CHAPTER 1

The ingredients of machine learning

MACHINE 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*.

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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 models'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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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`, 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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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,

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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` 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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

	Predictive model	*Descriptive model*
Supervised learning	classification, regression	subgroup discovery
Unsupervised learning	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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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.

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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 $\mathbf{p} = \frac{1}{n} \sum_{x \in P} \mathbf{x}$, 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.[1] 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

[1] It is a simplified version of linear discriminants.

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

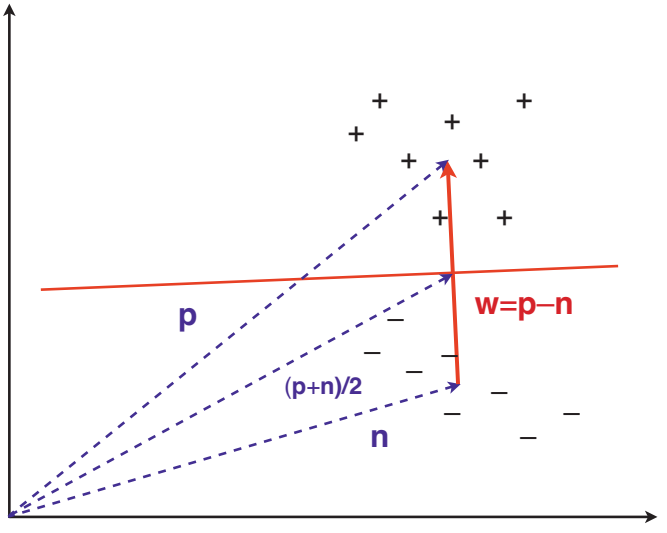

Figure 1.1. The basic linear classifier constructs a decision boundary by half-way intersecting the line between the positive and negative centres of mass. It is described by the equation $\mathbf{w} \cdot \mathbf{x} = t$, with $\mathbf{w} = \mathbf{p} - \mathbf{n}$; the decision threshold can be found 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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

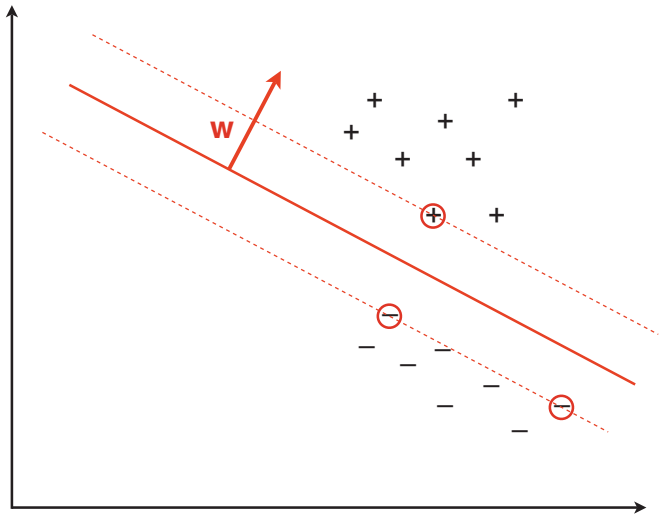

Figure 1.2. The decision boundary learned by a support vector machine from the linearly separable data from Figure 1. The decision boundary maximises the margin, which is indicated by the dotted lines. The circled data points are the support vectors.

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:[2] $\sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$. 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,

[2] 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 θ is the angle between $\mathbf{x}$ and $\mathbf{y}$.

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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

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

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 θ 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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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

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.

$1, \text{lottery} = 0)$ 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'

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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* $P(X|Y)$.[3] 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-

[3] 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.

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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 enumerator 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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

Y	$P(\text{Viagra} = 1\|Y)$	$P(\text{Viagra} = 0\|Y)$
spam	0.40	0.60
ham	0.12	0.88

Y	$P(\text{lottery} = 1\|Y)$	$P(\text{lottery} = 0\|Y)$
spam	0.21	0.79
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., $P(\text{Viagra}|Y) = \sum_{\text{lottery}} P(\text{Viagra}, \text{lottery}|Y)$.

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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*.[4] 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-

[4] 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.

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

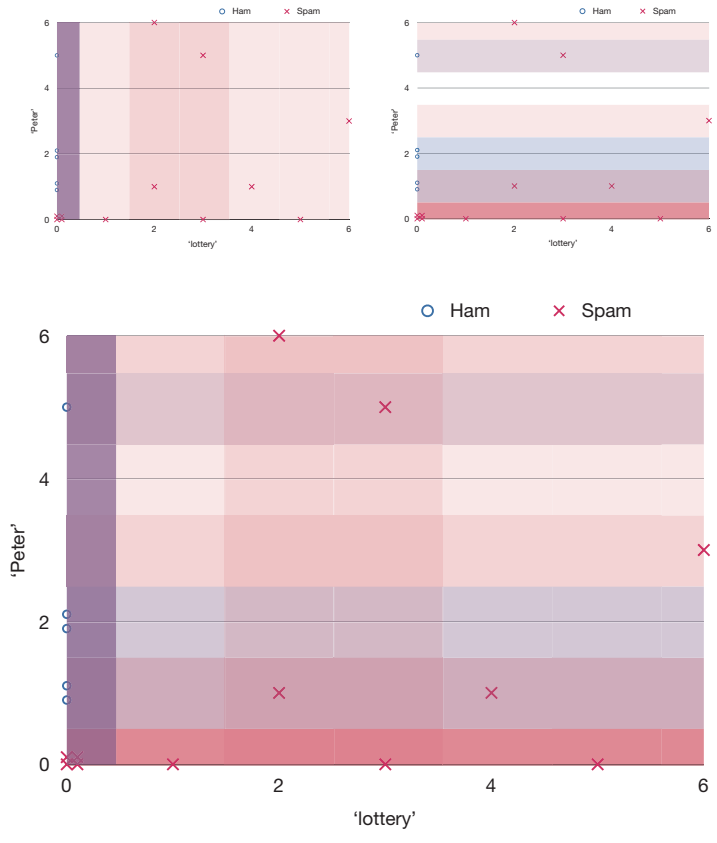

Figure 1.3. **(top)** Visualisation of two marginal likelihoods as estimated from a small data set. The colours indicate whether the likelihood points to spam or ham. **(bottom)** Combining the two marginal likelihoods gives a pattern not unlike that of a Scottish tartan. The colour of a particular cell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

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

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![5]

[5] Sometimes we need to slightly adapt the raw counts for very frequent or very infrequent words, as we shall see in Section 2.3.

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

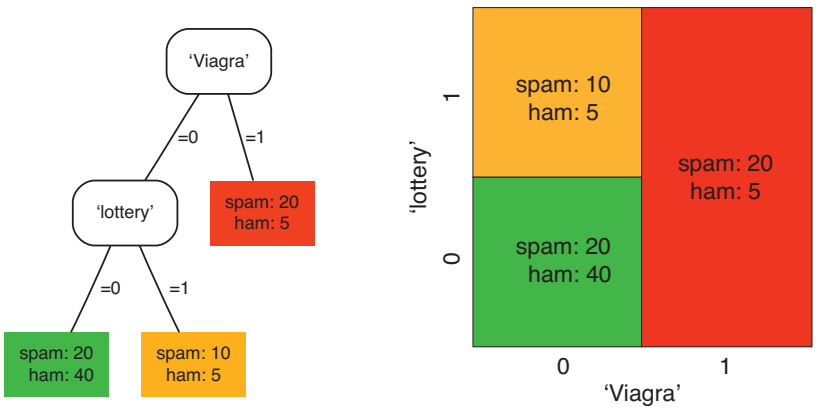

Figure 1.4. (left) A feature tree combining two Boolean features. Each internal node or split is labelled with a feature, and each edge emanating from a split is labelled with a feature value. Each leaf therefore corresponds to a unique combination of feature values. Also indicated in each leaf is the 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 $j = 0, 1, 2, \ldots$. 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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

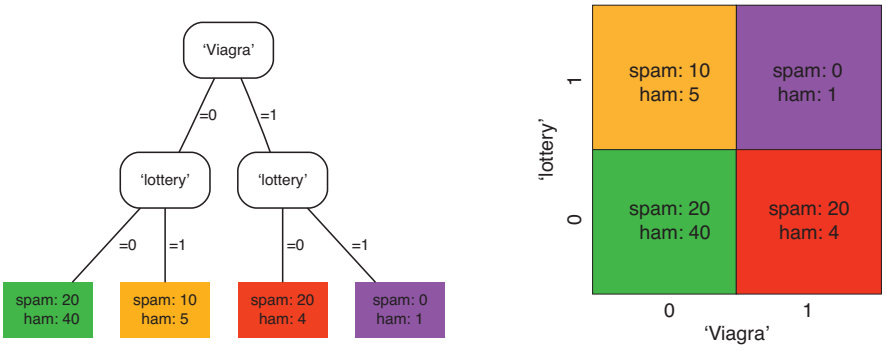

Figure 1.5. (left) A complete feature tree built from two Boolean features. **(right)** The corresponding instance space partition is the finest partition that can be achieved with those two features.

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 rightmost 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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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 ∨ lottery = 1 **then** Class = Y = spam·
·**else** Class = Y = ham·

·**if** Viagra = 0 ∧ 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 ∨. This selects a single non-rectangular area in instance space. The second model formulates a *conjunctive* condition ('and', denoted by ∧) 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 ∧ lottery = 1 **then** Class = Y = spam·
·**if** Viagra = 0 ∧ 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 Viagra = 0), every pair of rules will have at least some mutually exclusive conditions (such as lottery = 1 in the second rule and 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 lottery = 1 ∧ Peter = 1, for which they make contradictory predictions. Furthermore, they fail to make any predictions for lottery = 0 ∧ Peter = 0.

---

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

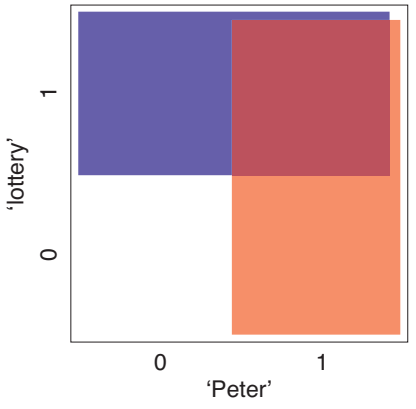

Figure 1.6. The effect of overlapping rules in instance space. The two rules make contradictory predictions in the top right-hand corner, and no prediction at all in the bottom left-hand corner.

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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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 ≥ 0.168; or
7. it has a hydroxy oxygen with a partial charge ≥ −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 ≤ −0.144 and tests positive on Progol's mutagenicity rules.[6]

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

[6] 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.

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

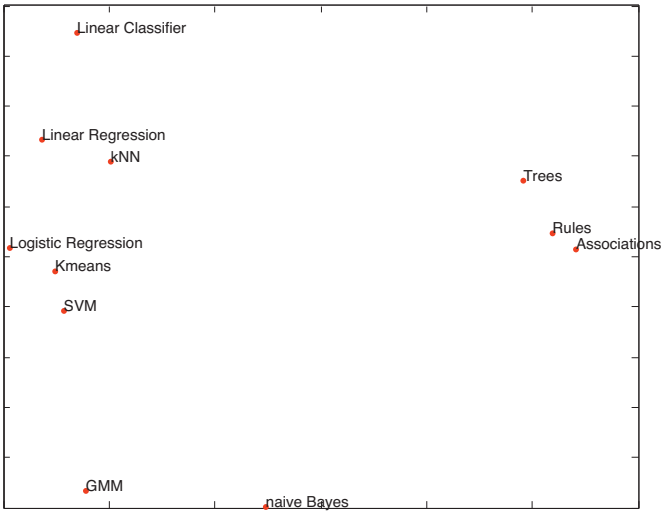

Figure 1.7. A 'map' of some of the models that will be considered in this book. Models that share characteristics are plotted closer together: logical models to the right, geometric models on the top left and probabilistic models on the bottom left. The horizontal dimension roughly ranges from grading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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

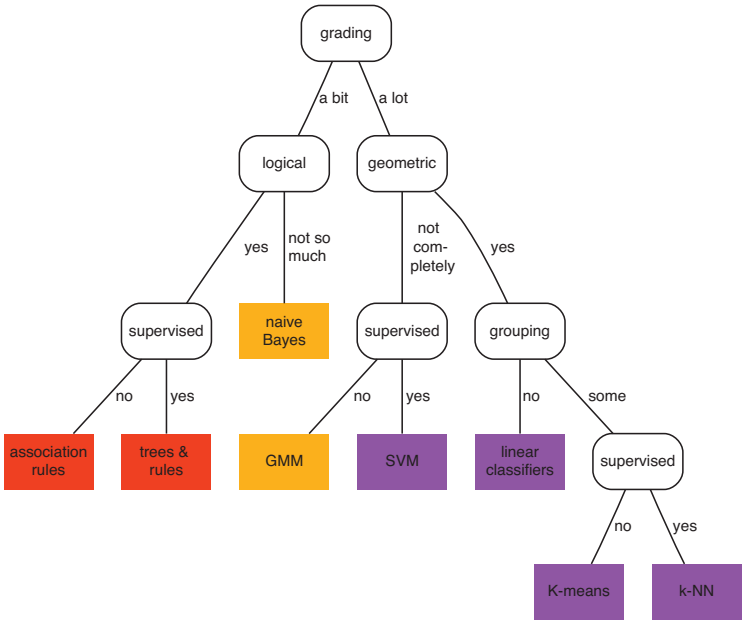

Figure 1.8. A taxonomy describing machine learning methods in terms of the extent to which they are grading or grouping models, logical, geometric or a combination, and supervised or unsupervised. The colours indicate the type of model, from left to right: logical (red), probabilistic (orange) and geometric (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.[7] 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-

[7] The figures have been generated from data explained in Example 1.7 below.

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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.[8] 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

[8] 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.

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

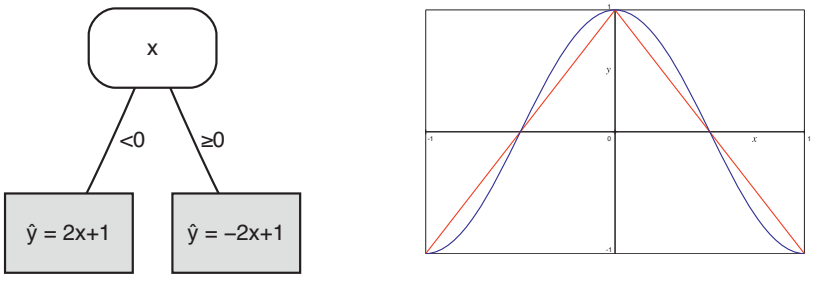

Figure 1.9. (left) A regression tree combining a one-split feature tree with linear regression models in the leaves. Notice how x is used as both a splitting feature and a regression variable. **(right)** The function $y = \cos\pi x$ on the interval $-1 \leq x \leq 1$, and the piecewise linear approximation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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

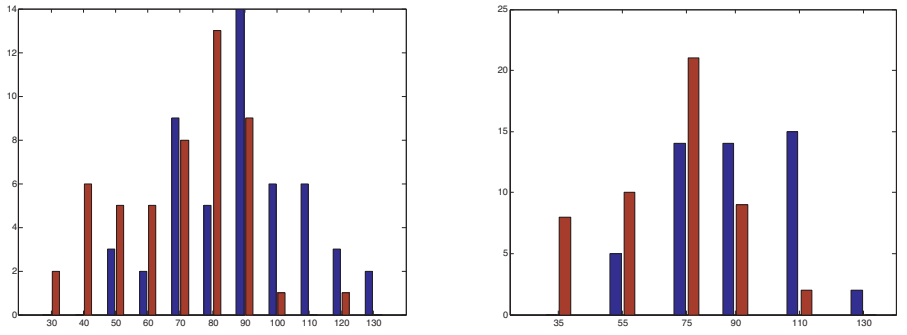

Figure 1.10. (left) Artificial data depicting a histogram of body weight measurements of people with (blue) and without (red) diabetes, with eleven fixed intervals of 10 kilograms width each. **(right)** By joining the first and second, third and fourth, fifth and sixth, and the eighth, ninth and tenth 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.

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

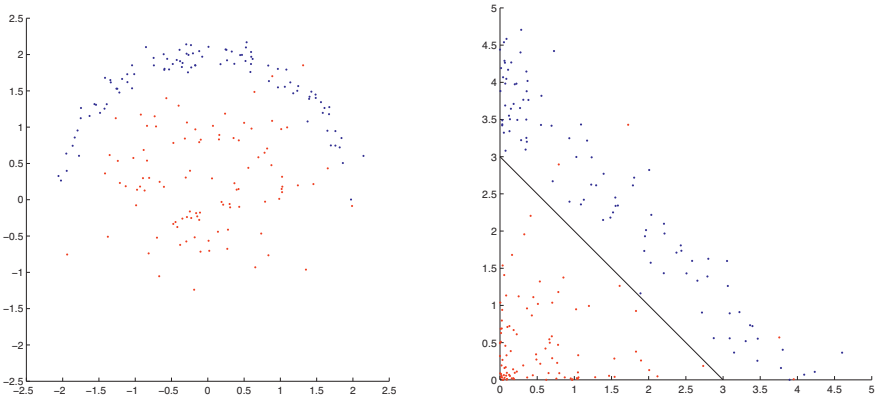

Figure 1.11. (left) A linear classifier would perform poorly on this data. **(right)** By transforming the original (x, y) data into $(x', y') = (x^2, y^2)$, the data becomes more 'linear', and a linear decision boundary $x' + y' = 3$ separates the data fairly well. In the original space this corresponds 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 $\mathbf{x}'_1 = (x_1^2, y_1^2, \sqrt{2}x_1 y_1)$ and $\mathbf{x}'_2 = (x_2^2, y_2^2, \sqrt{2}x_2 y_2)$. 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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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'

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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 $1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + \ldots + 6 \cdot \frac{1}{6} = 3.5$. 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 $x_1, \ldots, x_n$ of outcomes of the random process, the expected value is what we expect the *sample mean* $\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ 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 $\mathbb{E}\left[(X - \mathbb{E}[X])^2\right] = \sum_x (x - \mathbb{E}[X])^2 P(x)$ – this measures the spread of the distribution around the expected value. Notice that

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

We can similarly define the *sample variance* as $\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2$, which decomposes as $\frac{1}{n}\sum_{i=1}^{n} x_i^2 - \overline{x}^2$. You will sometimes see the sample variance defined as $\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$: 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 $\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y}) = \frac{1}{n}\sum_{i=1}^{n} x_i y_i - \overline{x}\,\overline{y}$. 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.

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press

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.

https://doi.org/10.1017/CBO9780511973000.003 Published online by Cambridge University Press