\ 0 & 2 & 2 & 3 \end{pmatrix}$$

Imagine these represent ratings by six different people (in rows), on a scale of 0 to 3, of four different films – say *The Shawshank Redemption*, *The Usual Suspects*, *The Godfather*, *The Big Lebowski*, (in columns, from left to right). *The Godfather* seems to be the most popular of the four with an average rating of 1.5, and *The Shawshank Redemption* is the least appreciated with an average rating of 0.5. Can you see any structure in this matrix?

If you are inclined to say no, try to look for columns or rows that are combinations of other columns or rows. For instance, the third column turns out to be the sum of the first and second columns. Similarly, the fourth row is the sum of the first and second rows. What this means is that the fourth person combines the ratings of the first and second person. Similarly, *The Godfather's* ratings are the sum of the ratings of the first two films. This is made more explicit by writing the matrix as the following product:

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 2 \\ 1 & 0 & 1 & 1 \\ 0 & 2 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

You might think I just made matters worse – instead of one matrix we now have three! However, notice that the first and third matrix on the right-hand side are now Boolean,

and the middle one is diagonal (all off-diagonal entries are zero). Moreover, these matrices have a very natural interpretation in terms of film *genres*. The right-most matrix associates films (in columns) with genres (in rows): *The Shawshank Redemption* and *The Usual Suspects* belong to two different genres, say drama and crime, *The Godfather* belongs to both, and *The Big Lebowski* is a crime film and also introduces a new genre (say comedy). The tall, 6-by-3 matrix then expresses people's preferences in terms of genres: the first, fourth and fifth person like drama, the second, fourth and fifth person like crime films, and the third, fifth and sixth person like comedies. Finally, the middle matrix states that the crime genre is twice as important as the other two genres in terms of determining people's preferences.

Methods for discovering hidden variables such as film genres really come into their own when the number of values of the hidden variable (here: the number of genres) is much smaller than the number of rows and columns of the original matrix. For instance, at the time of writing [www.imdb.com](http://www.imdb.com) lists about 630 000 rated films with 4 million people voting, but only 27 film categories (including the ones above). While it would be naive to assume that film ratings can be completely broken down by genres – genre boundaries are often diffuse, and someone may only like comedies made by the Coen brothers – this kind of *matrix decomposition* can often reveal useful hidden structure. It will be further examined in [Chapter 10](#).

This is a good moment to summarise some terminology that we will be using. We have already seen the distinction between supervised learning from labelled data and unsupervised learning from unlabelled data. We can similarly draw a distinction between whether the model output involves the target variable or not: we call it a *predictive model* if it does, and a *descriptive model* if it does not. This leads to the four different machine learning settings summarised in [Table 1.1](#).

- ☞ The most common setting is supervised learning of predictive models – in fact, this is what people commonly mean when they refer to supervised learning. Typical tasks are classification and regression.
- ☞ It is also possible to use labelled training data to build a descriptive model that is not primarily intended to predict the target variable, but instead identifies, say, subsets of the data that behave differently with respect to the target variable. This example of supervised learning of a descriptive model is called *subgroup discovery*; we will take a closer look at it in [Section 6.3](#).
- ☞ Descriptive models can naturally be learned in an unsupervised setting, and we have just seen a few examples of that (clustering, association rule discovery and matrix decomposition). This is often the implied setting when people talk about unsupervised learning.
- ☞ A typical example of unsupervised learning of a predictive model occurs when

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

**Table 1.1.** An overview of different machine learning settings. The rows refer to whether the training data is labelled with a target variable, while the columns indicate whether the models learned are used to predict a target variable or rather describe the given data.

we cluster data with the intention of using the clusters to assign class labels to new data. We will call this *predictive clustering* to distinguish it from the previous, *descriptive* form of clustering.

Although we will not cover it in this book, it is worth pointing out a fifth setting of *semi-supervised learning* of predictive models. In many problem domains data is cheap, but labelled data is expensive. For example, in web page classification you have the whole world-wide web at your disposal, but constructing a labelled training set is a painstaking process. One possible approach in semi-supervised learning is to use a small labelled training set to build an initial model, which is then refined using the unlabelled data. For example, we could use the initial model to make predictions on the unlabelled data, and use the most confident predictions as new training data, after which we retrain the model on this enlarged training set.

### *Evaluating performance on a task*

An important thing to keep in mind with all these machine learning problems is that they don't have a 'correct' answer. This is different from many other problems in computer science that you might be familiar with. For instance, if you sort the entries in your address book alphabetically on last name, there is only one correct result (unless two people have the same last name, in which case you can use some other field as tie-breaker, such as first name or age). This is not to say that there is only one way of achieving that result – on the contrary, there is a wide range of sorting algorithms available: insertion sort, bubblesort, quicksort, to name but a few. If we were to compare the performance of these algorithms, it would be in terms of how fast they are, and how much data they could handle: e.g., we could test this experimentally on real data, or analyse it using computational complexity theory. However, what we wouldn't do is compare different algorithms with respect to the correctness of the result, because an algorithm that isn't guaranteed to produce a sorted list every time is useless as a sorting algorithm.

Things are different in machine learning (and not just in machine learning: see

**Background 1.1).** We can safely assume that the perfect spam e-mail filter doesn't exist – if it did, spammers would immediately 'reverse engineer' it to find out ways to trick the spam filter into thinking a spam e-mail is actually ham. In many cases the data is 'noisy' – examples may be mislabelled, or features may contain errors – in which case it would be detrimental to try too hard to find a model that correctly classifies the training data, because it would lead to overfitting, and hence wouldn't generalise to new data. In some cases the features used to describe the data only give an indication of what their class might be, but don't contain enough 'signal' to predict the class perfectly. For these and other reasons, machine learners take performance evaluation of learning algorithms very seriously, which is why it will play a prominent role in this book. We need to have some idea of how well an algorithm is expected to perform on new data, not in terms of runtime or memory usage – although this can be an issue too – but in terms of classification performance (if our task is a classification task).

Suppose we want to find out how well our newly trained spam filter does. One thing we can do is count the number of correctly classified e-mails, both spam and ham, and divide that by the total number of examples to get a proportion which is called the *accuracy* of the classifier. However, this doesn't indicate whether overfitting is occurring. A better idea would be to use only 90% (say) of the data for training, and the remaining 10% as a *test set*. If overfitting occurs, the test set performance will be considerably lower than the training set performance. However, even if we select the test instances randomly from the data, every once in a while we may get lucky, if most of the test instances are similar to training instances – or unlucky, if the test instances happen to be very non-typical or noisy. In practice this train–test split is therefore repeated in a process called *cross-validation*, further discussed in [Chapter 12](#). This works as follows: we randomly divide the data in ten parts of equal size, and use nine parts for training and one part for testing. We do this ten times, using each part once for testing. At the end, we compute the average test set performance (and usually also its standard deviation, which is useful to determine whether small differences in average performance of different learning algorithms are meaningful). Cross-validation can also be applied to other supervised learning problems, but unsupervised learning methods typically need to be evaluated differently.

In [Chapters 2](#) and [3](#) we will take a much closer look at the various tasks that can be approached using machine learning methods. In each case we will define the task and look at different variants. We will pay particular attention to evaluating performance of models learned to solve those tasks, because this will give us considerable additional insight into the nature of the tasks.

---

Long before machine learning came into existence, philosophers knew that generalising from particular cases to general rules is not a well-posed problem with well-defined solutions. Such inference by generalisation is called *induction* and is to be contrasted with *deduction*, which is the kind of reasoning that applies to problems with well-defined correct solutions. There are many versions of this so-called *problem of induction*. One version is due to the eighteenth-century Scottish philosopher David Hume, who claimed that the only justification for induction is itself inductive: since it appears to work for certain inductive problems, it is expected to work for all inductive problems. This doesn't just say that induction cannot be deductively justified but that its justification is circular, which is much worse.

A related problem is stated by the *no free lunch theorem*, which states that no learning algorithm can outperform another when evaluated over all possible classification problems, and thus the performance of any learning algorithm, over the set of all possible learning problems, is no better than random guessing. Consider, for example, the 'guess the next number' questions popular in psychological tests: what comes after 1, 2, 4, 8, ...? If all number sequences are equally likely, then there is no hope that we can improve – on average – on random guessing (I personally always answer '42' to such questions). Of course, some sequences are very much more likely than others, at least in the world of psychological tests. Likewise, the distribution of learning problems in the real world is highly non-uniform. The way to escape the curse of the no free lunch theorem is to find out more about this distribution and exploit this knowledge in our choice of learning algorithm.

---

**Background 1.1.** Problems of induction and free lunches.

---

## 1.2 Models: the output of machine learning

Models form the central concept in machine learning as they are what is being learned from the data, in order to solve a given task. There is a considerable – not to say bewildering – range of machine learning models to choose from. One reason for this is the ubiquity of the tasks that machine learning aims to solve: classification, regression, clustering, association discovery, to name but a few. Examples of each of these tasks can be found in virtually every branch of science and engineering. Mathematicians, engineers, psychologists, computer scientists and many others have discovered – and sometimes rediscovered – ways to solve these tasks. They have all brought their



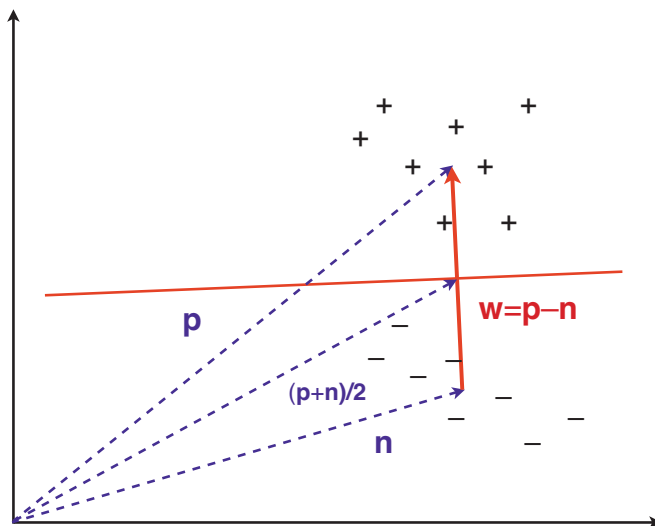
specific background to bear, and consequently the principles underlying these models are also diverse. My personal view is that this diversity is a good thing as it helps to make machine learning the powerful and exciting discipline it is. It doesn't, however, make the task of writing a machine learning book any easier! Luckily, a few common themes can be observed, which allow me to discuss machine learning models in a somewhat more systematic way. I will discuss three groups of models: geometric models, probabilistic models, and logical models. These groupings are not meant to be mutually exclusive, and sometimes a particular kind of model has, for instance, both a geometric and a probabilistic interpretation. Nevertheless, it provides a good starting point for our purposes.

### Geometric models

The *instance space* is the set of all possible or describable instances, whether they are present in our data set or not. Usually this set has some geometric structure. For instance, if all features are numerical, then we can use each feature as a coordinate in a Cartesian coordinate system. A *geometric model* is constructed directly in instance space, using geometric concepts such as lines, planes and distances. For instance, the linear classifier depicted in Figure 1 on p.5 is a geometric classifier. One main advantage of geometric classifiers is that they are easy to visualise, as long as we keep to two or three dimensions. It is important to keep in mind, though, that a Cartesian instance space has as many coordinates as there are features, which can be tens, hundreds, thousands, or even more. Such high-dimensional spaces are hard to imagine but are nevertheless very common in machine learning. Geometric concepts that potentially apply to high-dimensional spaces are usually prefixed with 'hyper-': for instance, a decision boundary in an unspecified number of dimensions is called a *hyperplane*.

If there exists a linear decision boundary separating the two classes, we say that the data is *linearly separable*. As we have seen, a linear decision boundary is defined by the equation  $\mathbf{w} \cdot \mathbf{x} = t$ , where  $\mathbf{w}$  is a vector perpendicular to the decision boundary,  $\mathbf{x}$  points to an arbitrary point on the decision boundary, and  $t$  is the decision threshold. A good way to think of the vector  $\mathbf{w}$  is as pointing from the 'centre of mass' of the negative examples,  $\mathbf{n}$ , to the centre of mass of the positives  $\mathbf{p}$ . In other words,  $\mathbf{w}$  is proportional (or equal) to  $\mathbf{p} - \mathbf{n}$ . One way to calculate these centres of mass is by averaging. For instance, if  $P$  is a set of  $n$  positive examples, then we can define  $\mathbf{p} = \frac{1}{n} \sum_{\mathbf{x} \in P} \mathbf{x}$ , and similarly for  $\mathbf{n}$ . By setting the decision threshold appropriately, we can intersect the line from  $\mathbf{n}$  to  $\mathbf{p}$  half-way (Figure 1.1). We will call this the *basic linear classifier* in this book.<sup>1</sup> It has the advantage of simplicity, being defined in terms of addition, subtraction and rescaling of examples only (in other words,  $\mathbf{w}$  is a *linear combination* of the examples). Indeed, under certain additional assumptions about the data it is the best thing we

<sup>1</sup> It is a simplified version of linear discriminants.

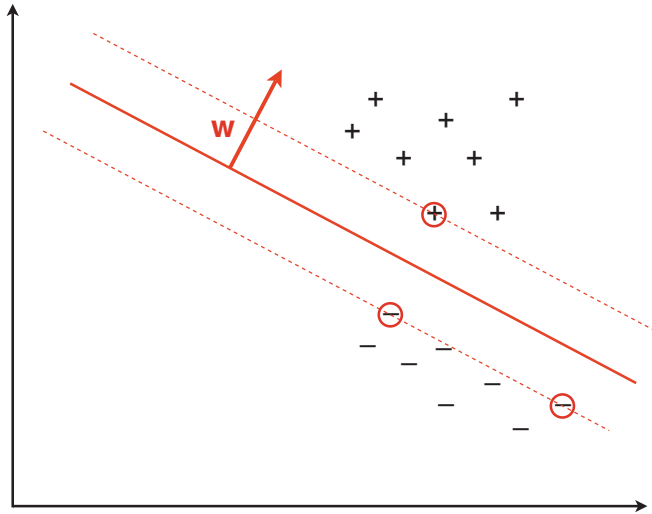


**Figure 1.1.** The basic linear classifier constructs a decision boundary by half-way intersecting the line between the positive and negative centres of mass. It is described by the equation  $\mathbf{w} \cdot \mathbf{x} = t$ , with  $\mathbf{w} = \mathbf{p} - \mathbf{n}$ ; the decision threshold can be found by noting that  $(\mathbf{p} + \mathbf{n})/2$  is on the decision boundary, and hence  $t = (\mathbf{p} - \mathbf{n}) \cdot (\mathbf{p} + \mathbf{n})/2 = (||\mathbf{p}||^2 - ||\mathbf{n}||^2)/2$ , where  $||\mathbf{x}||$  denotes the length of vector  $\mathbf{x}$ .

can hope to do, as we shall see later. However, if those assumptions do not hold, the basic linear classifier can perform poorly – for instance, note that it may not perfectly separate the positives from the negatives, even if the data is linearly separable.

Because data is usually noisy, linear separability doesn't occur very often in practice, unless the data is very sparse, as in text classification. Recall that we used a large vocabulary, say 10 000 words, each word corresponding to a Boolean feature indicating whether or not that word occurs in the document. This means that the instance space has 10 000 dimensions, but for any one document no more than a small percentage of the features will be non-zero. As a result there is much 'empty space' between instances, which increases the possibility of linear separability. However, because linearly separable data doesn't uniquely define a decision boundary, we are now faced with a problem: which of the infinitely many decision boundaries should we choose? One natural option is to prefer large margin classifiers, where the *margin* of a linear classifier is the distance between the decision boundary and the closest instance. [Support vector machines](#), discussed in [Chapter 7](#), are a powerful kind of linear classifier that find a decision boundary whose margin is as large as possible ([Figure 1.2](#)).

Geometric concepts, in particular linear transformations, can be very helpful to understand the similarities and differences between machine learning methods ([Background 1.2](#)). For instance, we would expect most if not all learning algorithms



**Figure 1.2.** The decision boundary learned by a support vector machine from the linearly separable data from Figure 1. The decision boundary maximises the margin, which is indicated by the dotted lines. The circled data points are the support vectors.

to be translation-invariant, i.e., insensitive to where we put the origin of our coordinate system. Some algorithms may also be rotation-invariant, e.g., linear classifiers or support vector machines; but many others aren't, including Bayesian classifiers. Similarly, some algorithms may be sensitive to non-uniform scaling.

A very useful geometric concept in machine learning is the notion of *distance*. If the distance between two instances is small then the instances are similar in terms of their feature values, and so nearby instances would be expected to receive the same classification or belong to the same cluster. In a Cartesian coordinate system, distance can be measured by *Euclidean distance*, which is the square root of the sum of the squared distances along each coordinate:<sup>2</sup>  $\sqrt{\sum_{i=1}^d (x_i - y_i)^2}$ . A very simple distance-based classifier works as follows: to classify a new instance, we retrieve from memory the most similar training instance (i.e., the training instance with smallest Euclidean distance from the instance to be classified), and simply assign that training instance's class. This classifier is known as the *nearest-neighbour classifier*. Endless variations on this simple yet powerful theme exist: we can retrieve the  $k$  most similar training instances and take a vote ( $k$ -nearest neighbour); we can weight each neighbour's vote inversely to its distance; we can apply the same idea to regression tasks by averaging the training instances' function value; and so on. What they all have in common is that predictions are local in the sense that they are based on only a few training instances,

<sup>2</sup>This can be expressed in vector notation as  $\|\mathbf{x} - \mathbf{y}\| = \sqrt{(\mathbf{x} - \mathbf{y}) \cdot (\mathbf{x} - \mathbf{y})} = \sqrt{\mathbf{x} \cdot \mathbf{x} - 2\mathbf{x} \cdot \mathbf{y} + \mathbf{y} \cdot \mathbf{y}} = \sqrt{\|\mathbf{x}\|^2 - 2\|\mathbf{x}\|\|\mathbf{y}\|\cos\theta + \|\mathbf{y}\|^2}$ , where  $\theta$  is the angle between  $\mathbf{x}$  and  $\mathbf{y}$ .

Transformations in  $d$ -dimensional Cartesian coordinate systems can be conveniently represented by means of matrix notation. Let  $\mathbf{x}$  be a  $d$ -vector representing a data point, then  $\mathbf{x} + \mathbf{t}$  is the resulting point after *translating* over  $\mathbf{t}$  (another  $d$ -vector). Translating a set of points over  $\mathbf{t}$  can be equivalently understood as translating the origin over  $-\mathbf{t}$ . Using *homogeneous coordinates* – the addition of an extra dimension set to 1 – translations can be expressed by matrix multiplication: e.g., in two dimensions we have

$$\mathbf{x}^\circ = \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix} \quad \mathbf{T} = \begin{pmatrix} 1 & 0 & 0 \\ t_1 & 1 & 0 \\ t_2 & 0 & 1 \end{pmatrix} \quad \mathbf{T}\mathbf{x}^\circ = \begin{pmatrix} 1 \\ x_1 + t_1 \\ x_2 + t_2 \end{pmatrix}$$

A *rotation* is defined by any  $d$ -by- $d$  matrix  $\mathbf{D}$  whose transpose is its inverse (which means it is orthogonal) and whose determinant is 1. In two dimensions a rotation matrix can be written as  $\mathbf{R} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$ , representing a clockwise rotation over angle  $\theta$  about the origin. For instance,  $\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$  is a 90 degrees clockwise rotation.

A *scaling* is defined by a diagonal matrix; in two dimensions  $\mathbf{S} = \begin{pmatrix} s_1 & 0 \\ 0 & s_2 \end{pmatrix}$ . A *uniform scaling* applies the same scaling factor  $s$  in all dimensions and can be written as  $s\mathbf{I}$ , where  $\mathbf{I}$  is the identity matrix. Notice that a uniform scaling with scaling factor  $-1$  is a rotation (over 180 degrees in the two-dimensional case).

A common scenario which utilises all these transformations is the following. Given an  $n$ -by- $d$  matrix  $\mathbf{X}$  representing  $n$  data points in  $d$ -dimensional space, we first calculate the centre of mass or mean vector  $\boldsymbol{\mu}$  by averaging each column. We then zero-centre the data set by subtracting  $-\boldsymbol{\mu}$  from each row, which corresponds to a translation. Next, we rotate the data such that as much variance (a measure of the data's 'spread' in a certain direction) as possible is aligned with our coordinate axes; this can be achieved by a matrix transformation known as *principal component analysis*, about which you will learn more in [Chapter 10](#). Finally, we scale the data to unit variance along each coordinate.

---

**Background 1.2.** Linear transformations.

---

rather than being derived from a global model built from the entire data set.

There is a nice relationship between Euclidean distance and the mean of a set of

points: there is no other point which has smaller total squared Euclidean distance to the given points (see [Theorem 8.1](#) on [p.238](#) for a proof of this). Consequently, we can use the mean of a set of nearby points as a representative *exemplar* for those points. Suppose we want to cluster our data into  $K$  clusters, and we have an initial guess of how the data should be clustered. We then calculate the means of each initial cluster, and reassign each point to the nearest cluster mean. Unless our initial guess was a lucky one, this will have changed some of the clusters, so we repeat these two steps (calculating the cluster means and reassigning points to clusters) until no change occurs. This clustering algorithm, which is called *K-means* and is further discussed in [Chapter 8](#), is very widely used to solve a range of clustering tasks. It remains to be decided how we construct our initial guess. This is usually done randomly: either by randomly partitioning the data set into  $K$  ‘clusters’ or by randomly guessing  $K$  ‘cluster centres’. The fact that these initial ‘clusters’ or ‘cluster centres’ will bear little resemblance to the actual data is not a problem, as this will quickly be rectified by running the algorithm for a number of iterations.

To summarise, geometric notions such as planes, translations and rotations, and distance are very useful in machine learning as they allow us to understand many key concepts in intuitive ways. Geometric models exploit these intuitions and are simple, powerful and allow many variations with little effort. For instance, instead of using Euclidean distance, which can be sensitive to outliers, other distances can be used such as *Manhattan distance*, which sums the distances along each coordinate:  $\sum_{i=1}^d |x_i - y_i|$ .

### Probabilistic models

The second type of models are probabilistic in nature, like the Bayesian classifier we considered earlier. Many of these models are based around the following idea. Let  $X$  denote the variables we know about, e.g., our instance’s feature values; and let  $Y$  denote the *target variables* we’re interested in, e.g., the instance’s class. The key question in machine learning is how to model the relationship between  $X$  and  $Y$ . The statistician’s approach is to assume that there is some underlying random process that generates the values for these variables, according to a well-defined but unknown probability distribution. We want to use the data to find out more about this distribution. Before we look into that, let’s consider how we could use that distribution once we have learned it.

Since  $X$  is known for a particular instance but  $Y$  may not be, we are particularly interested in the conditional probabilities  $P(Y|X)$ . For instance,  $Y$  could indicate whether the e-mail is spam, and  $X$  could indicate whether the e-mail contains the words ‘Viagra’ and ‘lottery’. The probability of interest is then  $P(Y|\text{Viagra, lottery})$ , with Viagra and lottery two Boolean variables which together constitute the feature vector  $X$ . For a particular e-mail we know the feature values and so we might write  $P(Y|\text{Viagra} =$

Viagra	lottery	$P(Y = \text{spam} \text{Viagra}, \text{lottery})$	$P(Y = \text{ham} \text{Viagra}, \text{lottery})$
0	0	0.31	<b>0.69</b>
0	1	<b>0.65</b>	0.35
1	0	<b>0.80</b>	0.20
1	1	0.40	<b>0.60</b>

**Table 1.2.** An example posterior distribution. ‘Viagra’ and ‘lottery’ are two Boolean features;  $Y$  is the class variable, with values ‘spam’ and ‘ham’. In each row the most likely class is indicated in bold.

1, lottery = 0) if the e-mail contains the word ‘Viagra’ but not the word ‘lottery’. This is called a *posterior probability* because it is used *after* the features  $X$  are observed.

Table 1.2 shows an example of how these probabilities might be distributed. From this distribution you can conclude that, if an e-mail doesn’t contain the word ‘Viagra’, then observing the word ‘lottery’ increases the probability of the e-mail being spam from 0.31 to 0.65; but if the e-mail does contain the word ‘Viagra’, then observing the word ‘lottery’ as well decreases the spam probability from 0.80 to 0.40. Even though this example table is small, it will grow unfeasibly large very quickly (with  $n$  Boolean variables  $2^n$  cases have to be distinguished). We therefore don’t normally have access to the full joint distribution and have to approximate it using additional assumptions, as we will see below.

Assuming that  $X$  and  $Y$  are the only variables we know and care about, the posterior distribution  $P(Y|X)$  helps us to answer many questions of interest. For instance, to classify a new e-mail we determine whether the words ‘Viagra’ and ‘lottery’ occur in it, look up the corresponding probability  $P(Y = \text{spam}|\text{Viagra}, \text{lottery})$ , and predict spam if this probability exceeds 0.5 and ham otherwise. Such a recipe to predict a value of  $Y$  on the basis of the values of  $X$  and the posterior distribution  $P(Y|X)$  is called a *decision rule*. We can do this even without knowing all the values of  $X$ , as the following example shows.

**Example 1.2 (Missing values).** Suppose we skimmed an e-mail and noticed that it contains the word ‘lottery’ but we haven’t looked closely enough to determine whether it uses the word ‘Viagra’. This means that we don’t know whether to use the second or the fourth row in Table 1.2 to make a prediction. This is a problem, as we would predict spam if the e-mail contained the word ‘Viagra’ (second row) and ham if it didn’t (fourth row).

The solution is to average these two rows, using the probability of ‘Viagra’

occurring in any e-mail (spam or not):

$$P(Y|\text{lottery}) = P(Y|\text{Viagra} = 0, \text{lottery})P(\text{Viagra} = 0) \\ + P(Y|\text{Viagra} = 1, \text{lottery})P(\text{Viagra} = 1)$$

For instance, suppose for the sake of argument that one in ten e-mails contain the word 'Viagra', then  $P(\text{Viagra} = 1) = 0.10$  and  $P(\text{Viagra} = 0) = 0.90$ . Using the above formula, we obtain  $P(Y = \text{spam}|\text{lottery} = 1) = 0.65 \cdot 0.90 + 0.40 \cdot 0.10 = 0.625$  and  $P(Y = \text{ham}|\text{lottery} = 1) = 0.35 \cdot 0.90 + 0.60 \cdot 0.10 = 0.375$ . Because the occurrence of 'Viagra' in any e-mail is relatively rare, the resulting distribution deviates only a little from the second row in Table 1.2.

As a matter of fact, statisticians work very often with different conditional probabilities, given by the *likelihood function*  $P(X|Y)$ .<sup>3</sup> This seems counter-intuitive at first: why would we be interested in the probability of an event we know has occurred ( $X$ ), conditioned on something we don't know anything about ( $Y$ )? I like to think of these as thought experiments: if somebody were to send me a spam e-mail, how likely would it be that it contains exactly the words of the e-mail I'm looking at? And how likely if it were a ham e-mail instead? 'Not very likely at all in either case', you might think, and you would be right: with so many words to choose from, the probability of any particular combination of words would be very small indeed. What really matters is not the magnitude of these likelihoods, but their ratio: how much more likely is it to observe this combination of words in a spam e-mail than it is in a non-spam e-mail. For instance, suppose that for a particular e-mail described by  $X$  we have  $P(X|Y = \text{spam}) = 3.5 \cdot 10^{-5}$  and  $P(X|Y = \text{ham}) = 7.4 \cdot 10^{-6}$ , then observing  $X$  in a spam e-mail is nearly five times more likely than it is in a ham e-mail. This suggests the following decision rule: predict spam if the likelihood ratio is larger than 1 and ham otherwise.

So which one should we use: posterior probabilities or likelihoods? As it turns out, we can easily transform one into the other using *Bayes' rule*, a simple property of conditional probabilities which states that

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}$$

Here,  $P(Y)$  is the *prior probability*, which in the case of classification tells me how likely each of the classes is *a priori*, i.e., before I have observed the data  $X$ .  $P(X)$  is the prob-

<sup>3</sup>It is called the likelihood function rather than the 'likelihood distribution' because, for fixed  $X$ ,  $P(X|Y)$  is a mapping from  $Y$  to probabilities, but these don't sum to 1 and therefore don't establish a probability distribution over  $Y$ .

ability of the data, which is independent of  $Y$  and in most cases can be ignored (or inferred in a normalisation step, as it is equal to  $\sum_Y P(X|Y)P(Y)$ ). The first decision rule above suggested that we predict the class with maximum posterior probability, which using Bayes' rule can be written in terms of the likelihood function:

$$y_{\text{MAP}} = \arg\max_Y P(Y|X) = \arg\max_Y \frac{P(X|Y)P(Y)}{P(X)} = \arg\max_Y P(X|Y)P(Y)$$

This is usually called the *maximum a posteriori (MAP)* decision rule. Now, if we assume a uniform prior distribution (i.e.,  $P(Y)$  the same for every value of  $Y$ ) this reduces to the *maximum likelihood (ML)* decision rule:

$$y_{\text{ML}} = \arg\max_Y P(X|Y)$$

A useful rule of thumb is: *use likelihoods if you want to ignore the prior distribution or assume it uniform, and posterior probabilities otherwise.*

If we have only two classes it is convenient to work with ratios of posterior probabilities or likelihood ratios. If we want to know how much the data favours one of two classes, we can calculate the *posterior odds*: e.g.,

$$\frac{P(Y = \text{spam}|X)}{P(Y = \text{ham}|X)} = \frac{P(X|Y = \text{spam})}{P(X|Y = \text{ham})} \frac{P(Y = \text{spam})}{P(Y = \text{ham})}$$

In words: the *posterior odds* are the product of the *likelihood ratio* and the *prior odds*. If the odds are larger than 1 we conclude that the class in the numerator is the more likely of the two; if it is smaller than 1 we take the class in the denominator instead. In very many cases the prior odds is a simple constant factor that can be manually set, estimated from the data, or optimised to maximise performance on a test set.

**Example 1.3 (Posterior odds).** Using the data from Table 1.2, and assuming a uniform prior distribution, we arrive at the following posterior odds:

$$\begin{aligned} \frac{P(Y = \text{spam}|\text{Viagra} = 0, \text{lottery} = 0)}{P(Y = \text{ham}|\text{Viagra} = 0, \text{lottery} = 0)} &= \frac{0.31}{0.69} = 0.45 \\ \frac{P(Y = \text{spam}|\text{Viagra} = 1, \text{lottery} = 1)}{P(Y = \text{ham}|\text{Viagra} = 1, \text{lottery} = 1)} &= \frac{0.40}{0.60} = 0.67 \\ \frac{P(Y = \text{spam}|\text{Viagra} = 0, \text{lottery} = 1)}{P(Y = \text{ham}|\text{Viagra} = 0, \text{lottery} = 1)} &= \frac{0.65}{0.35} = 1.9 \\ \frac{P(Y = \text{spam}|\text{Viagra} = 1, \text{lottery} = 0)}{P(Y = \text{ham}|\text{Viagra} = 1, \text{lottery} = 0)} &= \frac{0.80}{0.20} = 4.0 \end{aligned}$$

Using a MAP decision rule (which in this case is the same as the ML decision rule, since we assumed a uniform prior) we predict ham in the top two cases and spam



$Y$	$P(\text{Viagra} = 1 Y)$	$P(\text{Viagra} = 0 Y)$	$Y$	$P(\text{lottery} = 1 Y)$	$P(\text{lottery} = 0 Y)$
spam	0.40	0.60	spam	0.21	0.79
ham	0.12	0.88	ham	0.13	0.87

Table 1.3. Example marginal likelihoods.

in the bottom two. Given that the full posterior distribution is all there is to know about the domain in a statistical sense, these predictions are the best we can do: they are *Bayes-optimal*.

It is clear from the above analysis that the likelihood function plays an important role in statistical machine learning. It establishes what is called a *generative model*: a probabilistic model from which we can sample values of all variables involved. Imagine a box with two buttons labelled ‘ham’ and ‘spam’. Pressing the ‘ham’ button generates a random e-mail according to  $P(X|Y = \text{ham})$ ; pressing the ‘spam’ button generates a random e-mail according to  $P(X|Y = \text{spam})$ . The question now is what we put inside the box. Let’s try a model that is so simplistic it’s almost laughable. Assuming a vocabulary of 10 000 words, you have two bags with 10 000 coins each, one for each word in the vocabulary. In order to generate a random e-mail, you take the appropriate bag depending on which button was pressed, and toss each of the 10 000 coins in that bag to decide which words should go in the e-mail (say heads is in and tails is out).

In statistical terms, each coin – which isn’t necessarily fair – represents a parameter of the model, so we have 20 000 parameters. If ‘Viagra’ is a word in the vocabulary, then the coin labelled ‘Viagra’ in the bag labelled ‘spam’ represents  $P(\text{Viagra}|Y = \text{spam})$  and the coin labelled ‘Viagra’ in the bag labelled ‘ham’ represents  $P(\text{Viagra}|Y = \text{ham})$ . Together, these two coins represent the left table in Table 1.3. Notice that by using different coins for each word we have tacitly assumed that likelihoods of individual words are independent within the same class, which – if true – allows us to decompose the joint likelihood into a product of *marginal likelihoods*:

$$P(\text{Viagra}, \text{lottery}|Y) = P(\text{Viagra}|Y)P(\text{lottery}|Y)$$

Effectively, this independence assumption means that knowing whether one word occurs in the e-mail doesn’t tell you anything about the likelihood of other words. The probabilities on the right are called marginal likelihoods because they are obtained by ‘marginalising’ some of the variables in the joint distribution: e.g.,  $P(\text{Viagra}|Y) = \sum_{\text{lottery}} P(\text{Viagra}, \text{lottery}|Y)$ .

**Example 1.4 (Using marginal likelihoods).** Assuming these estimates come out as in Table 1.3, we can then calculate likelihood ratios (the previously calculated odds from the full posterior distribution are shown in brackets):

$$\frac{P(\text{Viagra} = 0|Y = \text{spam})}{P(\text{Viagra} = 0|Y = \text{ham})} \frac{P(\text{lottery} = 0|Y = \text{spam})}{P(\text{lottery} = 0|Y = \text{ham})} = \frac{0.60}{0.88} \frac{0.79}{0.87} = 0.62 \quad (0.45)$$

$$\frac{P(\text{Viagra} = 0|Y = \text{spam})}{P(\text{Viagra} = 0|Y = \text{ham})} \frac{P(\text{lottery} = 1|Y = \text{spam})}{P(\text{lottery} = 1|Y = \text{ham})} = \frac{0.60}{0.88} \frac{0.21}{0.13} = 1.1 \quad (1.9)$$

$$\frac{P(\text{Viagra} = 1|Y = \text{spam})}{P(\text{Viagra} = 1|Y = \text{ham})} \frac{P(\text{lottery} = 0|Y = \text{spam})}{P(\text{lottery} = 0|Y = \text{ham})} = \frac{0.40}{0.12} \frac{0.79}{0.87} = 3.0 \quad (4.0)$$

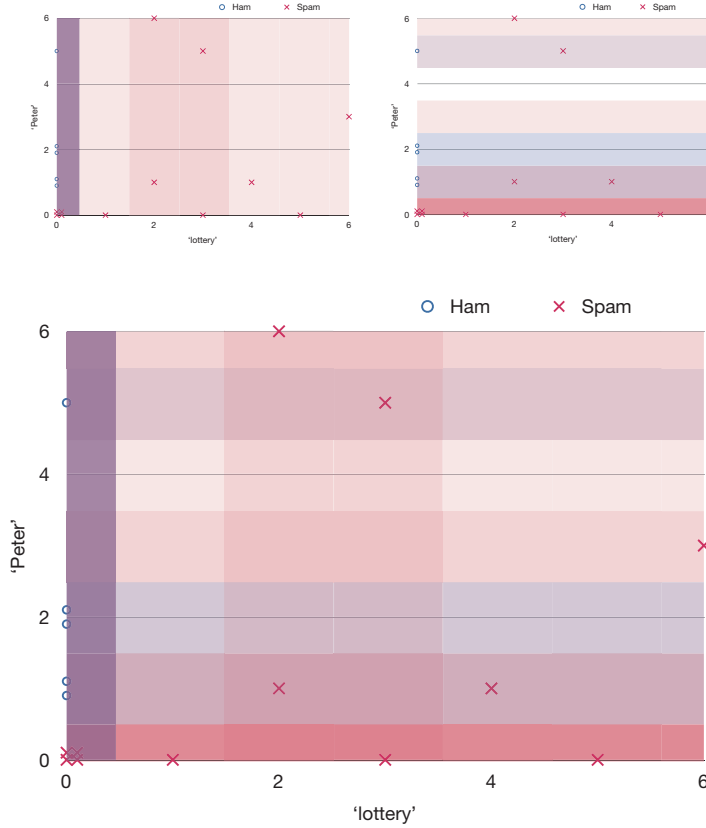
$$\frac{P(\text{Viagra} = 1|Y = \text{spam})}{P(\text{Viagra} = 1|Y = \text{ham})} \frac{P(\text{lottery} = 1|Y = \text{spam})}{P(\text{lottery} = 1|Y = \text{ham})} = \frac{0.40}{0.12} \frac{0.21}{0.13} = 5.4 \quad (0.67)$$

We see that, using a maximum likelihood decision rule, our very simple model arrives at the Bayes-optimal prediction in the first three cases, but not in the fourth ('Viagra' and 'lottery' both present), where the marginal likelihoods are actually very misleading. A possible explanation is that these terms are very unlikely to occur together in any e-mail, but slightly more likely in ham than spam – for instance, I might be making exactly this point in an e-mail!

One might call the independence assumption that allows us to decompose joint likelihoods into a product of marginal likelihoods 'naive' – which is exactly what machine learners do when they refer to this simplified Bayesian classifier as *naive Bayes*. This shouldn't be taken as a derogatory term – on the contrary, it illustrates a very important guideline in machine learning: *everything should be made as simple as possible, but not simpler*.<sup>4</sup> In our statistical context, this rule boils down to using the simplest generative model that solves our task. For instance, we may decide to stick to naive Bayes on the grounds that the cases in which the marginal probabilities are misleading are very unlikely to occur in reality and therefore will be difficult to learn from data.

We now have some idea what a probabilistic model looks like, but how do we learn such a model? In many cases this will be a matter of estimating the model parameters from data, which is usually achieved by straightforward counting. For example, in the coin toss model of spam recognition we had two coins for every word  $w_i$  in our vocab-

<sup>4</sup>This formulation is often attributed to Einstein, although the source is unclear. Other rules in the same spirit include 'Entities should not be multiplied unnecessarily' (called *Occam's razor*, after William of Ockham); 'We are to admit no more causes of natural things than such as are both true and sufficient to explain their appearances' (Isaac Newton); and 'Scientists must use the simplest means of arriving at their results and exclude everything not perceived by the senses' (Ernst Mach). Whether any of these rules are more than methodological rules of thumbs and point to some fundamental property of nature is heavily debated.



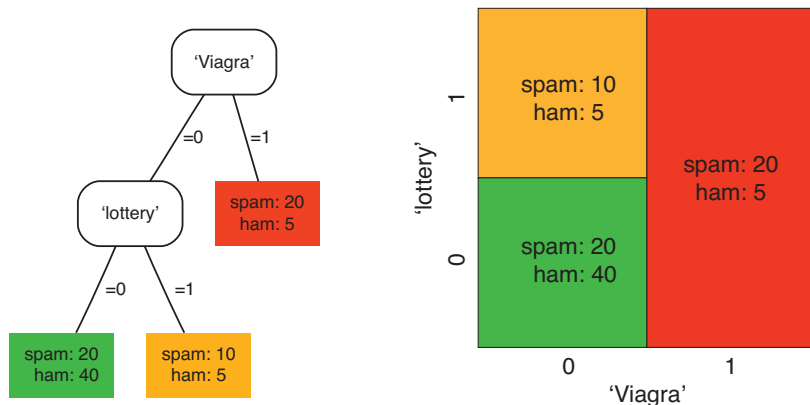
**Figure 1.3. (top)** Visualisation of two marginal likelihoods as estimated from a small data set. The colours indicate whether the likelihood points to **spam** or **ham**. **(bottom)** Combining the two marginal likelihoods gives a pattern not unlike that of a Scottish tartan. The colour of a particular cell is a result of the colours in the corresponding row and column.

ulary, one of which is to be tossed if we are generating a spam e-mail and the other for ham e-mails. Let's say that the spam coin comes up heads with probability  $\theta_i^{\oplus}$  and the ham coin with probability  $\theta_i^{\ominus}$ , then these parameters characterise all the likelihoods:

$$\begin{aligned} P(w_i = 1 | Y = \text{spam}) &= \theta_i^{\oplus} & P(w_i = 0 | Y = \text{spam}) &= 1 - \theta_i^{\oplus} \\ P(w_i = 1 | Y = \text{ham}) &= \theta_i^{\ominus} & P(w_i = 0 | Y = \text{ham}) &= 1 - \theta_i^{\ominus} \end{aligned}$$

In order to estimate the parameters  $\theta_i^{\pm}$  we need a training set of e-mails labelled spam or ham. We take the spam e-mails and count how many of them  $w_i$  occurs in: dividing by the total number of spam e-mails gives us an estimate of  $\theta_i^{\oplus}$ . Repeating this for the ham e-mails results in an estimate of  $\theta_i^{\ominus}$ . And that's all there is to it!<sup>5</sup>

<sup>5</sup>Sometimes we need to slightly adapt the raw counts for very frequent or very infrequent words, as we shall see in [Section 2.3](#).

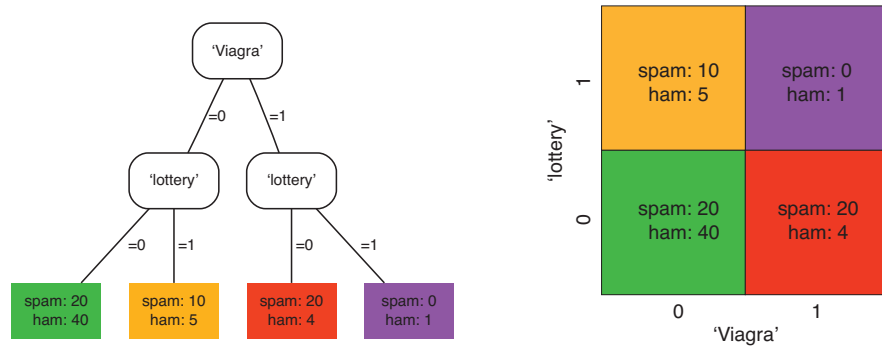


**Figure 1.4. (left)** A feature tree combining two Boolean features. Each internal node or split is labelled with a feature, and each edge emanating from a split is labelled with a feature value. Each leaf therefore corresponds to a unique combination of feature values. Also indicated in each leaf is the class distribution derived from the training set. **(right)** A feature tree partitions the instance space into rectangular regions, one for each leaf. We can clearly see that the majority of ham lives in the lower left-hand corner.

Figure 1.3 visualises this for a variant of the naive Bayes classifier discussed above. In this variant, we record the number of times a particular word occurs in an e-mail, rather than just whether it occurs or not. We thus need a parameter  $p_{ij\pm}$  for each likelihood  $P(w_i = j | Y = \pm)$ , where  $j = 0, 1, 2, \dots$ . For example, we see that there are two spam e-mails in which 'lottery' occurs twice, and one ham e-mail in which 'Peter' occurs five times. Combining the two sets of marginal likelihoods, we get the tartan-like pattern of Figure 1.3 (bottom), which is why I like to call naive Bayes the 'Scottish classifier'. This is a visual reminder of the fact that a multivariate naive Bayes model *decomposes* into a bunch of univariate ones. We will return to this issue of decomposition several times in the book.

### Logical models

The third type of model we distinguish is more algorithmic in nature, drawing inspiration from computer science and engineering. I call this type 'logical' because models of this type can be easily translated into rules that are understandable by humans, such as *if Viagra = 1 then Class = Y = spam*. Such rules are easily organised in a tree structure, such as the one in Figure 1.4, which I will call a *feature tree*. The idea of such a tree is that features are used to iteratively partition the instance space. The leaves of the tree therefore correspond to rectangular areas in the instance space (or hyper-rectangles, more generally) which we will call *instance space segments*, or segments for short. Depending on the task we are solving, we can then label the leaves with a class, a



**Figure 1.5. (left)** A complete feature tree built from two Boolean features. **(right)** The corresponding instance space partition is the finest partition that can be achieved with those two features.

probability, a real value, and so on. Feature trees whose leaves are labelled with classes are commonly called *decision trees*.

**Example 1.5 (Labelling a feature tree).** The leaves of the tree in Figure 1.4 could be labelled, from left to right, as ham – spam – spam, employing a simple decision rule called *majority class*. Alternatively, we could label them with the proportion of spam e-mail occurring in each leaf: from left to right, 1/3, 2/3, and 4/5. Or, if our task was a regression task, we could label the leaves with predicted real values or even linear functions of some other, real-valued features.

Feature trees are very versatile and will play a major role in this book. Even models that do not appear tree-based at first sight can be understood as being built on a feature tree. Consider, for instance, the naive Bayes classifier discussed previously. Since it employs marginal likelihoods such as the ones in Table 1.3 on p.29, it partitions the instance space in as many regions as there are combinations of feature values. This means that it can be thought of as employing a *complete* feature tree, which contains all features, one at each level of the tree (Figure 1.5). Incidentally, notice that the right-most leaf is the one where naive Bayes made a wrong prediction. Since this leaf covers only a single example, there is a danger that this tree is overfitting the data and that the previous tree is a better model. Decision tree learners often employ *pruning* techniques which delete splits such as these.

A *feature list* is a binary feature tree which always branches in the same direction, either left or right. The tree in Figure 1.4 is a left-branching feature list. Such feature

lists can be written as nested if-then-else statements that will be familiar to anyone with a bit of programming experience. For instance, if we were to label the leaves in Figure 1.4 by majority class we obtain the following *decision list*:

```
·if Viagra = 1 then Class = Y = spam·
·else if lottery = 1 then Class = Y = spam·
·else Class = Y = ham·
```

Logical models often have different, equivalent formulations. For instance, two alternative formulations for this model are

```
·if Viagra = 1  $\vee$  lottery = 1 then Class = Y = spam·
·else Class = Y = ham·

·if Viagra = 0  $\wedge$  lottery = 0 then Class = Y = ham·
·else Class = Y = spam·
```

The first of these alternative formulations combines the two rules in the original decision list by means of *disjunction* ('or'), denoted by  $\vee$ . This selects a single non-rectangular area in instance space. The second model formulates a *conjunctive* condition ('and', denoted by  $\wedge$ ) for the opposite class (ham) and declares everything else as spam.

We can also represent the same model as un-nested rules:

```
·if Viagra = 1 then Class = Y = spam·
·if Viagra = 0  $\wedge$  lottery = 1 then Class = Y = spam·
·if Viagra = 0  $\wedge$  lottery = 0 then Class = Y = ham·
```

Here, every path from root to a leaf is translated into a rule. As a result, although rules from the same sub-tree share conditions (such as  $\text{Viagra} = 0$ ), every pair of rules will have at least some mutually exclusive conditions (such as  $\text{lottery} = 1$  in the second rule and  $\text{lottery} = 0$  in the third). However, this is not always the case: rules can have a certain overlap.

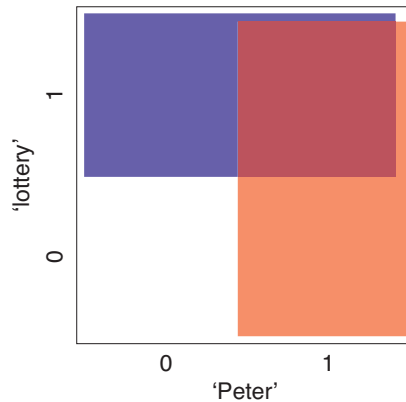
---

**Example 1.6 (Overlapping rules).** Consider the following rules:

```
·if lottery = 1 then Class = Y = spam·
·if Peter = 1 then Class = Y = ham·
```

As can be seen in Figure 1.6, these rules overlap for  $\text{lottery} = 1 \wedge \text{Peter} = 1$ , for which they make contradictory predictions. Furthermore, they fail to make any predictions for  $\text{lottery} = 0 \wedge \text{Peter} = 0$ .

---



**Figure 1.6.** The effect of overlapping rules in instance space. The two rules make contradictory predictions in the top right-hand corner, and no prediction at all in the bottom left-hand corner.

A logician would say that rules such as these are both *inconsistent* and *incomplete*. To address incompleteness, we could add a *default rule* to predict, e.g., the majority class for instances not covered by any rule. There are a number of options to deal with overlapping rules, which will be further considered in [Chapter 6](#).

Tree-learning algorithms typically work in a top-down fashion. The first task is to find a good feature to split on at the top of the tree. The aim here is to find splits that result in improved purity of the nodes on the next level, where the purity of a node refers to the degree in which the training examples belonging to that node are of the same class. Once the algorithm has found such a feature, the training set is partitioned into subsets, one for each node resulting from the split. For each of these subsets, we again find a good feature to split on, and so on. An algorithm that works by repeatedly splitting a problem into small sub-problems is what computer scientists call a divide-and-conquer algorithm. We stop splitting a node when all training examples belonging to that node are of the same class. Most rule learning algorithms also work in a top-down fashion. We learn a single rule by repeatedly adding conditions to the rule until the rule only covers examples of a single class. We then remove the covered examples of that class, and repeat the process. This is sometimes called a separate-and-conquer approach.

An interesting aspect of logical models, which sets them aside from most geometric and probabilistic models, is that they can, to some extent, provide *explanations* for their predictions. For example, a prediction assigned by a decision tree could be explained by reading off the conditions that led to the prediction from root to leaf. The model itself can also easily be inspected by humans, which is why they are sometimes called *declarative*. Declarative models do not need to be restricted to the simple rules that we have considered so far. The logical rule learning system [Progol](#) found the

following set of conditions to predict whether a molecular compound is carcinogenic (causes cancer):

1. it tests positive in the Salmonella assay; or
2. it tests positive for sex-linked recessive lethal mutation in *Drosophila*; or
3. it tests negative for chromosome aberration; or
4. it has a carbon in a six-membered aromatic ring with a partial charge of  $-0.13$ ; or
5. it has a primary amine group and no secondary or tertiary amines; or
6. it has an aromatic (or resonant) hydrogen with partial charge  $\geq 0.168$ ; or
7. it has a hydroxy oxygen with a partial charge  $\geq -0.616$  and an aromatic (or resonant) hydrogen; or
8. it has a bromine; or
9. it has a tetrahedral carbon with a partial charge  $\leq -0.144$  and tests positive on Progol's mutagenicity rules.<sup>6</sup>

The first three conditions concerned certain tests that were carried out for all molecules and whose results were recorded in the data as Boolean features. In contrast, the remaining six rules all refer to the structure of the molecule and were constructed entirely by Progol. For instance, rule 4 predicts that a molecule is carcinogenic if it contains a carbon atom with certain properties. This condition is different from the first three in that it is not a pre-recorded feature in the data, but a new feature that is constructed by Progol during the learning process because it helps to explain the data.

### Grouping and grading

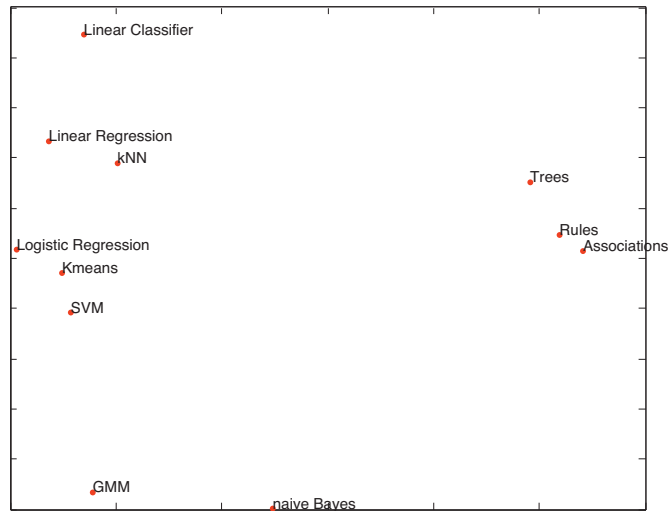
We have looked at three general types of models: geometric models, probabilistic models and logical models. As I indicated, although there are some underlying principles pertaining to each of these groups of models, the main reason for dividing things up along this dimension is one of convenience. Before I move on to the third main ingredient of machine learning, features, I want to briefly introduce another important but somewhat more abstract dimension that is in some sense orthogonal to the geometric–probabilistic–logical dimension. This is the distinction between *grouping models* and *grading models*. The key difference between these models is the way they handle the instance space.

Grouping models do this by breaking up the instance space into groups or *segments*, the number of which is determined at training time. One could say that grouping models have a fixed and finite ‘resolution’ and cannot distinguish between individual instances beyond this resolution. What grouping models do at this finest resolution

---

<sup>6</sup>Mutagenic molecules cause mutations in DNA and are often carcinogenic. This last rule refers to a set of rules that was learned earlier by Progol to predict mutagenicity.



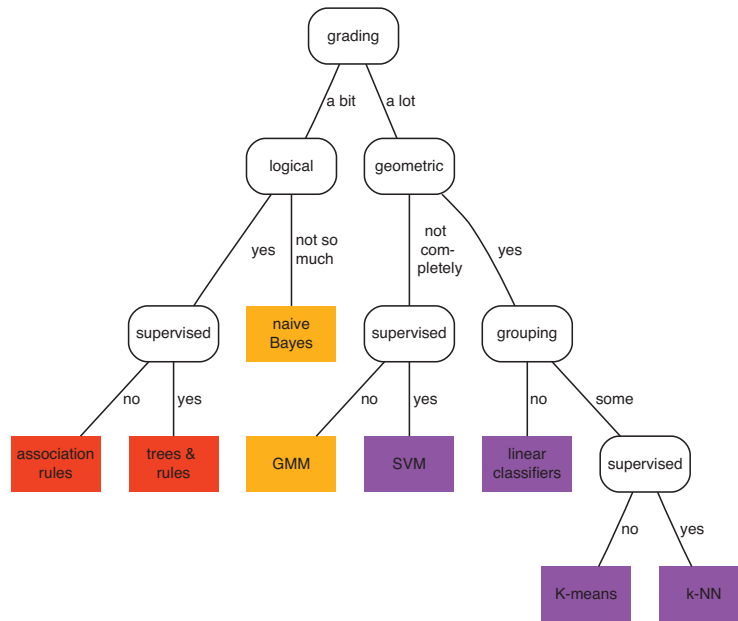


**Figure 1.7.** A ‘map’ of some of the models that will be considered in this book. Models that share characteristics are plotted closer together: logical models to the right, geometric models on the top left and probabilistic models on the bottom left. The horizontal dimension roughly ranges from grading models on the left to grouping models on the right.

is often something very simple, such as assigning the majority class to all instances that fall into the segment. The main emphasis of training a grouping model is then on determining the right segments so that we can get away with this very simple labelling at the local segment level. Grading models, on the other hand, do not employ such a notion of segment. Rather than applying very simple, local models, they form one global model over the instance space. Consequently, grading models are (usually) able to distinguish between arbitrary instances, no matter how similar they are. Their resolution is, in theory, infinite, particularly when working in a Cartesian instance space.

A good example of grouping models are the tree-based models we have just considered. They work by repeatedly splitting the instance space into smaller subsets. Because trees are usually of limited depth and don’t contain all the available features, the subsets at the leaves of the tree partition the instance space with some finite resolution. Instances filtered into the same leaf of the tree are treated the same, regardless of any features not in the tree that might be able to distinguish them. Support vector machines and other geometric classifiers are examples of grading models. Because they work in a Cartesian instance space, they are able to represent and exploit the minutest differences between instances. As a consequence, it is always possible to come up with a new test instance that receives a score that has not been given to any previous test instance.

The distinction between grouping and grading models is relative rather than



**Figure 1.8.** A taxonomy describing machine learning methods in terms of the extent to which they are grading or grouping models, logical, geometric or a combination, and supervised or unsupervised. The colours indicate the type of model, from left to right: logical (red), probabilistic (orange) and geometric (purple).

absolute, and some models combine both features. For instance, even though linear classifiers are a prime example of a grading model, it is easy to think of instances that a linear model can't distinguish, namely instances on a line or plane parallel to the decision boundary. The point is not so much that there aren't any segments, but that there are infinitely many. On the other end of the spectrum, regression trees combine grouping and grading features, as we shall see a little later. The overall picture is thus somewhat like what is depicted in Figure 1.7. A taxonomy of eight different models discussed in the book is given in Figure 1.8.<sup>7</sup> These models will be discussed in detail in Chapters 4–9.

### 1.3 Features: the workhorses of machine learning

Now that we have seen some more examples of machine learning tasks and models, we turn to the third and final main ingredient. Features determine much of the success of a machine learning application, because a model is only as good as its features. A fea-

<sup>7</sup>The figures have been generated from data explained in Example 1.7 below.

Model	geom	stats	logic	group	grad	disc	real	sup	unsup	multi
Trees	1	0	3	3	0	3	2	3	2	3
Rules	0	0	3	3	1	3	2	3	0	2
naive Bayes	1	3	1	3	1	3	1	3	0	3
kNN	3	1	0	2	2	1	3	3	0	3
Linear Classifier	3	0	0	0	3	1	3	3	0	0
Linear Regression	3	1	0	0	3	0	3	3	0	1
Logistic Regression	3	2	0	0	3	1	3	3	0	0
SVM	2	2	0	0	3	2	3	3	0	0
Kmeans	3	2	0	1	2	1	3	0	3	1
GMM	1	3	0	0	3	1	3	0	3	1
Associations	0	0	3	3	0	3	1	0	3	1

**Table 1.4.** The MLM data set describing properties of machine learning models. Both [Figure 1.7](#) and [Figure 1.8](#) were generated from this data.

ture can be thought of as a kind of measurement that can be easily performed on any instance. Mathematically, they are functions that map from the instance space to some set of feature values called the *domain* of the feature. Since measurements are often numerical, the most common feature domain is the set of real numbers. Other typical feature domains include the set of integers, for instance when the feature counts something, such as the number of occurrences of a particular word; the Booleans, if our feature is a statement that can be true or false for a particular instance, such as ‘this e-mail is addressed to Peter Flach’; and arbitrary finite sets, such as a set of colours, or a set of shapes.

**Example 1.7 (The MLM data set).** Suppose we have a number of learning models that we want to describe in terms of a number of properties:

- ☞ the extent to which the models are geometric, probabilistic or logical;
- ☞ whether they are grouping or grading models;
- ☞ the extent to which they can handle discrete and/or real-valued features;
- ☞ whether they are used in supervised or unsupervised learning; and
- ☞ the extent to which they can handle multi-class problems.

The first two properties could be expressed by discrete features with three and two values, respectively; or if the distinctions are more gradual, each aspect could be rated on some numerical scale. A simple approach would be to measure each property on an integer scale from 0 to 3, as in [Table 1.4](#). This table establishes a data set in which each row represents an instance and each column a feature. For example, according to this (highly simplified) data some models are

purely grouping models (Trees, Associations) or purely grading models (the Linear models, Logistic Regression and GMM), whereas others are more mixed. We can also see that Trees and Rules have very similar values for most of the features, whereas GMM and Associations have mostly different values.

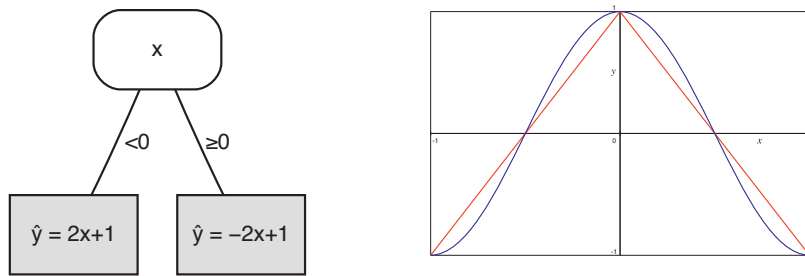
This small data set will be used in several examples throughout the book. In fact, the taxonomy in Figure 1.8 was adapted by hand from a decision tree learned from this small data set, using the models as classes. And the plot in Figure 1.7 was constructed using a dimensionality reduction technique which preserves pairwise distances as much as possible.

### Two uses of features

It is worth noting that features and models are intimately connected, not just because models are defined in terms of features, but because a single feature can be turned into what is sometimes called a *univariate model*. We can therefore distinguish two uses of features that echo the distinction between grouping and grading models. A very common use of features, particularly in logical models, is to zoom in on a particular area of the instance space. Let  $f$  be a feature counting the number of occurrences of the word ‘Viagra’ in an e-mail, and let  $x$  stand for an arbitrary e-mail, then the condition  $f(x) = 0$  selects e-mails that don’t contain the word ‘Viagra’,  $f(x) \neq 0$  or  $f(x) > 0$  selects e-mails that do,  $f(x) \geq 2$  selects e-mails that contain the word at least twice, and so on. Such conditions are called *binary splits*, because they divide the instance space into two groups: those that satisfy the condition, and those that don’t. Non-binary splits are also possible: for instance, if  $g$  is a feature that has the value ‘tweet’ for e-mails with up to 20 words, ‘short’ for e-mails with 21 to 50 words, ‘medium’ for e-mails with 51 to 200 words, and ‘long’ for e-mails with more than 200 words, then the expression  $g(x)$  represents a four-way split of the instance space. As we have already seen, such splits can be combined in a feature tree, from which a model can be built.

A second use of features arises particularly in supervised learning. Recall that a linear classifier employs a decision rule of the form  $\sum_{i=1}^n w_i x_i > t$ , where  $x_i$  is a numerical feature.<sup>8</sup> The linearity of this decision rule means that each feature makes an independent contribution to the score of an instance. This contribution depends on the weight  $w_i$ : if this is large and positive, a positive  $x_i$  increases the score; if  $w_i \ll 0$ , a positive  $x_i$  decreases the score; if  $w_i \approx 0$ ,  $x_i$ ’s influence is negligible. Thus, the feature

<sup>8</sup>Notice we employ two different notations for features: sometimes we write  $f(x)$  if it is more convenient to view a feature as a function applied to instance  $x$ , and sometimes we write  $x_i$  if it is more convenient to view an instance as a vector of feature values.



**Figure 1.9. (left)** A regression tree combining a one-split feature tree with linear regression models in the leaves. Notice how  $x$  is used as both a splitting feature and a regression variable. **(right)** The function  $y = \cos \pi x$  on the interval  $-1 \leq x \leq 1$ , and the piecewise linear approximation achieved by the regression tree.

makes a precise and measurable contribution to the final prediction. Also note that that individual features are not ‘thresholded’, but their full ‘resolution’ is used in computing an instance’s score. These two uses of features – ‘features as splits’ and ‘features as predictors’ – are sometimes combined in a single model.

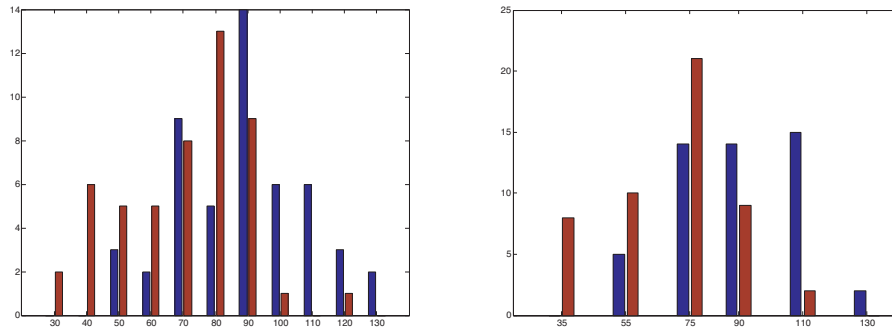
---

**Example 1.8 (Two uses of features).** Suppose we want to approximate  $y = \cos \pi x$  on the interval  $-1 \leq x \leq 1$ . A linear approximation is not much use here, since the best fit would be  $y = 0$ . However, if we split the  $x$ -axis in two intervals  $-1 \leq x < 0$  and  $0 \leq x \leq 1$ , we could find reasonable linear approximations on each interval. We can achieve this by using  $x$  both as a splitting feature and as a regression variable (Figure 1.9).

---

### Feature construction and transformation

There is a lot of scope in machine learning for playing around with features. In the spam filter example, and text classification more generally, the messages or documents don’t come with built-in features; rather, they need to be constructed by the developer of the machine learning application. This *feature construction* process is absolutely crucial for the success of a machine learning application. Indexing an e-mail by the words that occur in it (called a *bag of words* representation as it disregards the order of the words in the e-mail) is a carefully engineered representation that manages to amplify the ‘signal’ and attenuate the ‘noise’ in spam e-mail filtering and related classification tasks. However, it is easy to conceive of problems where this would be exactly

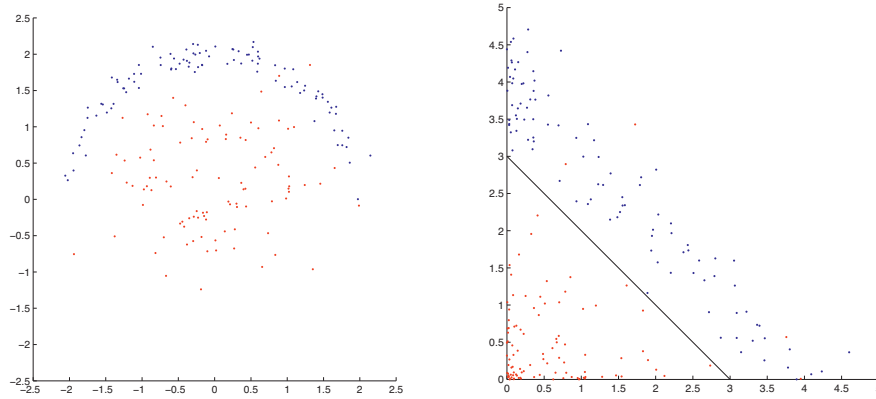


**Figure 1.10. (left)** Artificial data depicting a histogram of body weight measurements of people with (blue) and without (red) diabetes, with eleven fixed intervals of 10 kilograms width each. **(right)** By joining the first and second, third and fourth, fifth and sixth, and the eighth, ninth and tenth intervals, we obtain a discretisation such that the proportion of diabetes cases increases from left to right. This discretisation makes the feature more useful in predicting diabetes.

the wrong thing to do: for instance if we aim to train a classifier to distinguish between grammatical and ungrammatical sentences, word order is clearly signal rather than noise, and a different representation is called for.

It is often natural to build a model in terms of the given features. However, we are free to change the features as we see fit, or even to introduce new features. For instance, real-valued features often contain unnecessary detail that can be removed by *discretisation*. Imagine you want to analyse the body weight of a relatively small group of, say, 100 people, by drawing a histogram. If you measure everybody's weight in kilograms with one position after the decimal point (i.e., your precision is 100 grams), then your histogram will be sparse and spiky. It is hard to draw any general conclusions from such a histogram. It would be much more useful to discretise the body weight measurements into intervals of 10 kilograms. If we are in a classification context, say we're trying to relate body weight to diabetes, we could then associate each bar of the histogram with the proportion of people having diabetes among the people whose weight falls in that interval. In fact, as we shall see in [Chapter 10](#), we can even choose the intervals such that this proportion is monotonically increasing ([Figure 1.10](#)).

The previous example gives another illustration of how, for a particular task such as classification, we can improve the signal-to-noise ratio of a feature. In more extreme cases of feature construction we transform the entire instance space. Consider [Figure 1.11](#): the data on the left is clearly not linearly separable, but by mapping the instance space into a new 'feature space' consisting of the squares of the original features we see that the data becomes almost linearly separable. In fact, by adding in a third feature we can perform a remarkable trick: we can build this feature space classifier without actually constructing the feature space.



**Figure 1.11. (left)** A linear classifier would perform poorly on this data. **(right)** By transforming the original  $(x, y)$  data into  $(x', y') = (x^2, y^2)$ , the data becomes more 'linear', and a linear decision boundary  $x' + y' = 3$  separates the data fairly well. In the original space this corresponds to a circle with radius  $\sqrt{3}$  around the origin.

**Example 1.9 (The kernel trick).** Let  $\mathbf{x}_1 = (x_1, y_1)$  and  $\mathbf{x}_2 = (x_2, y_2)$  be two data points, and consider the mapping  $(x, y) \mapsto (x^2, y^2, \sqrt{2}xy)$  to a three-dimensional feature space. The points in feature space corresponding to  $\mathbf{x}_1$  and  $\mathbf{x}_2$  are  $\mathbf{x}'_1 = (x_1^2, y_1^2, \sqrt{2}x_1y_1)$  and  $\mathbf{x}'_2 = (x_2^2, y_2^2, \sqrt{2}x_2y_2)$ . The dot product of these two feature vectors is

$$\mathbf{x}'_1 \cdot \mathbf{x}'_2 = x_1^2 x_2^2 + y_1^2 y_2^2 + 2x_1 y_1 x_2 y_2 = (x_1 x_2 + y_1 y_2)^2 = (\mathbf{x}_1 \cdot \mathbf{x}_2)^2$$

That is, by squaring the dot product in the original space we obtain the dot product in the new space *without actually constructing the feature vectors*! A function that calculates the dot product in feature space directly from the vectors in the original space is called a *kernel* – here the kernel is  $\kappa(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1 \cdot \mathbf{x}_2)^2$ .

We can apply this *kernel trick* to the basic linear classifier if we modify the way the decision boundary is calculated. Recall that the basic linear classifier learns a decision boundary  $\mathbf{w} \cdot \mathbf{x} = t$  with  $\mathbf{w} = \mathbf{p} - \mathbf{n}$  being the difference between the mean of the positive examples and the mean of the negative examples. As an example, suppose we have  $\mathbf{n} = (0, 0)$  and  $\mathbf{p} = (0, 1)$ , and let's assume for the sake of argument that the positive mean has been obtained from two training examples  $\mathbf{p}_1 = (-1, 1)$  and  $\mathbf{p}_2 = (1, 1)$ . This means that  $\mathbf{p} = \frac{1}{2}(\mathbf{p}_1 + \mathbf{p}_2)$  and we can rewrite the decision boundary as  $\frac{1}{2}\mathbf{p}_1 \cdot \mathbf{x} + \frac{1}{2}\mathbf{p}_2 \cdot \mathbf{x} - \mathbf{n} \cdot \mathbf{x} = t$ . Applying the kernel trick we obtain the following decision boundary:  $\frac{1}{2}\kappa(\mathbf{p}_1, \mathbf{x}) + \frac{1}{2}\kappa(\mathbf{p}_2, \mathbf{x}) - \kappa(\mathbf{n}, \mathbf{x}) = t$ . Using

the kernel defined earlier we have  $\kappa(\mathbf{p}_1, \mathbf{x}) = (-x + y)^2$ ,  $\kappa(\mathbf{p}_2, \mathbf{x}) = (x + y)^2$  and  $\kappa(\mathbf{n}, \mathbf{x}) = 0$ , from which we derive the decision boundary  $\frac{1}{2}(-x + y)^2 + \frac{1}{2}(x + y)^2 = x^2 + y^2 = t$ , i.e., a circle around the origin with radius  $\sqrt{t}$ . Figure 1.11 illustrates this further for a larger data set.

The key point in this ‘kernelisation’ of the basic linear classifier is that we don’t summarise the training data by the positive and negative means – rather, we keep the training data (here:  $\mathbf{p}_1$ ,  $\mathbf{p}_2$  and  $\mathbf{n}$ ), so that when classifying a new instance we can evaluate the kernel on it paired with each training example. In return for this more elaborate calculation we get the ability to construct much more flexible decision boundaries.

### Interaction between features

One fascinating and multi-faceted aspect of features is that they may interact in various ways. Sometimes such interaction can be exploited, sometimes it can be ignored, and sometimes it poses a challenge. We have already seen an example of feature interaction when we talked about Bayesian spam filtering. Clearly, if we notice the term ‘Viagra’ in an e-mail, we are not really surprised to find that the e-mail also contains the phrase ‘blue pill’. Ignoring this interaction, as the naive Bayes classifier does, means that we are overestimating the amount of information conveyed by observing both phrases in the same e-mail. Whether we can get away with this depends on our task: in spam e-mail classification it turns out not to be a big problem, apart from the fact that we may need to adapt the decision threshold to account for this effect.

We can observe other examples of feature interaction in Table 1.4 on p.39. Consider the features ‘grad’ and ‘real’, which assess the extent to which models are of the grading kind, and the extent to which they can handle real-valued features. You may observe that the values of these two features differ by at most 1 for all but one model. Statisticians say that these features are positively correlated (see Background 1.3). Another pair of positively correlated features is ‘logic’ and ‘disc’, indicating logical models and the ability to handle discrete features. We can also see some negatively correlated features, where the value of one goes up when the other goes down: this holds naturally for ‘split’ and ‘grad’, indicating whether models are primarily grouping or grading models; and also for ‘logic’ and ‘grad’. Finally, pairs of uncorrelated features are ‘unsup’ and ‘multi’, standing for unsupervised models and the ability to handle more than two classes; and ‘disc’ and ‘sup’, the latter of which indicates supervised models.

In classification, features may be differently correlated depending on the class. For instance, it is conceivable that for somebody whose last name is Hilton and who works for the Paris city council, e-mails with just the word ‘Paris’ or just the word ‘Hilton’



**Random variables** describe possible outcomes of a random process. They can be either discrete (e.g., the possible outcomes of rolling a die are  $\{1, 2, 3, 4, 5, 6\}$ ) or continuous (e.g., the possible outcomes of measuring somebody's weight in kilograms). Random variables do not need to range over integer or real numbers, but it does make the mathematics quite a bit simpler so that is what we assume here.

If  $X$  is a discrete random variable with probability distribution  $P(X)$  then the **expected value** of  $X$  is  $\mathbb{E}[X] = \sum_x xP(x)$ . For instance, the expected value of tossing a fair die is  $1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + \dots + 6 \cdot \frac{1}{6} = 3.5$ . Notice that this is not actually a possible outcome. For a continuous random variable we need to replace the sum with an integral, and the probability distribution with a probability density function:  $\mathbb{E}[X] = \int_{-\infty}^{+\infty} xp(x) dx$ . The idea of this rather abstract concept is that if we take a sample  $x_1, \dots, x_n$  of outcomes of the random process, the expected value is what we expect the **sample mean**  $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$  to be – this is the celebrated **law of large numbers** first proved by Jacob Bernoulli in 1713. For this reason the expected value is often called the **population mean**, but it is important to realise that the latter is a theoretical value, while the sample mean is an empirical **estimate** of that theoretical value.

The expectation operator can be applied to functions of random variables. For instance, the (population) **variance** of a discrete random variable is defined as  $\mathbb{E}[(X - \mathbb{E}[X])^2] = \sum_x (x - \mathbb{E}[X])^2 P(x)$  – this measures the spread of the distribution around the expected value. Notice that

$$\mathbb{E}[(X - \mathbb{E}[X])^2] = \sum_x (x - \mathbb{E}[X])^2 P(x) = \mathbb{E}[X^2] - \mathbb{E}[X]^2$$

We can similarly define the **sample variance** as  $\sigma^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$ , which decomposes as  $\frac{1}{n} \sum_{i=1}^n x_i^2 - \bar{x}^2$ . You will sometimes see the sample variance defined as  $\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$ : dividing by  $n-1$  rather than  $n$  results in a slightly larger estimate, which compensates for the fact that we are calculating the spread around the sample mean rather than the population mean.

The (population) **covariance** between two discrete random variables  $X$  and  $Y$  is defined as  $\mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] = \mathbb{E}[X \cdot Y] - \mathbb{E}[X] \cdot \mathbb{E}[Y]$ . The variance of  $X$  is a special case of this, with  $Y = X$ . Unlike the variance, the covariance can be positive as well as negative. Positive covariance means that both variables tend to increase or decrease together; negative covariance means that if one variable increases, the other tends to decrease. If we have a sample of pairs of values of  $X$  and  $Y$ , **sample covariance** is defined as  $\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y}) = \frac{1}{n} \sum_{i=1}^n x_i y_i - \bar{x} \bar{y}$ . By dividing the covariance between  $X$  and  $Y$  by  $\sqrt{\sigma_X^2 \sigma_Y^2}$  we obtain the **correlation coefficient**, which is a number between  $-1$  and  $1$ .

---

**Background 1.3.** Expectations and estimators.

are indicative of ham, whereas e-mails with both terms are indicative of spam. Put differently, within the spam class these features are positively correlated, while within the ham class they are negatively correlated. In such a case, ignoring these interactions will be detrimental for classification performance. In other cases, feature correlations may obscure the true model – we shall see examples of this later in the book. On the other hand, feature correlation sometimes helps us to zoom in on the relevant part of the instance space.


There are other ways in which features can be related. Consider the following three features that can be true or false of a molecular compound:

1. it has a carbon in a six-membered aromatic ring;
2. it has a carbon with a partial charge of  $-0.13$ ;
3. it has a carbon in a six-membered aromatic ring with a partial charge of  $-0.13$ .

We say that the third feature is more *specific* (or less *general*) than the other two, because if the third feature is true, then so are the first and the second. However, the converse does not hold: if both first and second feature are true, the third feature may still be false (because the carbon in the six-membered ring may not be the same as the one with a partial charge of  $-0.13$ ). We can exploit these relationships when searching for features to add to our logical model. For instance, if we find that the third feature is true of a particular negative example that we're trying to exclude, then there is no point in considering the more general first and second features, because they will not help us in excluding the negative either. Similarly, if we find that the first feature is false of a particular positive we're trying to include, there is no point in considering the more specific third feature instead. In other words, these relationships help us to structure our search for predictive features.

## 1.4 Summary and outlook

My goal in this chapter has been to take you on a tour to admire the machine learning landscape, and to raise your interest sufficiently to want to read the rest of the book. Here is a summary of the things we have been looking at.

 Machine learning is about using the right features to build the right models that achieve the right tasks. These tasks include: binary and multi-class classification, regression, clustering and descriptive modelling. Models for the first few of these tasks are learned in a supervised fashion requiring labelled training data. For instance, if you want to train a spam filter using machine learning, you need a training set of e-mails labelled spam and ham. If you want to know how good the model is you also need labelled test data that is distinct from the training

data, as evaluating your model on the data it was trained on will paint too rosy a picture: a test set is needed to expose any overfitting that occurs.

- ☞ Unsupervised learning, on the other hand, works with unlabelled data and so there is no test data as such. For instance, to evaluate a particular partition of data into clusters, one can calculate the average distance from the cluster centre. Other forms of unsupervised learning include learning associations (things that tend to occur together) and identifying hidden variables such as film genres. Overfitting is also a concern in unsupervised learning: for instance, assigning each data point its own cluster will reduce the average distance to the cluster centre to zero, yet is clearly not very useful.
- ☞ On the output side we can distinguish between predictive models whose outputs involve the target variable and descriptive models which identify interesting structure in the data. Often, predictive models are learned in a supervised setting while descriptive models are obtained by unsupervised learning methods, but there are also examples of supervised learning of descriptive models (e.g., subgroup discovery which aims at identifying regions with an unusual class distribution) and unsupervised learning of predictive models (e.g., predictive clustering where the identified clusters are interpreted as classes).
- ☞ We have loosely divided machine learning models into geometric models, probabilistic models and logical models. Geometric models are constructed in Cartesian instance spaces, using geometric concepts such as planes and distances. The prototypical geometric model is the basic linear classifier, which constructs a decision plane orthogonal to the line connecting the positive and negative centres of mass. Probabilistic models view learning as a process of reducing uncertainty using data. For instance, a Bayesian classifier models the posterior distribution  $P(Y|X)$  (or its counterpart, the likelihood function  $P(X|Y)$ ) which tells me the class distribution  $Y$  after observing the feature values  $X$ . Logical models are the most 'declarative' of the three, employing if-then rules built from logical conditions to single out homogeneous areas in instance space.
- ☞ We have also introduced a distinction between grouping and grading models. Grouping models divide the instance space into segments which are determined at training time, and hence have a finite resolution. On each segment, grouping models usually fit a very simple kind of model, such as 'always predict this class'. Grading models fit a more global model, graded by the location of an instance in instance space (typically, but not always, a Cartesian space). Logical models are typical examples of grouping models, while geometric models tend to be grading in nature, although this distinction isn't clear-cut. While this sounds very

abstract at the moment, the distinction will become much clearer when we discuss coverage curves in the next chapter.

- ☞ Last but not least, we have discussed the role of features in machine learning. No model can exist without features, and sometimes a single feature is enough to build a model. Data doesn't always come with ready-made features, and often we have to transform or even construct features. Because of this, machine learning is often an iterative process: we only know we have captured the right features after we have constructed the model, and if the model doesn't perform satisfactorily we need to analyse its performance to understand in what way the features need to be improved.

### *What you'll find in the rest of the book*

In the next nine chapters, we will follow the structure laid out above, and look in detail at

- ☞ machine learning tasks in [Chapters 2 and 3](#);
- ☞ logical models: concept learning in [Chapter 4](#), tree models in [Chapter 5](#) and rule models in [Chapter 6](#);
- ☞ geometric models: linear models in [Chapter 7](#) and distance-based models in [Chapter 8](#);
- ☞ probabilistic models in [Chapter 9](#); and
- ☞ features in [Chapter 10](#).

[Chapter 11](#) is devoted to techniques for training 'ensembles' of models that have certain advantages over single models. In [Chapter 12](#) we will consider a number of methods for what machine learners call 'experiments', which involve training and evaluating models on real data. Finally, in the [Epilogue](#) we will wrap up the book and take a look ahead.



---

## Binary classification and related tasks

---

**I**N THIS CHAPTER and the next we take a bird's-eye view of the wide range of different tasks that can be solved with machine learning techniques. 'Task' here refers to whatever it is that machine learning is intended to improve performance of (recall the definition of machine learning on p.3), for example, e-mail spam recognition. Since this is a classification task, we need to learn an appropriate classifier from training data. Many different types of classifiers exist: linear classifiers, Bayesian classifiers, distance-based classifiers, to name a few. We will refer to these different types as models; they are the subject of Chapters 4–9. Classification is just one of a range of possible tasks for which we can learn a model: other tasks that will pass the review in this chapter are class probability estimation and ranking. In the next chapter we will discuss regression, clustering and descriptive modelling. For each of these tasks we will discuss what it is, what variants exist, how performance at the task could be assessed, and how it relates to other tasks. We will start with some general notation that is used in this chapter and throughout the book (see Background 2.1 for the relevant mathematical concepts).

The objects of interest in machine learning are usually referred to as *instances*. The set of all possible instances is called the *instance space*, denoted  $\mathcal{X}$  in this book. To illustrate,  $\mathcal{X}$  could be the set of all possible e-mails that can be written using the Latin

alphabet.<sup>1</sup> We furthermore distinguish between the *label space*  $\mathcal{L}$  and the *output space*  $\mathcal{Y}$ . The label space is used in supervised learning to label the examples. In order to achieve the task under consideration we need a *model*: a mapping from the instance space to the output space. For instance, in classification the output space is a set of classes, while in regression it is the set of real numbers. In order to learn such a model we require a *training set*  $Tr$  of *labelled instances*  $(x, l(x))$ , also called *examples*, where  $l: \mathcal{X} \rightarrow \mathcal{L}$  is a labelling function.

Based on this terminology and notation, and concentrating on supervised learning of predictive models for the duration of the chapter, Table 2.1 distinguishes a number of specific scenarios. The most commonly encountered machine learning scenario is where the label space coincides with the output space. That is,  $\mathcal{Y} = \mathcal{L}$  and we are trying to learn an approximation  $\hat{l}: \mathcal{X} \rightarrow \mathcal{L}$  to the true labelling function  $l$ , which is only known through the labels it assigned to the training data. This scenario covers both classification and regression. In cases where the label space and the output space differ, this usually serves the purpose of learning a model that outputs more than just a label – for instance, a score for each possible label. In this case we have  $\mathcal{Y} = \mathbb{R}^k$ , with  $k = |\mathcal{L}|$  the number of labels.

Matters may be complicated by *noise*, which can take the form of *label noise* – instead of  $l = l(x)$  we observe some corrupted label  $l'$  – or *instance noise* – instead of  $x$  we observe an instance  $x'$  that is corrupted in some way. One consequence of noisy data is that it is generally not advisable to try to match the training data exactly, as this may lead to overfitting the noise. Some of the labelled data is usually set aside for evaluating or testing a classifier, in which case it is called a *test set* and denoted by  $Te$ . We use superscripts to restrict training or test set to a particular class: e.g.,  $Te^{\oplus} = \{(x, l(x)) | x \in Te, l(x) = \oplus\}$  is the set of positive test examples, and  $Te^{\ominus}$  is the set of negative test examples.

The simplest kind of input space arises when instances are described by a fixed number of *features*, also called attributes, predictor variables, explanatory variables or independent variables. Indicating the set of values or *domain* of a feature by  $\mathcal{F}_i$ , we then have that  $\mathcal{X} = \mathcal{F}_1 \times \mathcal{F}_2 \times \dots \times \mathcal{F}_d$ , and thus every instance is a  $d$ -vector of feature values. In some domains the features to use readily suggest themselves, whereas in other domains they need to be constructed. For example, in the spam filter example in the *Prologue* we constructed a large number of features, one for each word in a vocabulary, counting the number of occurrences of that word in the e-mail. Even when features are given explicitly we often want to transform them to maximise their usefulness for the task at hand. We will discuss this in considerable detail in Chapter 10.

<sup>1</sup> It is perhaps worth emphasising that an instance space like this is an unimaginably vast set (e.g., the set of all possible text messages of 160 characters using only lower-case letters, spaces and full stops is  $28^{160}$ , a number too large for most pocket calculators), and that only a minuscule fraction of this set carries enough meaning to be possibly encountered in the real world.

We briefly review some important concepts from discrete mathematics. A *set* is a collection of objects, usually of the same kind (e.g., the set of all natural numbers  $\mathbb{N}$  or the set of real numbers  $\mathbb{R}$ ). We write  $x \in A$  if  $x$  is an element of set  $A$ , and  $A \subseteq B$  if all elements of  $A$  are also elements of  $B$  (this includes the possibility that  $A$  and  $B$  are the same set, which is equivalent to  $A \subseteq B$  and  $B \subseteq A$ ). The *intersection* and *union* of two sets are defined as  $A \cap B = \{x | x \in A \text{ and } x \in B\}$  and  $A \cup B = \{x | x \in A \text{ or } x \in B\}$ . The *difference* of two sets is defined as  $A \setminus B = \{x | x \in A \text{ and } x \notin B\}$ . It is customary to fix a *universe of discourse*  $U$  such that all sets under consideration are subsets of  $U$ . The *complement* of a set  $A$  is defined as  $\bar{A} = U \setminus A$ . Two sets are *disjoint* if their intersection is empty:  $A \cap B = \emptyset$ . The *cardinality* of a set  $A$  is its number of elements and is denoted  $|A|$ . The *powerset* of a set  $A$  is the set of all its subsets  $2^A = \{B | B \subseteq A\}$ ; its cardinality is  $|2^A| = 2^{|A|}$ . The *characteristic function* of a set  $A$  is the function  $f : U \rightarrow \{\text{true}, \text{false}\}$  such that  $f(x) = \text{true}$  if  $x \in A$  and  $f(x) = \text{false}$  if  $x \in U \setminus A$ .

If  $A$  and  $B$  are sets, the *Cartesian product*  $A \times B$  is the set of all pairs  $\{(x, y) | x \in A \text{ and } y \in B\}$ ; this generalises to products of more than two sets. A (binary) *relation* is a set of pairs  $R \subseteq A \times B$  for some sets  $A$  and  $B$ ; if  $A = B$  we say the relation is over  $A$ . Instead of  $(x, y) \in R$  we also write  $xRy$ . A relation over  $A$  is (i) *reflexive* if  $xRx$  for all  $x \in A$ ; (ii) *symmetric* if  $xRy$  implies  $yRx$  for all  $x, y \in A$ ; (iii) *antisymmetric* if  $xRy$  and  $yRx$  implies  $x = y$  for all  $x, y \in A$ ; (iv) *transitive* if  $xRy$  and  $yRz$  implies  $xRz$  for all  $x, y, z \in A$ . (v) *total* if  $xRy$  or  $yRx$  for all  $x, y \in A$ .

A *partial order* is a binary relation that is reflexive, antisymmetric and transitive. For instance, the *subset* relation  $\subseteq$  is a partial order. A *total order* is a binary relation that is total (hence reflexive), antisymmetric and transitive. The  $\leq$  relation on real numbers is a total order. If  $xRy$  or  $yRx$  we say that  $x$  and  $y$  are *comparable*; otherwise they are *incomparable*. An *equivalence relation* is a binary relation  $\equiv$  that is reflexive, symmetric and transitive. The *equivalence class* of  $x$  is  $[x] = \{y | x \equiv y\}$ . For example, the binary relation ‘contains the same number of elements as’ over any set is an equivalence relation. Any two equivalence classes are disjoint, and the union of all equivalence classes is the whole set – in other words, the set of all equivalence classes forms a *partition* of the set. If  $A_1, \dots, A_n$  is a partition of a set  $A$ , i.e.  $A_1 \cup \dots \cup A_n = A$  and  $A_i \cap A_j = \emptyset$  for all  $i \neq j$ , we write  $A = A_1 \sqcup \dots \sqcup A_n$ .

To illustrate this, let  $T$  be a feature tree, and define a relation  $\sim_T \subseteq \mathcal{X} \times \mathcal{X}$  such that  $x \sim_T x'$  if and only if  $x$  and  $x'$  are assigned to the same leaf of feature tree  $T$ , then  $\sim_T$  is an equivalence relation, and its equivalence classes are precisely the instance space segments associated with  $T$ .

---

**Background 2.1.** Useful concepts from discrete mathematics.

The sections in this chapter are devoted to the first three scenarios in Table 2.1:

Task	Label space	Output space	Learning problem
Classification	$\mathcal{L} = \mathcal{C}$	$\mathcal{Y} = \mathcal{C}$	learn an approximation $\hat{c} : \mathcal{X} \rightarrow \mathcal{C}$ to the true labelling function $c$
Scoring and ranking	$\mathcal{L} = \mathcal{C}$	$\mathcal{Y} = \mathbb{R}^{ \mathcal{C} }$	learn a model that outputs a score vector over classes
Probability estimation	$\mathcal{L} = \mathcal{C}$	$\mathcal{Y} = [0, 1]^{ \mathcal{C} }$	learn a model that outputs a probability vector over classes
Regression	$\mathcal{L} = \mathbb{R}$	$\mathcal{Y} = \mathbb{R}$	learn an approximation $\hat{f} : \mathcal{X} \rightarrow \mathbb{R}$ to the true labelling function $f$

**Table 2.1.** Predictive machine learning scenarios.

classification in [Section 2.1](#), scoring and ranking in [Section 2.2](#) and class probability estimation in [Section 2.3](#). To keep things manageable we mostly restrict attention to two-class tasks in this chapter and deal with more than two classes in [Chapter 3](#). Regression, unsupervised and descriptive learning will also be considered there.

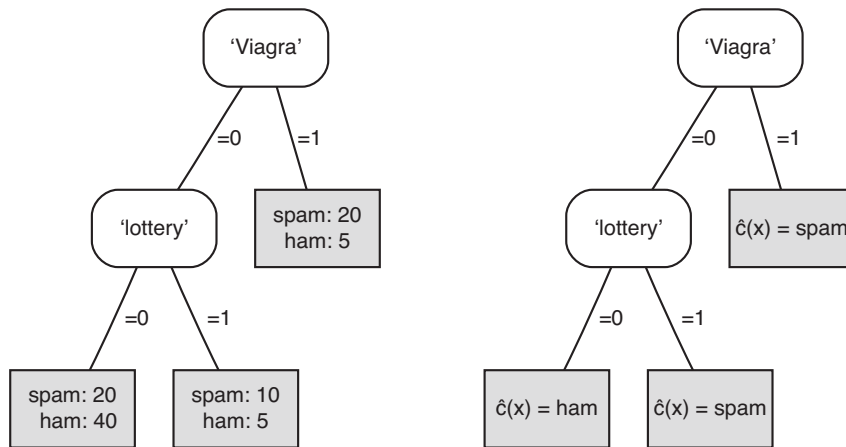
Throughout this chapter I will illustrate key concepts by means of examples using simple models of the kind discussed in the [Prologue](#). These models will either be simple tree-based models, representative of grouping models, or linear models, representative of grading models. Sometimes we will even construct models from single features, a setting that could be described as *univariate machine learning*. We will start dealing with the question of how to *learn* such models from [Chapter 4](#) onwards.

## 2.1 Classification

Classification is the most common task in machine learning. A *classifier* is a mapping  $\hat{c} : \mathcal{X} \rightarrow \mathcal{C}$ , where  $\mathcal{C} = \{C_1, C_2, \dots, C_k\}$  is a finite and usually small set of *class labels*. We will sometimes also use  $C_i$  to indicate the set of examples of that class. We use the ‘hat’ to indicate that  $\hat{c}(x)$  is an estimate of the true but unknown function  $c(x)$ . Examples for a classifier take the form  $(x, c(x))$ , where  $x \in \mathcal{X}$  is an instance and  $c(x)$  is the true class of the instance. Learning a classifier involves constructing the function  $\hat{c}$  such that it matches  $c$  as closely as possible (and not just on the training set, but ideally on the entire instance space  $\mathcal{X}$ ).

In the simplest case we have only two classes which are usually referred to as *positive* and *negative*,  $\oplus$  and  $\ominus$ , or  $+1$  and  $-1$ . Two-class classification is often called *binary classification* (or *concept learning*, if the positive class can be meaningfully called





**Figure 2.1.** (left) A feature tree with training set class distribution in the leaves. (right) A decision tree obtained using the majority class decision rule.

a concept). Spam e-mail filtering is a good example of binary classification, in which spam is conventionally taken as the positive class, and ham as the negative class (clearly, positive here doesn't mean 'good!'). Other examples of binary classification include medical diagnosis (the positive class here is having a particular disease) and credit card fraud detection.

The feature tree in Figure 2.1 (left) can be turned into a classifier by labelling each leaf with a class. The simplest way to do this is by assigning the *majority class* in each leaf, resulting in the decision tree in Figure 2.1 (right). The classifier works as follows: if an e-mail contains the word 'Viagra' it is classified as spam (right-most leaf); otherwise, the occurrence of the word 'lottery' decides whether it gets labelled spam or ham.<sup>2</sup> From the numbers in Figure 2.1 we can get an idea how well this classifier does. The left-most leaf correctly predicts 40 ham e-mails but also mislabels 20 spam e-mails that contain neither 'Viagra' nor 'lottery'. The middle leaf correctly classifies 10 spam e-mails but also erroneously labels 5 ham e-mails as spam. The 'Viagra' test correctly picks out 20 spam e-mails but also 5 ham e-mails. Taken together, this means that 30 out of 50 spam e-mails are classified correctly, and 40 out of 50 ham e-mails.

### Assessing classification performance

The performance of such classifiers can be summarised by means of a table known as a *contingency table* or *confusion matrix* (Table 2.2 (left)). In this table, each row refers to

<sup>2</sup>If you are keen to know how such a decision tree can be learned from data, you may want to take a sneak preview at Algorithm 5.1 on p.132.

actual classes as recorded in the test set, and each column to classes as predicted by the classifier. So, for instance, the first row states that the test set contains 50 positives, 30 of which were correctly predicted and 20 incorrectly. The last column and the last row give the *marginals* (i.e., column and row sums). Marginals are important because they allow us to assess statistical significance. For instance, the contingency table in Table 2.2 (right) has the same marginals, but the classifier clearly makes a random choice as to which predictions are positive and which are negative – as a result the distribution of actual positives and negatives in either predicted class is the same as the overall distribution (uniform in this case).

	Predicted $\oplus$	Predicted $\ominus$	
Actual $\oplus$	30	20	50
Actual $\ominus$	10	40	50
	40	60	100

	$\oplus$	$\ominus$	
$\oplus$	20	30	50
$\ominus$	20	30	50
	40	60	100

**Table 2.2. (left)** A two-class contingency table or confusion matrix depicting the performance of the decision tree in Figure 2.1. Numbers on the descending diagonal indicate correct predictions, while the ascending diagonal concerns prediction errors. **(right)** A contingency table with the same marginals but independent rows and columns.

From a contingency table we can calculate a range of performance indicators. The simplest of these is *accuracy*, which is the proportion of correctly classified test instances. In the notation introduced at the beginning of this chapter, accuracy over a test set  $Te$  is defined as

$$acc = \frac{1}{|Te|} \sum_{x \in Te} I[\hat{c}(x) = c(x)] \quad (2.1)$$

Here, the function  $I[\cdot]$  denotes the *indicator function*, which is 1 if its argument evaluates to true, and 0 otherwise. In this case it is a convenient way to count the number of test instances that are classified correctly by the classifier (i.e., the estimated class label  $\hat{c}(x)$  is equal to the true class label  $c(x)$ ). For example, in Table 2.2 (left) the accuracy of the classifier is 0.70 or 70%, and in Table 2.2 (right) it is 0.50. Alternatively, we can calculate the *error rate* as the proportion of incorrectly classified instances, here 0.30 and 0.50, respectively. Clearly, accuracy and error rate sum to 1.

Test set accuracy can be seen as an *estimate* of the probability that an arbitrary instance  $x \in \mathcal{X}$  is classified correctly: more precisely, it estimates the probability

$$P_{\mathcal{X}}(\hat{c}(x) = c(x))$$

(Notice that I write  $P_{\mathcal{X}}$  to emphasise that this is a probability distribution over the instance space  $\mathcal{X}$ ; I will often omit subscripts if this is clear from the context.) We

typically only have access to the true classes of a small fraction of the instance space and so an estimate is all we can hope to get. It is therefore important that the test set is as representative as possible. This is usually formalised by the assumption that the occurrence of instances in the world – i.e., how likely or typical a particular e-mail is – is governed by an unknown probability distribution on  $\mathcal{X}$ , and that the test set  $Te$  is generated according to this distribution.

It is often convenient – not to say necessary – to distinguish performance on the classes. To this end, we need some further terminology. Correctly classified positives and negatives are referred to as *true positives* and *true negatives*, respectively. Incorrectly classified positives are, perhaps somewhat confusingly, called *false negatives*; similarly, misclassified negatives are called *false positives*. A good way to think of this is to remember that positive/negative refers to the classifier's prediction, and true/false refers to whether the prediction is correct or not. So, a false positive is something that was incorrectly predicted as positive, and therefore an actual negative (e.g., a ham e-mail misclassified as spam, or a healthy patient misclassified as having the disease in question). In the previous example (Table 2.2 (left)) we have 30 true positives, 20 false negatives, 40 true negatives and 10 false positives.

The *true positive rate* is the proportion of positives correctly classified, and can be defined mathematically as

$$tpr = \frac{\sum_{x \in Te} I[\hat{c}(x) = c(x) = \oplus]}{\sum_{x \in Te} I[c(x) = \oplus]} \quad (2.2)$$

True positive rate is an estimate of the probability that an arbitrary positive is classified correctly, that is, an estimate of  $P_{\mathcal{X}}(\hat{c}(x) = \oplus | c(x) = \oplus)$ . Analogously, the *true negative rate* is the proportion of negatives correctly classified (see Table 2.3 on p.57 for the mathematical definition), and estimates  $P_{\mathcal{X}}(\hat{c}(x) = \ominus | c(x) = \ominus)$ . These rates, which are sometimes called *sensitivity* and *specificity*, can be seen as per-class accuracies. In the contingency table, the true positive and negative rates can be calculated by dividing the number on the descending (good) diagonal by the row total. We can also talk about per-class error rates, which is the *false negative rate* for the positives (i.e., the number of misclassified positives or false negatives as a proportion of the total number of positives) and the *false positive rate* for the negatives (sometimes called the *false alarm rate*). These rates can be found by dividing the number on the ascending (bad) diagonal by the row total.

In Table 2.2 (left) we have a true positive rate of 60%, a true negative rate of 80%, a false negative rate of 40% and a false positive rate of 20%. In Table 2.2 (right) we have a true positive rate of 40%, a true negative rate of 60%, a false negative rate of 60% and a false positive rate of 40%. Notice that the accuracy in both cases is the average of the true positive rate and the true negative rate (and the error rate is the average of the false positive rate and the false negative rate). However, this is true only if the test set

contains equal numbers of positives and negatives – in the general case we need to use a *weighted* average, where the weights are the proportions of positives and negatives in the test set.

**Example 2.1 (Accuracy as a weighted average).** Suppose a classifier's predictions on a test set are as in the following table:

	Predicted $\oplus$	Predicted $\ominus$	
Actual $\oplus$	60	15	75
Actual $\ominus$	10	15	25
	70	30	100

From this table, we see that the true positive rate is  $tpr = 60/75 = 0.80$  and the true negative rate is  $tnr = 15/25 = 0.60$ . The overall accuracy is  $acc = (60 + 15)/100 = 0.75$ , which is no longer the average of true positive and negative rates. However, taking into account the proportion of positives  $pos = 0.75$  and the proportion of negatives  $neg = 1 - pos = 0.25$ , we see that

$$acc = pos \cdot tpr + neg \cdot tnr \quad (2.3)$$

This equation holds in general: if the numbers of positives and negatives are equal, we obtain the unweighted average from the earlier example ( $acc = (tpr + tnr)/2$ ).

Equation 2.3 has a neat intuition: good performance on either class contributes to good classification accuracy, but the more prevalent class contributes more strongly. In order to achieve good accuracy, a classifier should concentrate on the *majority class*, particularly if the class distribution is highly unbalanced. However, it is often the case that the majority class is also the least interesting class. To illustrate, suppose you issue a query to an internet search engine,<sup>3</sup> and suppose that for that particular query there is only one relevant page in every 1 000 web pages. Now consider a 'reluctant' search engine that doesn't return *any* answers – i.e., it classifies every web page as irrelevant to your query. Consequently, it will achieve 0% true positive rate and 100% true negative rate. Because  $pos = 1/1000 = 0.1\%$  and  $neg = 99.9\%$ , the reluctant search engine's accuracy is very high (99.9%). Put differently, if we select a random web page uniformly

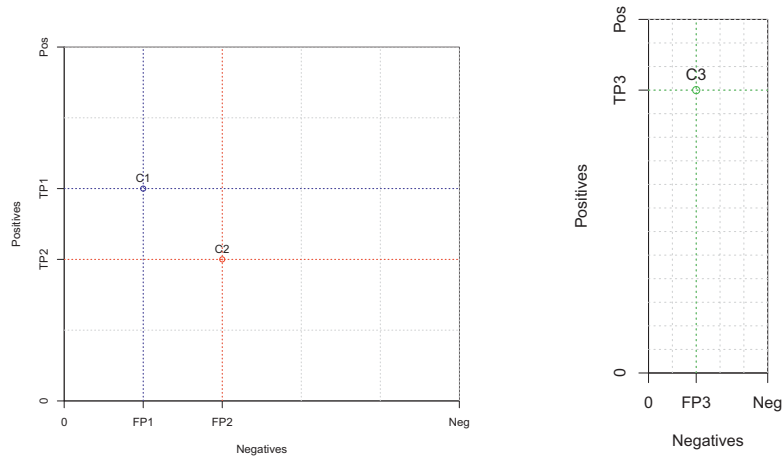
<sup>3</sup>An internet search engine can be seen as a binary classifier into the classes relevant and irrelevant, or interesting and not interesting, if we fix the query – not very realistic in practice, but a useful analogy for our purposes.

Measure	Definition	Equal to	Estimates
number of positives	$Pos = \sum_{x \in Te} I[c(x) = \oplus]$		
number of negatives	$Neg = \sum_{x \in Te} I[c(x) = \ominus]$	$ Te  - Pos$	
number of true positives	$TP = \sum_{x \in Te} I[\hat{c}(x) = c(x) = \oplus]$		
number of true negatives	$TN = \sum_{x \in Te} I[\hat{c}(x) = c(x) = \ominus]$		
number of false positives	$FP = \sum_{x \in Te} I[\hat{c}(x) = \oplus, c(x) = \ominus]$	$Neg - TN$	
number of false negatives	$FN = \sum_{x \in Te} I[\hat{c}(x) = \ominus, c(x) = \oplus]$	$Pos - TP$	
proportion of positives	$pos = \frac{1}{ Te } \sum_{x \in Te} I[c(x) = \oplus]$	$Pos/ Te $	$P(c(x) = \oplus)$
proportion of negatives	$neg = \frac{1}{ Te } \sum_{x \in Te} I[c(x) = \ominus]$	$1 - pos$	$P(c(x) = \ominus)$
class ratio	$clr = pos/neg$	$Pos/Neg$	
(*) accuracy	$acc = \frac{1}{ Te } \sum_{x \in Te} I[\hat{c}(x) = c(x)]$		$P(\hat{c}(x) = c(x))$
(*) error rate	$err = \frac{1}{ Te } \sum_{x \in Te} I[\hat{c}(x) \neq c(x)]$	$1 - acc$	$P(\hat{c}(x) \neq c(x))$
true positive rate, sensitivity, recall	$tpr = \frac{\sum_{x \in Te} I[\hat{c}(x)=c(x)=\oplus]}{\sum_{x \in Te} I[c(x)=\oplus]}$	$TP/Pos$	$P(\hat{c}(x) = \oplus   c(x) = \oplus)$
true negative rate, specificity, negative recall	$tnr = \frac{\sum_{x \in Te} I[\hat{c}(x)=c(x)=\ominus]}{\sum_{x \in Te} I[c(x)=\ominus]}$	$TN/Neg$	$P(\hat{c}(x) = \ominus   c(x) = \ominus)$
false positive rate, false alarm rate	$fpr = \frac{\sum_{x \in Te} I[\hat{c}(x)=\oplus, c(x)=\ominus]}{\sum_{x \in Te} I[c(x)=\ominus]}$	$FP/Neg = 1 - tnr$	$P(\hat{c}(x) = \oplus   c(x) = \ominus)$
false negative rate	$fnr = \frac{\sum_{x \in Te} I[\hat{c}(x)=\ominus, c(x)=\oplus]}{\sum_{x \in Te} I[c(x)=\oplus]}$	$FN/Pos = 1 - tpr$	$P(\hat{c}(x) = \ominus   c(x) = \oplus)$
precision, confidence	$prec = \frac{\sum_{x \in Te} I[\hat{c}(x)=c(x)=\oplus]}{\sum_{x \in Te} I[\hat{c}(x)=\oplus]}$	$TP/(TP + FP)$	$P(c(x) = \oplus   \hat{c}(x) = \oplus)$

**Table 2.3.** A summary of different quantities and evaluation measures for classifiers on a test set  $Te$ . Symbols starting with a capital letter denote absolute frequencies (counts), while lower-case symbols denote relative frequencies or ratios. All except those indicated with (\*) are defined only for binary classification. The right-most column specifies the instance space probabilities that these relative frequencies are estimating.

over all web pages, the probability of selecting a positive is only 0.001, and these are the only pages on which the reluctant engine makes an error. However, we are not normally selecting pages from the web uniformly, and hence accuracy is not a meaningful quantity in this context. To be of any use at all, a search engine should achieve a much better true positive rate, which usually comes at the expense of a worse true negative rate (and hence a drop in accuracy).

We conclude from this example that, if the minority class is the class of interest and very small, accuracy and performance on the majority class are not the right quantities to optimise. For this reason, an alternative to true negative rate called *precision* is usually considered in such cases. Precision is a counterpart to true positive rate in the following sense: while true positive rate is the proportion of predicted positives among the actual positives, precision is the proportion of actual positives among the predicted positives. In [Example 2.1](#) the classifier's precision on the test set is  $60/70 = 85.7\%$ . In



**Figure 2.2.** (left) A coverage plot depicting the two contingency tables in Table 2.2. The plot is square because the class distribution is uniform. (right) Coverage plot for Example 2.1, with a class ratio  $clr = 3$ .

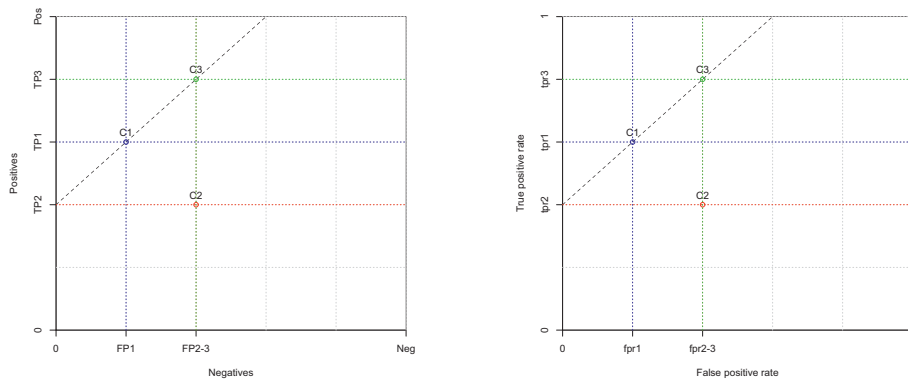
the reluctant search engine example we have not only 0 true positive rate (which in this context is usually called *recall*) but also 0 precision, which clearly demonstrates the problem with a search engine that doesn't return any answers. Table 2.3 summarises the evaluation measures introduced in this section.

### Visualising classification performance

I will now introduce an important tool for visualising the performance of classifiers and other models called a *coverage plot*. If you look at two-class contingency tables such as the ones depicted in Table 2.2, you realise that, even though the table contains nine numbers, only four of those can be chosen freely. For instance, once you've determined the true/false positives/negatives, the marginals are fixed. Or if you know the true positives, true negatives, total number of positives and size of the test set, you can reconstruct all other numbers. Statisticians say that the table has four *degrees of freedom*.<sup>4</sup>

Often we are particularly interested in the following four numbers that completely determine the contingency table: the number of positives *Pos*, the number of negatives *Neg*, the number of true positives *TP* and the number of false positives *FP*. A coverage plot visualises these four numbers by means of a rectangular coordinate system and a point. Imagine a rectangle with height *Pos* and width *Neg*. Imagine furthermore that all positives live on the *y*-axis of this rectangle, and all negatives on the *x*-axis. We don't

<sup>4</sup>More generally, a  $k$ -class contingency table has  $(k+1)^2$  entries and  $k^2$  degrees of freedom.



**Figure 2.3. (left)** C1 and C3 both dominate C2, but neither dominates the other. The diagonal line indicates that C1 and C3 achieve equal accuracy. **(right)** The same plot with normalised axes. We can interpret this plot as a merger of the two coverage plots in Figure 2.2, employing normalisation to deal with the different class distributions. The diagonal line now indicates that C1 and C3 have the same average recall.

really care how positives and negatives are ordered on their respective axes, as long as *positive predictions come before negative predictions*. This gives us enough information to depict the whole contingency table as a single point within the rectangle (Figure 2.2).

Consider the two classifiers marked C1 and C2 in Figure 2.2 (left). One reason why coverage plots are so useful is that we can immediately see that C1 is better than C2. How do we know that? Well, C1 has both more true positives and fewer false positives than C2, and so is better in both respects. Put differently, C1 achieves better performance than C2 on *both* classes. If one classifier outperforms another classifier on all classes, the first one is said to *dominate* the second.<sup>5</sup> However, things are not always that straightforward. Consider a third classifier C3, better than C1 on the positives but worse on the negatives (Figure 2.3 (left)). Although both C1 and C3 dominate C2, neither of them dominates the other. Which one we prefer depends on whether we put more emphasis on the positives or on the negatives.

We can make this a little bit more precise. Notice that the line segment connecting C1 and C3 has a slope of 1. Imagine travelling up that line: whenever we gain a true positive, we also lose a true negative (or gain a false positive, which is the same thing). This doesn't affect the sum of true positives and true negatives, and hence the accuracy is the same wherever we are on the line. It follows that C1 and C3 have the same accuracy. *In a coverage plot, classifiers with the same accuracy are connected by line segments with slope 1.* If true positives and true negatives are equally important, the

<sup>5</sup>This terminology comes from the field of *multi-criterion optimisation*. A dominated solution is one that is not on the *Pareto front*.

choice between C1 and C3 is arbitrary; if true positives are more important we should choose C3, if true negatives are more important we prefer C1.

Now consider [Figure 2.3 \(right\)](#). What I have done here is renormalise the axes by dividing the  $x$ -axis by  $Neg$  and the  $y$ -axis by  $Pos$ , resulting in a plot in the unit square with true positive rate on the  $y$ -axis and false positive rate on the  $x$ -axis. In this case the original coverage plot was already square ( $Pos = Neg$ ), so the relative position of the classifiers isn't affected by the normalisation. However, since the normalised plot will be square regardless of the shape of the original plot, normalisation is a way to combine differently shaped coverage plots, and thus to combine results on test sets with different class distributions. Suppose you would normalise [Figure 2.2 \(right\)](#): since C3's true and false positive rates are 80% and 40%, respectively (see [Example 2.1](#) on p.56), its position in a normalised plot is exactly the same as the one labelled C3 in [Figure 2.3 \(right\)](#)! In other words, classifiers occupying different points in different coverage spaces (e.g., C3 in [Figure 2.2 \(right\)](#) and C3 in [Figure 2.3 \(left\)](#)) can end up in the same point in a normalised plot.

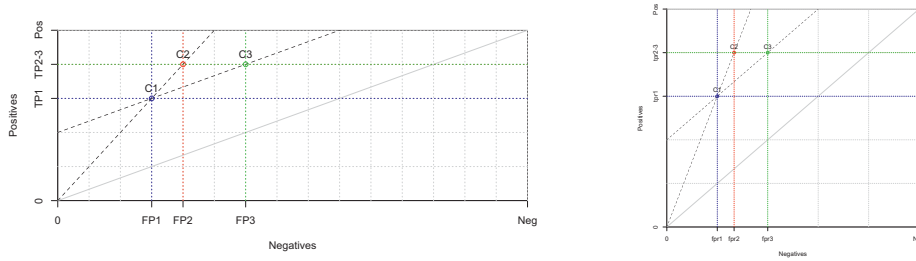
What is the meaning of the diagonal line connecting C1 and C3 in [Figure 2.3 \(right\)](#)? It can't have the same meaning as in the coverage plot, because in a normalised plot we know the true and false positive rates but not the class distribution, and so we cannot calculate accuracy (refer back to [Equation 2.3](#) on p.56 if you want to remind yourself why). The line is defined by the equation  $tpr = fpr + y_0$ , where  $y_0$  is the  $y$ -intercept (the value of  $tpr$  where the line intersects the  $y$ -axis). Now consider the average of the true positive rate and the true negative rate, which we will call *average recall*, denoted *avg-rec*.<sup>6</sup> On a line with slope 1 we have  $avg-rec = (tpr + tnr)/2 = (tpr + 1 - fpr)/2 = (1 + y_0)/2$ , which is a constant. *In a normalised coverage plot, line segments with slope 1 connect classifiers with the same average recall.* If recall on the positives and the negatives are equally important, the choice between C1 and C3 is arbitrary; if positive recall is more important we should choose C3, if negative recall is more important we prefer C1.

In the literature, normalised coverage plots are referred to as *ROC plots*, and we will follow that convention from now on.<sup>7</sup> ROC plots are much more common than coverage plots, but both have their specific uses. Broadly speaking, you should use a coverage plot if you explicitly want to take the class distribution into account, for instance when you are working with a single data set. An ROC plot is useful if you want to combine results from different data sets with different class distributions. Clearly, there are many connections between the two. Since an ROC plot is always square, lines of constant average recall (so-called average recall *isometrics*) do not only have

<sup>6</sup>Remember that recall is just a different name for true positive rate; negative recall is then the same as the true negative rate, and average recall is the average of positive recall (or true positive rate) and negative recall (or true negative rate). It is sometimes called *macro-averaged accuracy*.

<sup>7</sup>ROC stands for *receiver operating characteristic*, a term originating from *signal detection theory*.





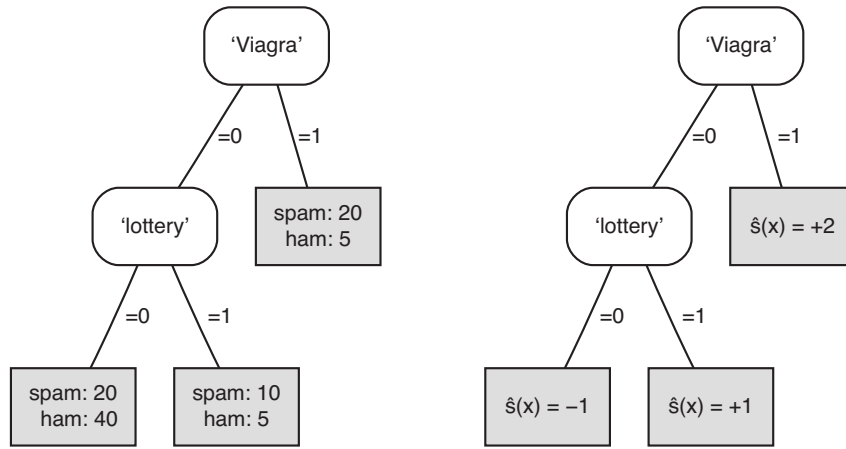
**Figure 2.4.** (left) In a coverage plot, accuracy isometrics have a slope of 1, and average recall isometrics are parallel to the ascending diagonal. (right) In the corresponding ROC plot, average recall isometrics have a slope of 1; the accuracy isometric here has a slope of 3, corresponding to the ratio of negatives to positives in the data set.

a slope of 1 but are parallel to the ascending diagonal. The latter property carries over to coverage plots. To illustrate, in the coverage plot in Figure 2.4, C1 and C2 have the same accuracy (they are connected by a line segment with slope 1), and C1 and C3 have the same average recall (they are connected by a line segment parallel to the diagonal). You can also argue that C2 has both higher accuracy and higher average recall than C3 (why?). In the corresponding ROC plot, the average recall isometric has a slope of 1, and the accuracy isometric's slope is  $Neg/Pos = 1/cfr$ .

## 2.2 Scoring and ranking

Many classifiers compute scores on which their class predictions are based. For instance, in the [Prologue](#) we saw how SpamAssassin calculates a weighted sum from the rules that ‘fire’ for a particular e-mail. Such scores contain additional information that can be beneficial in a number of ways, which is why we perceive scoring as a task in its own right. Formally, a *scoring classifier* is a mapping  $\hat{\mathbf{s}}: \mathcal{X} \rightarrow \mathbb{R}^k$ , i.e., a mapping from the instance space to a  $k$ -vector of real numbers. The boldface notation indicates that a scoring classifier outputs a vector  $\hat{\mathbf{s}}(x) = (\hat{s}_1(x), \dots, \hat{s}_k(x))$  rather than a single number;  $\hat{s}_i(x)$  is the score assigned to class  $C_i$  for instance  $x$ . This score indicates how likely it is that class label  $C_i$  applies. If we only have two classes, it usually suffices to consider the score for only one of the classes; in that case, we use  $\hat{s}(x)$  to denote the score of the positive class for instance  $x$ .

Figure 2.5 demonstrates how a feature tree can be turned into a scoring tree. In order to obtain a score for each leaf, we first calculate the ratio of spam to ham, which is 1/2 for the left leaf, 2 for the middle leaf and 4 for the right leaf. Because it is often more convenient to work with an additive scale, we obtain scores by taking the logarithm of the class ratio (the base of the logarithm is not really important; here we have



**Figure 2.5. (left)** A feature tree with training set class distribution in the leaves. **(right)** A scoring tree using the logarithm of the class ratio as scores; spam is taken as the positive class.

taken base-2 logarithms to get nice round numbers). Notice that the majority class decision tree corresponds to thresholding  $\hat{s}(x)$  at 0: i.e., predict spam if  $\hat{s}(x) > 0$  and ham otherwise.

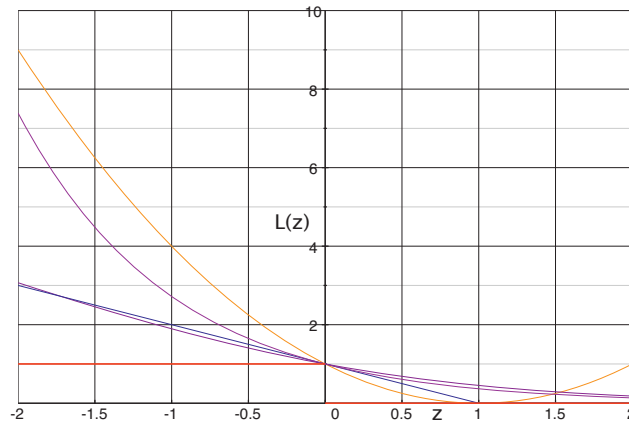
If we take the true class  $c(x)$  as  $+1$  for positive examples and  $-1$  for negative examples, then the quantity  $z(x) = c(x)\hat{s}(x)$  is positive for correct predictions and negative for incorrect predictions: this quantity is called the *margin* assigned by the scoring classifier to the example.<sup>8</sup> We would like to reward large positive margins, and penalise large negative values. This is achieved by means of a so-called *loss function*  $L: \mathbb{R} \rightarrow [0, \infty)$  which maps each example's margin  $z(x)$  to an associated loss  $L(z(x))$ . We will assume that  $L(0) = 1$ , which is the loss incurred by having an example on the decision boundary. We furthermore have  $L(z) \geq 1$  for  $z < 0$ , and usually also  $0 \leq L(z) < 1$  for  $z > 0$  (Figure 2.6). The average loss over a test set  $Te$  is  $\frac{1}{|Te|} \sum_{x \in Te} L(z(x))$ .

The simplest loss function is *0-1 loss*, which is defined as  $L_{01}(z) = 1$  if  $z \leq 0$  and  $L(z) = 0$  if  $z > 0$ . The average 0-1 loss is simply the proportion of misclassified test examples:

$$\frac{1}{|Te|} \sum_{x \in Te} L_{01}(z(x)) = \frac{1}{|Te|} \sum_{x \in Te} I[c(x)\hat{s}(x) \leq 0] = \frac{1}{|Te|} \sum_{x \in Te} I[c(x) \neq \hat{c}(x)] = err$$

where  $\hat{c}(x) = +1$  if  $\hat{s}(x) > 0$ ,  $\hat{c}(x) = 0$  if  $\hat{s}(x) = 0$ , and  $\hat{c}(x) = -1$  if  $\hat{s}(x) < 0$ . (It is sometimes more convenient to define the loss of examples on the decision boundary as  $1/2$ ). In other words, 0-1 loss ignores the magnitude of the margins of the examples, only

<sup>8</sup>Remember that in Chapter 1 we talked about the margin of a classifier as the distance between the decision boundary and the nearest example. Here we use margin in a slightly more general sense: each example has a margin, not just the nearest one. This will be further explained in Section 7.3.



**Figure 2.6.** Loss functions: from bottom-left (i) 0–1 loss  $L_{01}(z) = 1$  if  $z \leq 0$ , and  $L_{01}(z) = 0$  if  $z > 0$ ; (ii) hinge loss  $L_h(z) = (1 - z)$  if  $z \leq 1$ , and  $L_h(z) = 0$  if  $z > 1$ ; (iii) logistic loss  $L_{\log}(z) = \log_2(1 + \exp(-z))$ ; (iv) exponential loss  $L_{\exp}(z) = \exp(-z)$ ; (v) squared loss  $L_{\text{sq}}(z) = (1 - z)^2$  (this can be set to 0 for  $z > 1$ , just like hinge loss).

taking their sign into account. As a result, 0–1 loss doesn’t distinguish between scoring classifiers, as long as their predictions agree. This means that it isn’t actually that useful as a search heuristic or objective function when learning scoring classifiers. Figure 2.6 pictures several loss functions that are used in practice. Except for 0–1 loss, they are all *convex*: linear interpolation between any two points on the curve will never result in a point below the curve. Optimising a convex function is computationally more tractable.

One loss function that will be of interest later is the *hinge loss*, which is defined as  $L_h(z) = (1 - z)$  if  $z \leq 1$ , and  $L_h(z) = 0$  if  $z > 1$ . The name of this loss function comes from the fact that the loss ‘hinges’ on whether an example’s margin is greater than 1 or not: if so (i.e., the example is on the correct side of the decision boundary with a distance of at least 1) the example incurs zero loss; if not, the loss increases with decreasing margin. In effect, the loss function expresses that it is important to avoid examples having a margin (much) less than 1, but no additional value is placed on achieving large positive margins. This loss function is used when training a *support vector machine* (Section 7.3). We will also encounter *exponential loss* later when we discuss *boosting* in Section 11.2.

### Assessing and visualising ranking performance

It should be kept in mind that scores are assigned by a classifier, and are not a property inherent to instances. Scores are not estimated from ‘true scores’ – rather, a scoring classifier has to be learned from examples in the form of instances  $x$  labelled with

classes  $c(x)$ , just as a classifier. (The task where we learn a function  $\hat{f}$  from examples labelled with true function values  $(x, f(x))$  is called *regression* and is covered in [Section 3.2](#).) Often it is more convenient to keep the order imposed by scores on a set of instances, but ignore their magnitudes – this has the advantage, for instance, of being much less sensitive to outliers. It also means that we do not have to make any assumptions about the scale on which scores are expressed: in particular, a ranker does not assume a particular score threshold for separating positives from negatives. A *ranking* is defined as a total order on a set of instances, possibly with ties.<sup>9</sup>

**Example 2.2 (Ranking example).** The scoring tree in [Figure 2.5](#) produces the following ranking:  $[20+, 5-][10+, 5-][20+, 40-]$ . Here,  $20+$  denotes a sequence of 20 positive examples, and instances in square brackets [...] are tied. By selecting a split point in the ranking we can turn the ranking into a classification. In this case there are four possibilities: (A) setting the split point before the first segment, and thus assigning all segments to the negative class; (B) assigning the first segment to the positive class, and the other two to the negative class; (C) assigning the first two segments to the positive class; and (D) assigning all segments to the positive class. In terms of actual scores, this corresponds to (A) choosing any score larger than 2 as the threshold; (B) choosing a threshold between 1 and 2; (C) setting the threshold between  $-1$  and  $1$ ; and (D) setting it lower than  $-1$ .

Suppose  $x$  and  $x'$  are two instances such that  $x$  receives a lower score:  $\hat{s}(x) < \hat{s}(x')$ . Since higher scores express a stronger belief that the instance in question is positive, this would be fine except in one case: if  $x$  is an actual positive and  $x'$  is an actual negative. We will call this a *ranking error*. The total number of ranking errors can then be expressed as  $\sum_{x \in Te^{\oplus}, x' \in Te^{\ominus}} I[\hat{s}(x) < \hat{s}(x')]$ . Furthermore, for every positive and negative that receive the same score – a *tie* – we count half a ranking error. The maximum number of ranking errors is equal to  $|Te^{\oplus}| \cdot |Te^{\ominus}| = Pos \cdot Neg$ , and so the *ranking error rate* is defined as

$$rank-err = \frac{\sum_{x \in Te^{\oplus}, x' \in Te^{\ominus}} I[\hat{s}(x) < \hat{s}(x')] + \frac{1}{2} I[\hat{s}(x) = \hat{s}(x')]}{Pos \cdot Neg} \quad (2.4)$$

and analogously the *ranking accuracy*

$$rank-acc = \frac{\sum_{x \in Te^{\oplus}, x' \in Te^{\ominus}} I[\hat{s}(x) > \hat{s}(x')] + \frac{1}{2} I[\hat{s}(x) = \hat{s}(x')]}{Pos \cdot Neg} = 1 - rank-err \quad (2.5)$$

<sup>9</sup>A total order with ties should not be confused with a partial order (see [Background 2.1](#) on p.51). In a total order with ties (which is really a total order on equivalence classes), any two elements are comparable, either in one direction or in both. In a partial order some elements are incomparable.

Ranking accuracy can be seen as an estimate of the probability that an arbitrary positive–negative pair is ranked correctly.

---

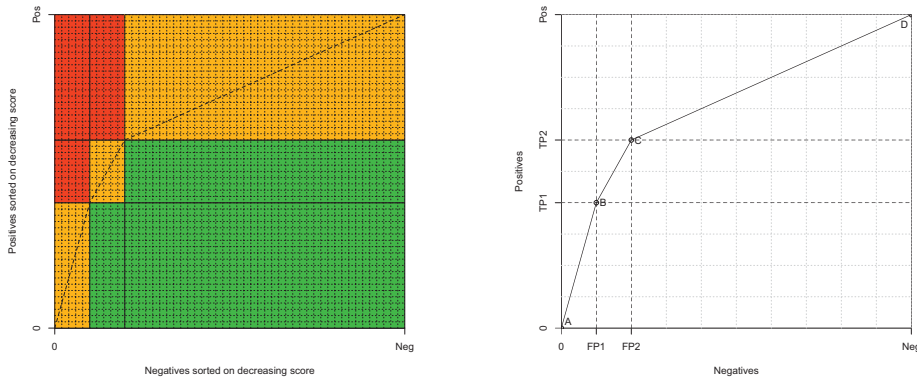
**Example 2.3 (Ranking accuracy).** We continue the previous example considering the scoring tree in [Figure 2.5](#), with the left leaf covering 20 spam and 40 ham, the middle leaf 10 spam and 5 ham, and the right leaf 20 spam and 5 ham. The 5 negatives in the right leaf are scored higher than the 10 positives in the middle leaf and the 20 positives in the left leaf, resulting in  $50 + 100 = 150$  ranking errors. The 5 negatives in the middle leaf are scored higher than the 20 positives in the left leaf, giving a further 100 ranking errors. In addition, the left leaf makes 800 half ranking errors (because 20 positives and 40 negatives get the same score), the middle leaf 50 and the right leaf 100. In total we have 725 ranking errors out of a possible  $50 \cdot 50 = 2500$ , corresponding to a ranking error rate of 29% or a ranking accuracy of 71%.

---

The coverage plots and ROC plots introduced in the previous section for visualising classifier performance provide an excellent tool for visualising ranking performance too. If *Pos* positives and *Neg* negatives are plotted on the vertical and horizontal axes, respectively, then each positive–negative pair occupies a unique ‘cell’ in this plot. If we order the positives and negatives on decreasing score, i.e., examples with higher scores are closer to the origin, then we can clearly distinguish the correctly ranked pairs at the bottom right, the ranking errors at the top left, and the ties in between ([Figure 2.7](#)). The number of cells in each area gives us the number of correctly ranked pairs, ranking errors and ties, respectively. The diagonal lines cut the ties area in half, so the area below those lines corresponds to the ranking accuracy multiplied by  $Pos \cdot Neg$ , and the area above corresponds to the ranking error rate times that same factor.

Concentrating on those diagonal lines gives us the piecewise linear curve shown in [Figure 2.7 \(right\)](#). This curve, which we will call a *coverage curve*, can be understood as follows. Each of the points marked A, B, C and D specifies the classification performance, in terms of true and false positives, achieved by the corresponding ranking split points or score thresholds from [Example 2.2](#). To illustrate, C would be obtained by a score threshold of 0, leading to  $TP2 = 20 + 10 = 30$  true positives and  $FP2 = 5 + 5 = 10$  false positives. Similarly, B would be obtained by a higher threshold of 1.5, leading to  $TP1 = 20$  true positives and  $FP1 = 5$  false positives. Point A would result if we set the threshold unattainably high, and D if we set the threshold trivially low.

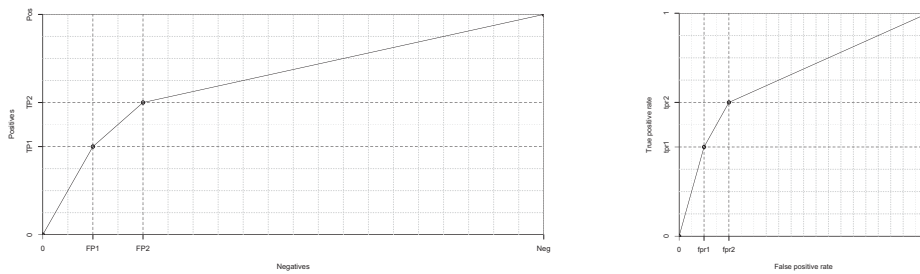
Why are these points connected by straight lines? How can we interpolate between, say, points C and D? Suppose we set the threshold exactly at  $-1$ , which is the score



**Figure 2.7.** (left) Each cell in the grid denotes a unique pair of one positive and one negative example: the green cells indicate pairs that are correctly ranked by the classifier, the red cells represent ranking errors, and the orange cells are half-errors due to ties. (right) The coverage curve of a tree-based scoring classifier has one line segment for each leaf of the tree, and one  $(FP, TP)$  pair for each possible threshold on the score.

assigned by the left leaf of the tree. The question is now what class we predict for the 20 positives and 40 negatives that filter down to that leaf. It would seem reasonable to decide this by tossing a fair coin, leading to half of the positives receiving a positive prediction (on average) and half of them a negative one, and similar for the negatives. The total number of true positives is then  $30 + 20/2 = 40$ , and the number of false positives is  $10 + 40/2 = 30$ . In other words, we land exactly in the middle of the CD line segment. We can apply the same procedure to achieve performance half-way BC, by setting the threshold at 1 and tossing the same fair coin to obtain uniformly distributed predictions for the 10 positives and 5 negatives in the middle leaf, leading to  $20 + 10/2 = 25$  true positives and  $5 + 5/2 = 7.5$  false positives (of course, we cannot achieve a non-integer number of false positives in any trial, but this number represents the expected number of false positives over many trials). And what's more, by biasing the coin towards positive or negative predictions we can achieve expected performance anywhere on the line.

More generally, a coverage curve is a piecewise linear curve that rises monotonically from  $(0,0)$  to  $(Neg, Pos)$  – i.e.,  $TP$  and  $FP$  can never decrease if we decrease the decision threshold. Each segment of the curve corresponds to an equivalence class of the instance space partition induced by the model in question (e.g., the leaves of a feature tree). Notice that the number of segments is never more than the number of test instances. Furthermore, the slope of each segment is equal to the ratio of positive to negative test instances in that equivalence class. For instance, in our example the first segment has a slope of 4, the second segment slope 2, and the third segment slope  $1/2$



**Figure 2.8.** (left) A coverage curve obtained from a test set with class ratio  $clr = 1/2$ . (right) The corresponding ROC curve is the same as the one corresponding to the coverage curve in Figure 2.7 (right).

— exactly the scores assigned in each leaf of the tree! This is not true in general, since the coverage curve depends solely on the ranking induced by the scores, not on the scores themselves. However, it is not a coincidence either, as we shall see in the next section on class probability estimation.

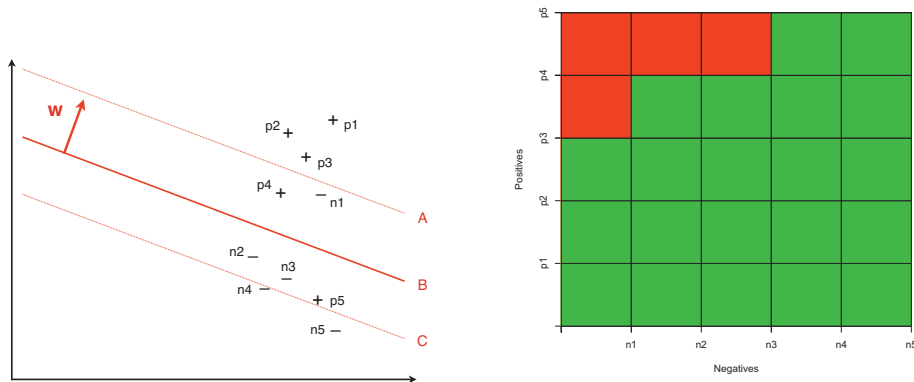
An *ROC curve* is obtained from a coverage curve by normalising the axes to  $[0, 1]$ . This doesn't make much of a difference in our running example, but in general coverage curves can be rectangular whereas ROC curves always occupy the unit square. One effect this has is that slopes are multiplied by  $Neg/Pos = 1/clr$ . Furthermore, while in a coverage plot the area under the coverage curve gives the absolute number of correctly ranked pairs, in an ROC plot *the area under the ROC curve is the ranking accuracy* as defined in Equation 2.5 on p.64. For that reason people usually write *AUC* for 'Area Under (ROC) Curve', a convention I will follow.

---

**Example 2.4 (Class imbalance).** Suppose we feed the scoring tree in Figure 2.5 on p.62 an extended test set, with an additional batch of 50 negatives. The added negatives happen to be identical to the original ones, so the net effect is that the number of negatives in each leaf doubles. As a result the coverage curve changes (because the class ratio changes), but the ROC curve stays the same (Figure 2.8). Note that the AUC stays the same as well: while the classifier makes twice as many ranking errors, there are also twice as many positive–negative pairs, so the ranking error rate doesn't change.

---

Let us now consider an example of a coverage curve for a grading classifier. Figure 2.9 (left) shows a linear classifier (the decision boundary is denoted B) applied to a



**Figure 2.9. (left)** A linear classifier induces a ranking by taking the signed distance to the decision boundary as the score. This ranking only depends on the orientation of the decision boundary: the three lines result in exactly the same ranking. **(right)** The grid of correctly ranked positive-negative pairs (in green) and ranking errors (in red).

small data set of five positive and five negative examples, achieving an accuracy of 0.80. We can derive a score from this linear classifier by taking the distance of an example from the decision boundary; if the example is on the negative side we take the negative distance. This means that the examples are ranked in the following order:  $p1 - p2 - p3 - n1 - p4 - n2 - n3 - p5 - n4 - n5$ . This ranking incurs four ranking errors:  $n1$  before  $p4$ , and  $n1$ ,  $n2$  and  $n3$  before  $p5$ . Figure 2.9 (right) visualises these four ranking errors in the top-left corner. The AUC of this ranking is  $21/25 = 0.84$ .

From this grid we obtain the coverage curve in Figure 2.10. Because of its stepwise character, this curve looks quite different from the coverage curves for scoring trees that we saw earlier in this section. The main reason is the absence of ties, which means that all segments in the curve are horizontal or vertical, and that there are as many segments as examples. We can generate this stepwise curve from the ranking as follows: starting in the lower left-hand corner, we go up one step if the next example in the ranking is positive, and right one step if the next example is negative. The result is a curve that goes three steps up (for  $p1-3$ ), one step to the right (for  $n1$ ), one step up ( $p4$ ), two steps to the right ( $n2-3$ ), one step up ( $p5$ ), and finally two steps to the right ( $n4-5$ ).

We can actually use the same procedure for grouping models if we handle ties as follows: in case of a tie between  $p$  positive examples and  $n$  negative examples, we go  $p$  steps up and *at the same time*  $n$  steps to the right. Looking back at Figure 2.7 on p.66, you will see that this is exactly what happens in the diagonal segments spanning the orange rectangles which arise as a result of the ties in the leaves of the decision tree. Thus, the principles underlying coverage and ROC curves are the same for both



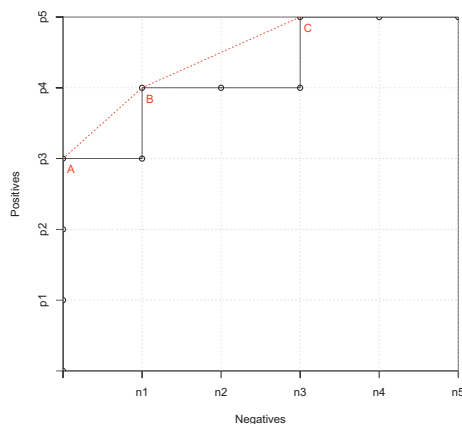
grouping and grading models, but the curves themselves look quite different in each case. *Grouping model ROC curves have as many line segments as there are instance space segments in the model; grading models have one line segment for each example in the data set.* This is a concrete manifestation of something I mentioned in the [Prologue](#): grading models have a much higher ‘resolution’ than grouping models; this is also called the model’s *refinement*.

Notice the three points in [Figure 2.10](#) labelled A, B and C. These points indicate the performance achieved by the decision boundaries with the same label in [Figure 2.9](#). As an illustration, the middle boundary B misclassifies one out of five positives ( $tpr = 0.80$ ) and one out of five negatives ( $fpr = 0.80$ ). Boundary A doesn’t misclassify any negatives, and boundary C correctly classifies all positives. In fact, while they should all have the same orientation, their exact location is not important, as long as boundary A is between  $p_3$  and  $n_1$ , boundary B is between  $p_4$  and  $n_2$ , and boundary C is between  $p_5$  and  $n_4$ . There are good reasons why I chose exactly these three boundaries, as we shall see shortly. For the moment, observe what happens if we use all three boundaries to turn the linear model into a grouping model with four segments: the area above A, the region between A and B, the bit between B and C, and the rest below C. The result is that we no longer distinguish between  $n_1$  and  $p_4$ , nor between  $n_{2-3}$  and  $p_5$ . The ties just introduced change the coverage curve to the dotted segments in [Figure 2.10](#). Notice that this results in a larger AUC of 0.90. Thus, *by decreasing a model’s refinement we sometimes achieve better ranking performance*. Training a model is not just about amplifying significant distinctions, but also about diminishing the effect of misleading distinctions.

### Turning rankers into classifiers

I mentioned previously that the main difference between rankers and scoring classifiers is that a ranker only assumes that a higher score means stronger evidence for the positive class, but otherwise makes no assumptions about the scale on which scores are expressed, or what would be a good score threshold to separate positives from negatives. We will now consider the question how to obtain such a threshold from a coverage curve or ROC curve.

The key concept is that of the accuracy isometric. Recall that in a coverage plot points of equal accuracy are connected by lines with slope 1. All we need to do, therefore, is to draw a line with slope 1 through the top-left point (which is sometimes called *ROC heaven*) and slide it down until we touch the coverage curve in one or more points. Each of those points achieves the highest accuracy possible with that model. In [Figure 2.10](#) this method would identify points A and B as the points with highest accuracy (0.80). They achieve this in different ways: e.g., model A is more conservative on the positives.

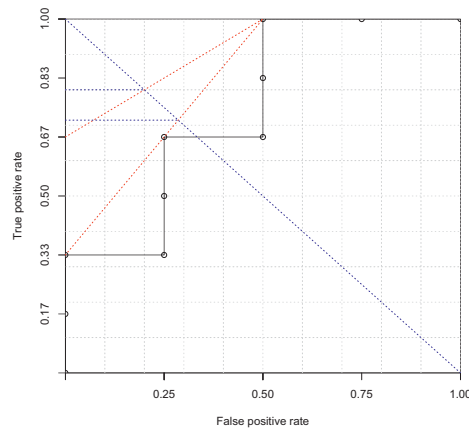


**Figure 2.10.** The coverage curve of the linear classifier in Figure 2.9. The points labelled A, B and C indicate the classification performance of the corresponding decision boundaries. The dotted lines indicate the improvement that can be obtained by turning the grading classifier into a grouping classifier with four segments.

A similar procedure can be followed with ROC plots, as long as you keep in mind that all slopes have to be multiplied by the reciprocal of the class ratio,  $1/clr = Neg/Pos$ .

**Example 2.5 (Tuning your spam filter).** You have carefully trained your Bayesian spam filter, and all that remains is setting the decision threshold. You select a set of six spam and four ham e-mails and collect the scores assigned by the spam filter. Sorted on decreasing score these are 0.89 (spam), 0.80 (spam), 0.74 (ham), 0.71 (spam), 0.63 (spam), 0.49 (ham), 0.42 (spam), 0.32 (spam), 0.24 (ham), and 0.13 (ham). If the class ratio of 3 spam against 2 ham is representative, you can select the optimal point on the ROC curve using an isometric with slope  $2/3$ . As can be seen in Figure 2.11, this leads to putting the decision boundary between the sixth spam e-mail and the third ham e-mail, and we can take the average of their scores as the decision threshold (0.28).

An alternative way of finding the optimal point is to iterate over all possible split points – from before the top ranked e-mail to after the bottom one – and calculate the number of correctly classified examples at each split: 4 – 5 – 6 – 5 – 6 – 7 – 6 – 7 – 8 – 7 – 6. The maximum is achieved at the same split point, yielding an accuracy of 0.80. A useful trick to find out which accuracy an isometric in an ROC plot represents is to intersect the isometric with the descending diagonal.



**Figure 2.11.** Selecting the optimal point on an ROC curve. The top dotted line is the accuracy isometric, with a slope of  $2/3$ . The lower isometric doubles the value (or prevalence) of negatives, and allows a choice of thresholds. By intersecting the isometrics with the descending diagonal we can read off the achieved accuracy on the  $y$ -axis.

Since accuracy is a weighted average of the true positive and true negative rates, and since these are the same in a point on the descending diagonal, we can read off the corresponding accuracy value on the  $y$ -axis.

If the class distribution in the data is *not* representative, we can simply adjust the slope of the isometric. For example, if ham is in fact twice as prevalent, we use an isometric with slope  $4/3$ . In the previous example this leads to three optimal points on the ROC curve.<sup>10</sup> Even if the class ratio in the data is representative, we may have other reasons to assign different weights to the classes. To illustrate, in the spam e-mail situation our spam filter may discard the false positives (ham e-mails misclassified as spam) so we may want to drive the false positive rate down by assigning a higher weight to the negatives (ham). This is often expressed as a *cost ratio*  $c = c_{FN} / c_{FP}$  of the cost of false negatives in proportion to the cost of false positives, which in this case would be set to a value smaller than 1. The relevant isometrics then have a slope of  $1/c$  in a coverage plot, and  $1/(c \cdot clr)$  in an ROC plot. The combination of cost ratio and class ratio gives a precise context in which the classifier is deployed and is referred to as the

<sup>10</sup>It seems reasonable to choose the middle of these three points, leading to a threshold of 0.56. An alternative is to treat all e-mails receiving a score in the interval  $[0.28, 0.77]$  as lying on the decision boundary, and to randomly assign a class to those e-mails.

*operating condition.*

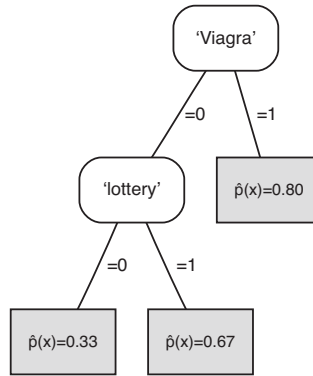
If the class or cost ratio is highly skewed, this procedure may result in a classifier that assigns the same class to all examples. For instance, if negatives are 1 000 times more prevalent than positives, accuracy isometrics are nearly vertical, leading to an unattainably high decision threshold and a classifier that classifies everything as negative. Conversely, if the profit of one true positive is 1 000 times the cost of a false positive, we would classify everything as positive – in fact, this is the very principle underlying spam e-mail! However, often such one-size-fits-all behaviour is unacceptable, indicating that accuracy is not the right thing to optimise here. In such cases we should use average recall isometrics instead. These run parallel to the ascending diagonal in both coverage and ROC plots, and help to achieve similar performance on both classes.

The procedure just described learns a decision threshold from labelled data by means of the ROC curve and the appropriate accuracy isometric. This procedure is often preferable over fixing a decision threshold in advance, particularly if scores are expressed on an arbitrary scale – for instance, this would provide a way to finetune the SpamAssassin decision threshold to our particular situation and preferences. Even if the scores are probabilities, as in the next section, these may not be sufficiently well estimated to warrant a fixed threshold of 0.5.

## 2.3 Class probability estimation

A *class probability estimator* – or probability estimator in short – is a scoring classifier that outputs probability vectors over classes, i.e., a mapping  $\hat{\mathbf{p}} : \mathcal{X} \rightarrow [0, 1]^k$ . We write  $\hat{\mathbf{p}}(x) = (\hat{p}_1(x), \dots, \hat{p}_k(x))$ , where  $\hat{p}_i(x)$  is the probability assigned to class  $C_i$  for instance  $x$ , and  $\sum_{i=1}^k \hat{p}_i(x) = 1$ . If we have only two classes, the probability associated with one class is 1 minus the probability of the other class; in that case, we use  $\hat{p}(x)$  to denote the estimated probability of the positive class for instance  $x$ . As with scoring classifiers, we usually do not have direct access to the true probabilities  $p_i(x)$ .

One way to understand the probabilities  $\hat{p}_i(x)$  is as estimates of the probability  $P_{\mathcal{C}}(c(x') = C_i | x' \sim x)$ , where  $x' \sim x$  stands for ‘ $x'$  is similar to  $x$ ’. In other words, how frequent are instances of this class among instances similar to  $x$ ? The intuition is that the more (or less) frequent they are, the more (or less) confident we should be in our belief that  $x$  belongs to that class as well. What we mean with similarity in this context will depend on the models we are considering – we will illustrate it here by means of a few two-class examples. First, assume a situation in which any two instances are similar to each other. We then have  $P_{\mathcal{C}}(c(x') = \oplus | x' \sim x) = P_{\mathcal{C}}(c(x') = \oplus)$  which is simply estimated by the proportion *pos* of positives in our data set (I am going to drop the subscript  $\mathcal{C}$  from now on). In other words, in this scenario we predict  $\hat{p}(x) = \text{pos}$  regardless



**Figure 2.12.** A probability estimation tree derived from the feature tree in Figure 1.4.

of whether we know anything about  $x$ 's true class. At the other extreme, consider a situation in which no two instances are similar unless they are the same, i.e.,  $x' \sim x$  if  $x' = x$ , and  $x' \not\sim x$  otherwise. In this case we have  $P(c(x') = \oplus | x' \sim x) = P(c(x) = \oplus)$ , which – because  $x$  is fixed – is 1 if  $c(x) = \oplus$  and 0 otherwise. Put differently, we predict  $\hat{p}(x) = 1$  for all known positives and  $\hat{p}(x) = 0$  for all known negatives, but we can't generalise this to unseen instances.

A feature tree allows us to strike a balance between these extreme and simplistic scenarios, using the similarity relation  $\sim_T$  associated with feature tree  $T$ :  $x' \sim_T x$  if, and only if,  $x$  and  $x'$  are assigned to the same leaf of the tree. In each leaf we then predict the proportion of positives assigned to that leaf. For example, in the right-most leaf in Figure 1.4 on p.32 the proportion of positives is  $40/50 = 0.80$ , and thus we predict  $\hat{p}(x) = 0.80$  for all instances  $x$  assigned to that leaf; similarly for the other two leaves (Figure 2.12). If we threshold  $\hat{p}(x)$  at 0.5 (i.e., predict spam if the spam probability is 0.5 or more and predict ham otherwise), we get the same classifier as obtained by predicting the majority class in each leaf of the feature tree.

### Assessing class probability estimates

As with classifiers, we can now ask the question of how good these class probability estimators are. A slight complication here is that, as already remarked, we do not have access to the true probabilities. One trick that is often applied is to define a binary vector  $(I[c(x) = C_1], \dots, I[c(x) = C_k])$ , which has the  $i$ -th bit set to 1 if  $x$ 's true class is  $C_i$  and all other bits set to 0, and use these as the 'true' probabilities. We can then define the *squared error (SE)* of the predicted probability vector  $\hat{\mathbf{p}}(x) = (\hat{p}_1(x), \dots, \hat{p}_k(x))$  as

$$\text{SE}(x) = \frac{1}{2} \sum_{i=1}^k (\hat{p}_i(x) - I[c(x) = C_i])^2 \quad (2.6)$$

and the *mean squared error* (*MSE*) as the average squared error over all instances in the test set:

$$\text{MSE}(Te) = \frac{1}{|Te|} \sum_{x \in Te} \text{SE}(x) \quad (2.7)$$

This definition of error in probability estimates is often used in *forecasting theory* where it is called the *Brier score*. The factor 1/2 in Equation 2.6 ensures that the squared error per example is normalised between 0 and 1: the worst possible situation is that the wrong class is predicted with probability 1, which means two ‘bits’ are wrong. For two classes this reduces to a single term  $(\hat{p}(x) - I[c(x) = \oplus])^2$  only referring to the positive class. Notice that, if a class probability estimator is ‘categorical’ – i.e., it assigns probability 1 to one class and probability 0 to the rest – it is effectively a classifier, and *MSE* reduces to accuracy as defined in Section 2.1.

**Example 2.6 (Squared error).** Suppose one model predicts (0.70, 0.10, 0.20) for a particular example  $x$  in a three-class task, while another appears much more certain by predicting (0.99, 0, 0.01). If the first class is the actual class, the second prediction is clearly better than the first: the SE of the first prediction is  $((0.70 - 1)^2 + (0.10 - 0)^2 + (0.20 - 0)^2)/2 = 0.07$ , while for the second prediction it is  $((0.99 - 1)^2 + (0 - 0)^2 + (0.01 - 0)^2)/2 = 0.0001$ . The first model gets punished more because, although mostly right, it isn’t quite sure of it.

However, if the third class is the actual class, the situation is reversed: now the SE of the first prediction is  $((0.70 - 0)^2 + (0.10 - 0)^2 + (0.20 - 1)^2)/2 = 0.57$ , and of the second  $((0.99 - 0)^2 + (0 - 0)^2 + (0.01 - 1)^2)/2 = 0.98$ . The second model gets punished more for not just being wrong, but being presumptuous.

Returning to the probability estimation tree in Figure 2.12, we calculate the squared error per leaf as follows (left to right):

$$\text{SE}_1 = 20(0.33 - 1)^2 + 40(0.33 - 0)^2 = 13.33$$

$$\text{SE}_2 = 10(0.67 - 1)^2 + 5(0.67 - 0)^2 = 3.33$$

$$\text{SE}_3 = 20(0.80 - 1)^2 + 5(0.80 - 0)^2 = 4.00$$

which leads to a mean squared error of  $\text{MSE} = \frac{1}{100}(\text{SE}_1 + \text{SE}_2 + \text{SE}_3) = 0.21$ . An interesting question is whether we can change the predicted probabilities in each leaf to obtain a lower mean squared error. It turns out that this is not possible: predicting probabilities obtained from the class distributions in each leaf is optimal in the sense of lowest *MSE*.

For instance, changing the predicted probabilities in the left-most leaf to 0.40 for spam and 0.60 for ham, or 0.20 for spam and 0.80 for ham, results in a higher squared error:

$$SE'_1 = 20(0.40 - 1)^2 + 40(0.40 - 0)^2 = 13.6$$

$$SE''_1 = 20(0.20 - 1)^2 + 40(0.20 - 0)^2 = 14.4$$

The reason for this becomes obvious if we rewrite the expression for two-class squared error of a leaf as follows, using the notation  $n^{\oplus}$  and  $n^{\ominus}$  for the numbers of positive and negative examples in the leaf:

$$\begin{aligned} n^{\oplus}(\hat{p} - 1)^2 + n^{\ominus}\hat{p}^2 &= (n^{\oplus} + n^{\ominus})\hat{p}^2 - 2n^{\oplus}\hat{p} + n^{\oplus} = (n^{\oplus} + n^{\ominus})[\hat{p}^2 - 2\hat{p}\hat{p} + \hat{p}] \\ &= (n^{\oplus} + n^{\ominus})[(\hat{p} - \dot{p})^2 + \dot{p}(1 - \dot{p})] \end{aligned}$$

where  $\dot{p} = n^{\oplus} / (n^{\oplus} + n^{\ominus})$  is the relative frequency of the positive class among the examples covered by the leaf, also called the *empirical probability*. As the term  $\dot{p}(1 - \dot{p})$  does not depend on the predicted probability  $\hat{p}$ , we see immediately that we achieve lowest squared error in the leaf if we assign  $\hat{p} = \dot{p}$ .

Empirical probabilities are important as they allow us to obtain or finetune probability estimates from classifiers or rankers. If we have a set  $S$  of labelled examples, and the number of examples in  $S$  of class  $C_i$  is denoted  $n_i$ , then the empirical probability vector associated with  $S$  is  $\dot{\mathbf{p}}(S) = (n_1/|S|, \dots, n_k/|S|)$ . In practice, it is almost always a good idea to *smooth* these relative frequencies to avoid issues with extreme values (0 or 1). The most common way to do this is to set

$$\dot{p}_i(S) = \frac{n_i + 1}{|S| + k} \quad (2.8)$$

This is called the *Laplace correction*, after the French mathematician Pierre-Simon Laplace, who introduced it for the case  $k = 2$  (also known as Laplace's rule of succession). In effect, we are adding uniformly distributed *pseudo-counts* to each of the  $k$  alternatives, reflecting our prior belief that the empirical probabilities will turn out uniform.<sup>11</sup> We can also apply non-uniform smoothing by setting

$$\dot{p}_i(S) = \frac{n_i + m \cdot \pi_i}{|S| + m} \quad (2.9)$$

This smoothing technique, known as the *m-estimate*, allows the choice of the number of pseudo-counts  $m$  as well as the prior probabilities  $\pi_i$ . The Laplace correction is a special case of the *m-estimate* with  $m = k$  and  $\pi_i = 1/k$ .

If all elements of  $S$  receive the same predicted probability vector  $\hat{\mathbf{p}}(S)$  – which happens if  $S$  is a segment of a grouping model – then a similar derivation to the one above

<sup>11</sup>This can be modelled mathematically by a prior probability distribution known as a *Dirichlet prior*.

allows us to write the total incurred squared error over  $S$  in terms of estimated and empirical probabilities as

$$\begin{aligned}\text{SE}(S) &= \sum_{x \in S} \text{SE}(x) = \sum_{x \in S} \frac{1}{2} \sum_{i=1}^k (\hat{p}_i(x) - I[c(x) = C_i])^2 \\ &= \frac{1}{2} |S| \sum_{i=1}^k (\hat{p}_i(S) - p_i(S))^2 + \frac{1}{2} |S| \sum_{i=1}^k (\hat{p}_i(S)(1 - \hat{p}_i(S)))\end{aligned}$$

The **first term** of the final expression is called the *calibration loss*, and measures squared error with respect to the empirical probabilities. It can be reduced to 0 in grouping models where we are free to choose the predicted probabilities for each segment, as in probability estimation trees. Models with low calibration loss are said to be well-calibrated. The **second term** is called the *refinement loss*; this depends only on the empirical probabilities, and is smaller if they are less uniform.

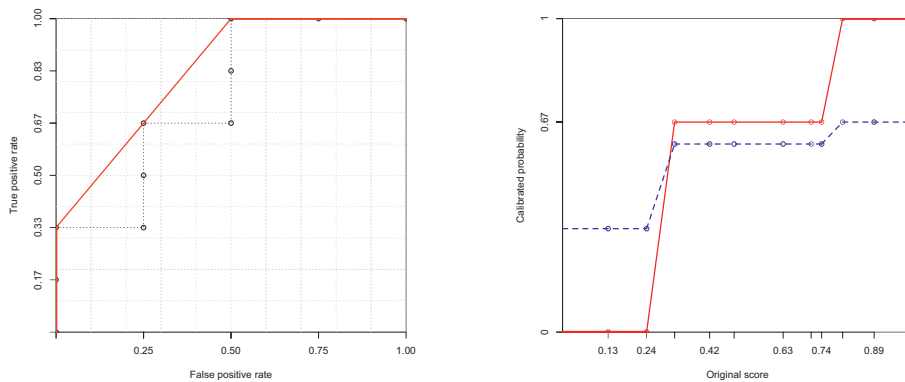
This analysis suggests that the best way of obtaining probability estimates is from empirical probabilities, obtained from the training set or from another set of labelled examples specifically set aside for the purpose. However, there are two issues we need to consider here. The first is that with some models we must make sure that the predicted probabilities obey the ranking imposed by the model. The second is that with grading models we don't have immediate access to empirical probabilities, since each example tends to get assigned an equivalence class of its own. We will now discuss this in a bit more detail.

### Turning rankers into class probability estimators

Consider again [Example 2.5](#) on [p.70](#), and imagine the scores are not probabilities but on some unknown scale, so that the spam filter is a ranker rather than a class probability estimator. Since each test example receives a different score, the 'empirical probabilities' are either 0 (for negative examples) or 1 (for positive examples), leading to a sequence of  $\hat{p}$ -values of 1 – 1 – 0 – 1 – 1 – 0 – 1 – 1 – 0 – 0 in order of decreasing scores. The obvious problem is that these  $\hat{p}$ -values do not obey the order imposed by the scores, and so cannot be used directly to obtain probability estimates. Smoothing the empirical probabilities using Laplace correction doesn't really address this problem, since all it does is replace 0 with 1/3 and 1 with 2/3. We need a different idea.

Looking at [Figure 2.11](#), we see that  $\hat{p} = 1$  corresponds to a vertical segment of the ROC curve, and  $\hat{p} = 0$  to a horizontal segment. The problem we have is caused by having a vertical segment following a horizontal one, or, more generally, a segment with steeper slope following a flatter segment. We will call a sequence of segments with increasing slope a *concavity*, as it forms a 'dent' in the ROC curve. A curve without concavities is a *convex* ROC curve. Our curve has two concavities: one formed by the third, fourth and fifth example, and the other by the sixth, seventh and eighth example.





**Figure 2.13.** (left) The solid red line is the convex hull of the dotted ROC curve. (right) The corresponding calibration map in red: the plateaus correspond to several examples being mapped to the same segment of the convex hull, and linear interpolation between example scores occurs when we transition from one convex hull segment to the next. A Laplace-corrected calibration map is indicated by the dashed line in blue: Laplace smoothing compresses the range of calibrated probabilities but can sometimes affect the ranking.

Suppose now that the third to the fifth example all receive the same score, say 0.7; and the sixth to the eighth example are also tied, say at 0.4. In that case the ROC curve would have six segments, with empirical probabilities  $1 - 1 - 2/3 - 2/3 - 0 - 0$ . As we see, the  $\hat{p}$ -values are now decreasing with the scores; in other words, the concavities have disappeared and the ROC curve has become convex.

More generally speaking, *concavities in ROC curves can be remedied by combining segments through tied scores*. This is achieved by identifying what are sometimes called *adjacent violators*. For instance, in the sequence  $1 - 1 - 0 - 1 - 1 - 0 - 1 - 1 - 0 - 0$ , the third and fourth example are adjacent violators, because they violate the rule that scores should be decreasing from left to right in the sequence (or, mathematically more accurate, they should be non-increasing). This is remedied by assigning them both their average score, leading to the sequence  $1 - 1 - [1/2 - 1/2] - 1 - 0 - 1 - 1 - 0 - 0$ . The newly introduced segment now forms an adjacent violator pair with the fourth example, so we give them all their mean score, leading to the sequence  $1 - 1 - [2/3 - 2/3 - 2/3] - 0 - 1 - 1 - 0 - 0$ .<sup>12</sup> The second  $0 - 1 - 1$  concavity is treated identically, and the final sequence is  $1 - 1 - [2/3 - 2/3 - 2/3] - [2/3 - 2/3 - 2/3] - 0 - 0$ .

The result is illustrated in Figure 2.13. On the left, we see how the two concavities are replaced with two diagonal line segments with the same slope. These diagonal segments coincide with the accuracy isometric that gives the three ‘outermost’ points

<sup>12</sup>These two steps can be combined into one: once a pair of adjacent violators is found, we can scan to the left and right to include examples with the same score as the left and right example in the pair, respectively.

involved in the concavities the same accuracy (Figure 2.11 on p.71). Jointly, the **red segments** constitute the **convex hull** of the ROC curve, which is the unique convex curve through the outermost points of the original ROC curve. The convex hull has a higher AUC than the original curve, because it replaces (some of) the ranking errors of the original curve with half-errors due to ties. In our example the original ranking incurs 6 out of 24 ranking errors (AUC = 0.75), while the convex hull turns all of these into half-errors (AUC = 0.83).

Once we have determined the convex hull, we can use the empirical probabilities in each segment of the convex hull as calibrated probabilities. Figure 2.13 (right) shows the resulting **calibration map**, which is a piecewise linear, non-decreasing curve mapping original scores on the  $x$ -axis to calibrated probabilities on the  $y$ -axis. Also shown is an alternative calibration map giving probability estimates after Laplace correction: for the given sequence these are  $2/3 - 2/3 - [3/5 - 3/5 - 3/5] - [3/5 - 3/5 - 3/5] - 1/3 - 1/3$ , giving rise to a much compressed range of probability estimates.

Let's now look at this process from the point of view of mean squared error, calibration and refinement. The original scores had a mean squared error of  $\frac{1}{10}[(0.89 - 1)^2 + (0.80 - 1)^2 + (0.74 - 0)^2 + (0.71 - 1)^2 + (0.63 - 1)^2 + (0.49 - 0)^2 + (0.42 - 1)^2 + (0.32 - 1)^2 + (0.24 - 0)^2 + (0.13 - 0)^2] = 0.19$ . Notice that this is entirely incurred by the calibration loss, as all empirical probabilities are either 0 or 1 and thus the refinement loss is zero. The calibrated scores have a mean squared error of  $\frac{1}{10}[(1 - 1)^2 + (1 - 1)^2 + (0.67 - 0)^2 + (0.67 - 1)^2 + (0.67 - 1)^2 + (0.67 - 0)^2 + (0.67 - 1)^2 + (0.67 - 1)^2 + (0 - 0)^2 + (0 - 0)^2] = 0.13$ . Now the entire mean squared error is incurred by refinement loss as the estimated probabilities are equal to the empirical ones in each segment by construction. We have traded an increase in refinement loss for a decrease in calibration loss; since the latter is larger than the former, the overall error decreases. The increase in refinement loss comes from the construction of the convex hull, which introduces diagonal segments. The technical term for this process of obtaining calibrated scores through the convex hull of the ROC curve is **isotonic calibration**, as the mathematical problem underlying it is called isotonic regression. Some caution is in order when applying isotonic calibration, as it is easy to overfit the data in this process. In the calibration map in Figure 2.13 (right), both the horizontal transition points and the vertical levels are directly obtained from the given data, and may not generalise well to unseen data. This is why it is advisable to apply the Laplace correction to the empirical probabilities, even though it will increase the calibration loss on the given data.

## 2.4 Binary classification and related tasks: Summary and further reading

In this chapter we have looked at binary classification, a ubiquitous task that forms the starting point of a lot of work in machine learning. Although we haven't talked much about learning in this chapter, my philosophy is that you will reach a better understanding of machine learning models and algorithms if you first study the tasks that these models are meant to address.

☞ In [Section 2.1](#) we defined the binary classification task and introduced an important tool to assess performance at such a task, namely the two-by-two contingency table. A wide range of performance indicators are derived from the counts in a contingency table. I introduced the coverage plot, which visualises a contingency table as a rectangle with size *Pos* up and size *Neg* across, and within that rectangle a point with *y*-coordinate *TP* and *x*-coordinate *FP*. We can visualise several models evaluated on the same data set by several points, and use the fact that accuracy is constant along line segments with slope 1 to visually rank these classifiers on accuracy. Alternatively, we can normalise the rectangle to be a unit square with true and false positive rate on the axes. In this so-called ROC space, line segments with slope 1 (i.e., those parallel to the ascending diagonal) connect points with the same average recall (sometimes also called macro-accuracy). The use of these kinds of plot in machine learning was pioneered by [Provost and Fawcett \(2001\)](#). Unnormalised coverage plots were introduced by [Fürnkranz and Flach \(2003\)](#).

☞ [Section 2.2](#) considered the more general task of calculating a score for each example (or a vector of scores in the general case of more than two classes). While the scale on which scores are expressed is unspecified, it is customary to put the decision threshold at  $\hat{s}(x) = 0$  and let the sign of the score stand for the prediction (positive or negative). Multiplying the score with the true class gives us the margin, which is positive for a correct prediction and negative for an incorrect one. A loss function determines how much negative margins are penalised and positive margins rewarded. The advantage of working with convex and continuously differentiable 'surrogate' loss functions (rather than with 0–1 loss, which is the loss function we ultimately want to optimise) is that this often leads to more tractable optimisation problems.

☞ Alternatively, we can ignore the scale on which scores are measured altogether and only work with their order. Such a ranker is visualised in coverage or ROC space by a piecewise continuous curve. For grouping models the line segments in these curves correspond to instance space segments (e.g., the leaves of a tree

model) whereas for grading models there is a segment for each unique score assigned by the model. The area under the ROC curve gives the ranking accuracy (an estimate of the probability that a random positive is ranked before a random negative) and is known in statistics as the Wilcoxon-Mann-Whitney statistic. These curves can be used to find a suitable operating point by translating the operating condition (class and cost distribution) into an isometric in ROC or coverage space. The origins of ROC curves are in signal detection theory (Egan, 1975); accessible introductions can be found in (Fawcett, 2006; Flach, 2010b).

☞ In Section 2.3 we looked at scoring models whose scores can be interpreted as estimates of the probability that the instance belongs to a particular class. Such models were pioneered in forecasting theory by Brier (1950) and Murphy and Winkler (1984), among others. We can assess the quality of class probability estimates by comparing them to the ‘ideal’ probabilities (1 for a positive, 0 for a negative) and taking mean squared error. Since there is no reason why the true probabilities should be categorical this is quite a crude assessment, and decomposing it into calibration loss and refinement loss provides useful additional information. We have also seen a very useful trick for smoothing relative frequency estimates of probabilities by adding pseudo-counts, either uniformly distributed (Laplace correction) or according to a chosen prior ( $m$ -estimate). Finally, we have seen how we can use the ROC convex hull to obtain calibrated class probability estimates. The approach has its roots in isotonic regression (Best and Chakravarti, 1990) and was introduced to the machine learning community by Zadrozny and Elkan (2002). Fawcett and Niculescu-Mizil (2007) and Flach and Matsubara (2007) show that the approach is equivalent to calibration by means of the ROC convex hull. (Note that in this chapter we have seen two different uses of the term ‘convex’: one in relation to loss functions, where convexity means that linear interpolation between any two points on the curve depicting the loss function will never result in a point below the curve; and the other in relation to the ROC convex hull, where it refers to the linearly interpolated boundary of a convex set which envelopes all points in the set.)

