CHAPTER 2

# Preliminaries

This chapter has two purposes: to present some of the overarching important concepts of machine learning, and to see how some of the basic ideas of data processing and statistics arise in machine learning. One of the most useful ways to break down the effects of learning, which is to put it in terms of the statistical concepts of bias and variance, is given in Section 2.5, following on from a section where those concepts are introduced for the beginner.

## 2.1 SOME TERMINOLOGY

We start by considering some of the terminology that we will use throughout the book; we've already seen a bit of it in the Introduction. We will talk about inputs and input vectors for our learning algorithms. Likewise, we will talk about the outputs of the algorithm. The inputs are the data that is fed into the algorithm. In general, machine learning algorithms all work by taking a set of input values, producing an output (answer) for that input vector, and then moving on to the next input. The input vector will typically be several real numbers, which is why it is described as a vector: it is written down as a series of numbers, e.g., $(0.2, 0.45, 0.75, -0.3)$. The size of this vector, i.e., the number of elements in the vector, is called the dimensionality of the input. This is because if we were to plot the vector as a point, we would need one dimension of space for each of the different elements of the vector, so that the example above has 4 dimensions. We will talk about this more in Section 2.1.1.

We will often write equations in vector and matrix notation, with lowercase boldface letters being used for vectors and uppercase boldface letters for matrices. A vector $\mathbf{x}$ has elements $(x_1, x_2, \ldots, x_m)$. We will use the following notation in the book:

**Inputs** An input vector is the data given as one input to the algorithm. Written as $\mathbf{x}$, with elements $x_i$, where $i$ runs from 1 to the number of input dimensions, $m$.

**Weights** $w_{ij}$, are the weighted connections between nodes $i$ and $j$. For neural networks these weights are analogous to the synapses in the brain. They are arranged into a matrix $\mathbf{W}$.

**Outputs** The output vector is $\mathbf{y}$, with elements $y_j$, where $j$ runs from 1 to the number of output dimensions, $n$. We can write $\mathbf{y}(\mathbf{x}, \mathbf{W})$ to remind ourselves that the output depends on the inputs to the algorithm and the current set of weights of the network.

**Targets** The target vector $\mathbf{t}$, with elements $t_j$, where $j$ runs from 1 to the number of output dimensions, $n$, are the extra data that we need for supervised learning, since they provide the 'correct' answers that the algorithm is learning about.

**Activation Function** For neural networks, $g(\cdot)$ is a mathematical function that describes the firing of the neuron as a response to the weighted inputs, such as the threshold function described in Section 3.1.2.

**Error** $E$, a function that computes the inaccuracies of the network as a function of the outputs $\mathbf{y}$ and targets $\mathbf{t}$.

### 2.1.1 Weight Space

When working with data it is often useful to be able to plot it and look at it. If our data has only two or three input dimensions, then this is pretty easy: we use the $x$-axis for feature 1, the $y$-axis for feature 2, and the $z$-axis for feature 3. We then plot the positions of the input vectors on these axes. The same thing can be extended to as many dimensions as we like provided that we don't actually want to look at it in our 3D world. Even if we have 200 input dimensions (that is, 200 elements in each of our input vectors) then we can try to imagine it plotted by using 200 axes that are all mutually orthogonal (that is, at right angles to each other). One of the great things about computers is that they aren't constrained in the same way we are—ask a computer to hold a 200-dimensional array and it does it. Provided that you get the algorithm right (always the difficult bit!), then the computer doesn't know that 200 dimensions is harder than 2 for us humans.

We can look at projections of the data into our 3D world by plotting just three of the features against each other, but this is usually rather confusing: things can look very close together in your chosen three axes, but can be a very long way apart in the full set. You've experienced this in your 2D view of the 3D world; Figure 1.2 shows two different views of some wind turbines. The two turbines appear to be very close together from one angle, but are obviously separate from another.

As well as plotting datapoints, we can also plot anything else that we feel like. In particular, we can plot some of the parameters of a machine learning algorithm. This is particularly useful for neural networks (which we will start to see in the next chapter) since the parameters of a neural network are the values of a set of weights that connect the neurons to the inputs. There is a schematic of a neural network on the left of Figure 2.1, showing the inputs on the left, and the neurons on the right. If we treat the weights that get fed into one of the neurons as a set of coordinates in what is known as weight space, then we can plot them. We think about the weights that connect into a particular neuron, and plot the strengths of the weights by using one axis for each weight that comes into the neuron, and plotting the position of the neuron as the location, using the value of $w_1$ as the position on the 1st axis, the value of $w_2$ on the 2nd axis, etc. This is shown on the right of Figure 2.1.

Now that we have a space in which we can talk about how close together neurons and inputs are, since we can imagine positioning neurons and inputs in the same space by plotting the position of each neuron as the location where its weights say it should be. The two spaces will have the same dimension (providing that we don't use a bias node (see Section 3.3.2), otherwise the weight space will have one extra dimension) so we can plot the position of neurons in the input space. This gives us a different way of learning, since by changing the weights we are changing the location of the neurons in this weight space. We can measure distances between inputs and neurons by computing the Euclidean distance, which in two dimensions can be written as:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \tag{2.1}$$

So we can use the idea of neurons and inputs being 'close together' in order to decide

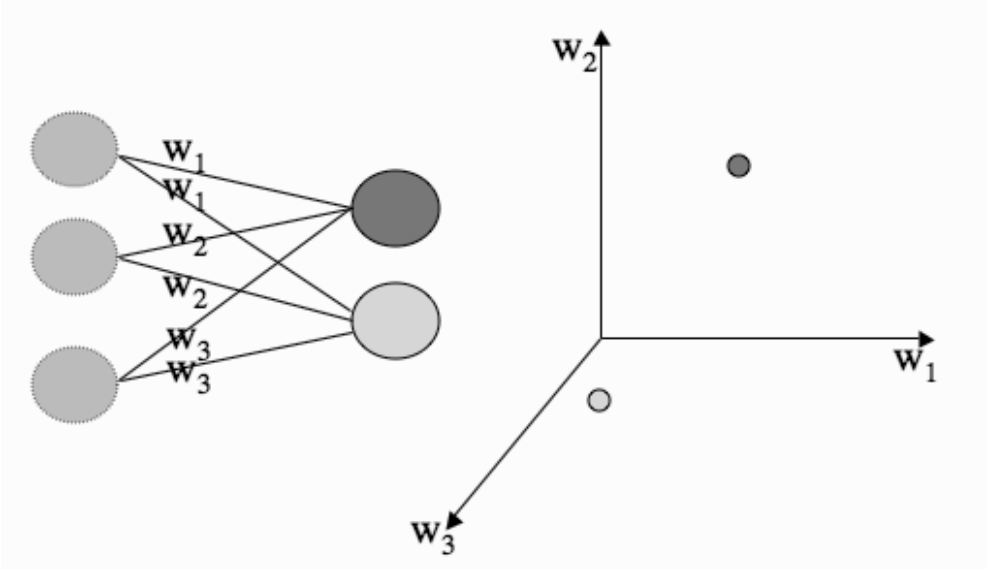


FIGURE 2.1 The position of two neurons in weight space. The labels on the network refer to the dimension in which that weight is plotted, not its value.

when a neuron should fire and when it shouldn't. If the neuron is close to the input in this sense then it should fire, and if it is not close then it shouldn't. This picture of weight space can be helpful for understanding another important concept in machine learning, which is what effect the number of input dimensions can have. The input vector is telling us everything we know about that example, and usually we don't know enough about the data to know what is useful and what is not (think back to the coin classification example in Section 1.4.2), so it might seem sensible to include all of the information that we can get, and let the algorithm sort out for itself what it needs. Unfortunately, we are about to see that doing this comes at a significant cost.

### 2.1.2 The Curse of Dimensionality

The curse of dimensionality is a very strong name, so you can probably guess that it is a bit of a problem. The essence of the curse is the realisation that as the number of dimensions increases, the volume of the unit hypersphere does not increase with it. The unit hypersphere is the region we get if we start at the origin (the centre of our coordinate system) and draw all the points that are distance 1 away from the origin. In 2 dimensions we get a circle of radius 1 around $(0, 0)$ (drawn in Figure 2.2), and in 3D we get a sphere around $(0, 0, 0)$ (Figure 2.3). In higher dimensions, the sphere becomes a hypersphere. The following table shows the size of the unit hypersphere for the first few dimensions, and the graph in Figure 2.4 shows the same thing, but also shows clearly that as the number of dimensions tends to infinity, so the volume of the hypersphere tends to zero.

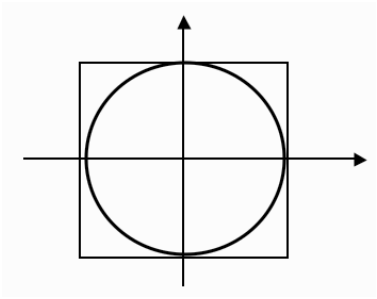

FIGURE 2.2 The unit circle in 2D with its bounding box.

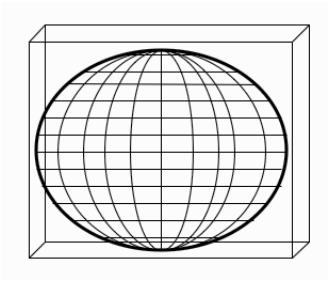

FIGURE 2.3 The unit sphere in 3D with its bounding cube. The sphere does not reach as far into the corners as the circle does, and this gets more noticeable as the number of dimensions increases.

| Dimension | Volume |
|---|---|
| 1 | 2.0000 |
| 2 | 3.1416 |
| 3 | 4.1888 |
| 4 | 4.9348 |
| 5 | 5.2636 |
| 6 | 5.1677 |
| 7 | 4.7248 |
| 8 | 4.0587 |
| 9 | 3.2985 |
| 10 | 2.5502 |

At first sight this seems completely counterintuitive. However, think about enclosing the hypersphere in a box of width 2 (between -1 and 1 along each axis), so that the box just touches the sides of the hypersphere. For the circle, almost all of the area inside the box is included in the circle, except for a little bit at each corner (see Figure 2.2) The same is true in 3D (Figure 2.3), but if we think about the 100-dimensional hypersphere (not necessarily something you want to imagine), and follow the diagonal line from the origin out to one of the corners of the box, then we intersect the boundary of the hypersphere when all the coordinates are 0.1. The remaining 90% of the line inside the box is outside the hypersphere, and so the volume of the hypersphere is obviously shrinking as the number of dimensions grows. The graph in Figure 2.4 shows that when the number of dimensions is above about 20, the volume is effectively zero. It was computed using the formula for the volume of the hypersphere of dimension $n$ as $v_n = (2\pi/n)v_{n-2}$. So as soon as $n > 2\pi$, the volume starts to shrink.

The curse of dimensionality will apply to our machine learning algorithms because as the number of input dimensions gets larger, we will need more data to enable the algorithm to generalise sufficiently well. Our algorithms try to separate data into classes based on the features; therefore as the number of features increases, so will the number of datapoints we need. For this reason, we will often have to be careful about what information we give to the algorithm, meaning that we need to understand something about the data in advance.

Regardless of how many input dimensions there are, the point of machine learning is to

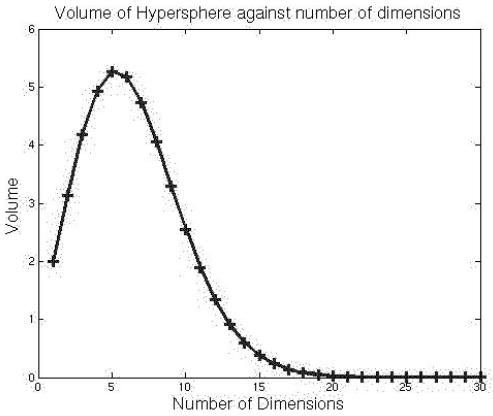


FIGURE 2.4 The volume of the unit hypersphere for different numbers of dimensions.

make predictions on data inputs. In the next section we consider how to evaluate how well an algorithm actually achieves this.

## 2.2 KNOWING WHAT YOU KNOW: TESTING MACHINE LEARNING ALGORITHMS

The purpose of learning is to get better at predicting the outputs, be they class labels or continuous regression values. The only real way to know how successfully the algorithm has learnt is to compare the predictions with known target labels, which is how the training is done for supervised learning. This suggests that one thing you can do is just to look at the error that the algorithm makes on the training set.

However, we want the algorithms to generalise to examples that were not seen in the training set, and we obviously can't test this by using the training set. So we need some different data, a test set, to test it on as well. We use this test set of (input, target) pairs by feeding them into the network and comparing the predicted output with the target, but we don't modify the weights or other parameters for them: we use them to decide how well the algorithm has learnt. The only problem with this is that it reduces the amount of data that we have available for training, but that is something that we will just have to live with.

### 2.2.1 Overfitting

Unfortunately, things are a little bit more complicated than that, since we might also want to know how well the algorithm is generalising as it learns: we need to make sure that we do enough training that the algorithm generalises well. In fact, there is at least as much danger in over-training as there is in under-training. The number of degrees of variability in most machine learning algorithms is huge — for a neural network there are lots of weights, and each of them can vary. This is undoubtedly more variation than there is in the function we are learning, so we need to be careful: if we train for too long, then we will overfit the data, which means that we have learnt about the noise and inaccuracies in the data as well as the actual function. Therefore, the model that we learn will be much too complicated, and won't be able to generalise.

Figure 2.5 shows this by plotting the predictions of some algorithm (as the curve) at

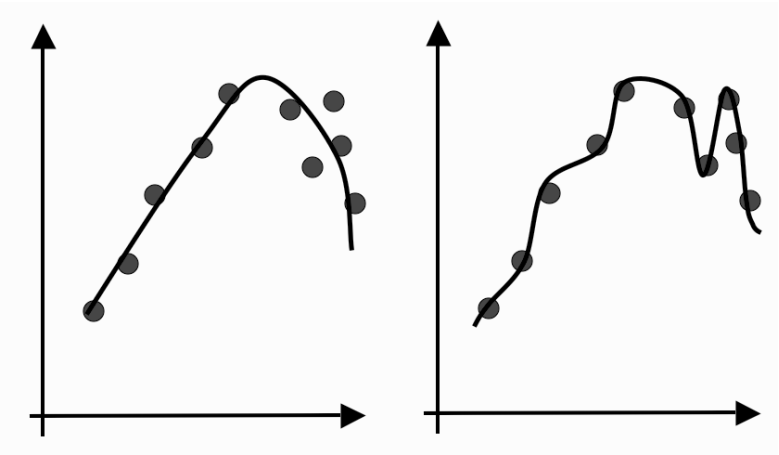

FIGURE 2.5 The effect of overfitting is that rather than finding the generating function (as shown on the left), the neural network matches the inputs perfectly, including the noise in them (on the right). This reduces the generalisation capabilities of the network.

two different points in the learning process. On the left of the figure the curve fits the overall trend of the data well (it has generalised to the underlying general function), but the training error would still not be that close to zero since it passes near, but not through, the training data. As the network continues to learn, it will eventually produce a much more complex model that has a lower training error (close to zero), meaning that it has memorised the training examples, including any noise component of them, so that is has overfitted the training data.

We want to stop the learning process before the algorithm overfits, which means that we need to know how well it is generalising at each timestep. We can't use the training data for this, because we wouldn't detect overfitting, but we can't use the testing data either, because we're saving that for the final tests. So we need a third set of data to use for this purpose, which is called the validation set because we're using it to validate the learning so far. This is known as cross-validation in statistics. It is part of model selection: choosing the right parameters for the model so that it generalises as well as possible.

### 2.2.2 Training, Testing, and Validation Sets

We now need three sets of data: the training set to actually train the algorithm, the validation set to keep track of how well it is doing as it learns, and the test set to produce the final results. This is becoming expensive in data, especially since for supervised learning it all has to have target values attached (and even for unsupervised learning, the validation and test sets need targets so that you have something to compare to), and it is not always easy to get accurate labels (which may well be why you want to learn about the data). The area of semi-supervised learning attempts to deal with this need for large amounts of labelled data; see the Further Reading section for some references.

Clearly, each algorithm is going to need some reasonable amount of data to learn from (precise needs vary, but the more data the algorithm sees, the more likely it is to have seen examples of each possible type of input, although more data also increases the computational time to learn). However, the same argument can be used to argue that the validation and

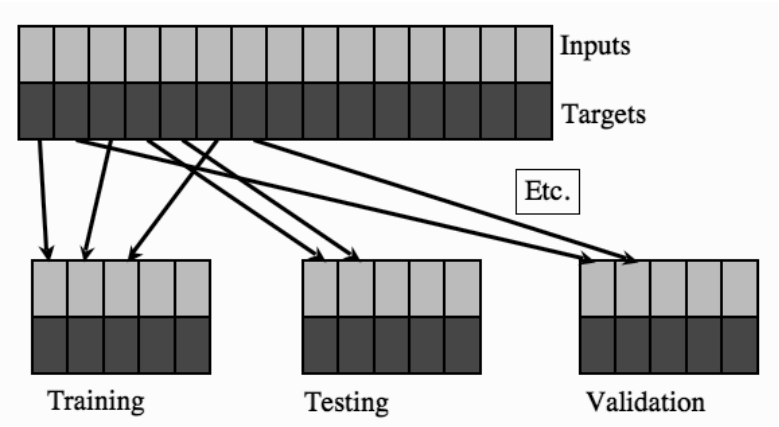


FIGURE 2.6 The dataset is split into different sets, some for training, some for validation, and some for testing.

test sets should also be reasonably large. Generally, the exact proportion of training to testing to validation data is up to you, but it is typical to do something like 50:25:25 if you have plenty of data, and 60:20:20 if you don't. How you do the splitting can also matter. Many datasets are presented with the first set of datapoints being in class 1, the next in class 2, and so on. If you pick the first few points to be the training set, the next the test set, etc., then the results are going to be pretty bad, since the training did not see all the classes. This can be dealt with by randomly reordering the data first, or by assigning each datapoint randomly to one of the sets, as is shown in Figure 2.6.

If you are really short of training data, so that if you have a separate validation set there is a worry that the algorithm won't be sufficiently trained; then it is possible to perform leave-some-out, multi-fold cross-validation. The idea is shown in Figure 2.7. The dataset is randomly partitioned into $K$ subsets, and one subset is used as a validation set, while the algorithm is trained on all of the others. A different subset is then left out and a new model is trained on that subset, repeating the same process for all of the different subsets. Finally, the model that produced the lowest validation error is tested and used. We've traded off data for computation time, since we've had to train $K$ different models instead of just one. In the most extreme case of this there is leave-one-out cross-validation, where the algorithm is validated on just one piece of data, training on all of the rest.

### 2.2.3 The Confusion Matrix

Regardless of how much data we use to test the trained algorithm, we still need to work out whether or not the result is good. We will look here at a method that is suitable for classification problems that is known as the confusion matrix. For regression problems things are more complicated because the results are continuous, and so the most common thing to use is the sum-of-squares error that we will use to drive the training in the following chapters. We will see these methods being used as we look at examples.

The confusion matrix is a nice simple idea: make a square matrix that contains all the possible classes in both the horizontal and vertical directions and list the classes along the top of a table as the predicted outputs, and then down the left-hand side as the targets. So for example, the element of the matrix at $(i, j)$ tells us how many input patterns were put

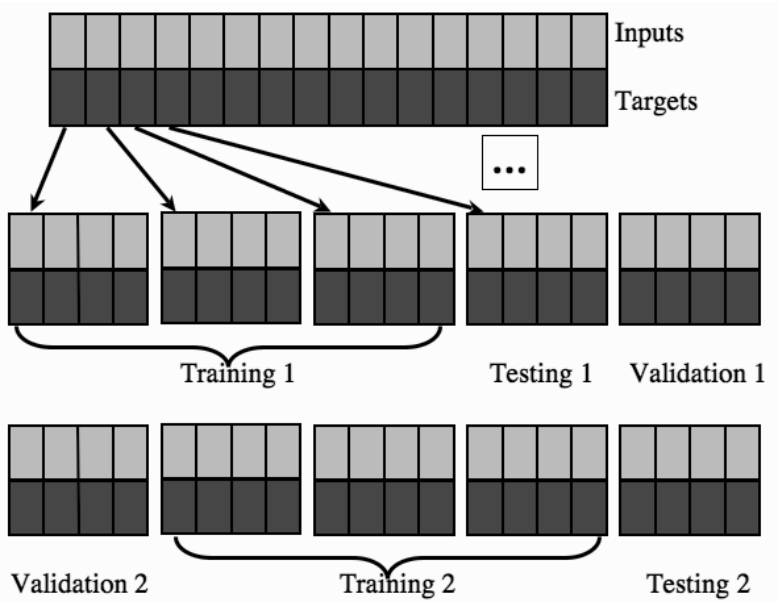


FIGURE 2.7 Leave-some-out, multi-fold cross-validation gets around the problem of data shortage by training many models. It works by splitting the data into sets, training a model on most sets and holding one out for validation (and another for testing). Different models are trained with different sets being held out.

into class $i$ in the targets, but class $j$ by the algorithm. Anything on the leading diagonal (the diagonal that starts at the top left of the matrix and runs down to the bottom right) is a correct answer. Suppose that we have three classes: $C_1, C_2$, and $C_3$. Now we count the number of times that the output was class $C_1$ when the target was $C_1$, then when the target was $C_2$, and so on until we've filled in the table:

| | Outputs | | |
|---|---|---|---|
| | $C_1$ | $C_2$ | $C_3$ |
| $C_1$ | 5 | 1 | 0 |
| $C_2$ | 1 | 4 | 1 |
| $C_3$ | 2 | 0 | 4 |

This table tells us that, for the three classes, most examples were classified correctly, but two examples of class $C_3$ were misclassified as $C_1$, and so on. For a small number of classes this is a nice way to look at the outputs. If you just want one number, then it is possible to divide the sum of the elements on the leading diagonal by the sum of all of the elements in the matrix, which gives the fraction of correct responses. This is known as the accuracy, and we are about to see that it is not the last word in evaluating the results of a machine learning algorithm.

### 2.2.4 Accuracy Metrics

We can do more to analyse the results than just measuring the accuracy. If you consider the possible outputs of the classes, then they can be arranged in a simple chart like this

(where a true positive is an observation correctly put into class 1, while a false positive is an observation incorrectly put into class 1, while negative examples (both true and false) are those put into class 2):

| True Positives | False Positives |
|---|---|
| False Negatives | True Negatives |

The entries on the leading diagonal of this chart are correct and those off the diagonal are wrong, just as with the confusion matrix. Note, however, that this chart and the concepts of false positives, etc., are based on binary classification.

Accuracy is then defined as the sum of the number of true positives and true negatives divided by the total number of examples (where # means 'number of', and TP stands for True Positive, etc.):

$$\text{Accuracy} = \frac{\#TP + \#FP}{\#TP + \#FP + \#TN + \#FN}. \tag{2.2}$$

The problem with accuracy is that it doesn't tell us everything about the results, since it turns four numbers into just one. There are two complementary pairs of measurements that can help us to interpret the performance of a classifier, namely sensitivity and specificity, and precision and recall. Their definitions are shown next, followed by some explanation.

$$\text{Sensitivity} = \frac{\#TP}{\#TP + \#FN} \tag{2.3}$$

$$\text{Specificity} = \frac{\#TN}{\#TN + \#FP} \tag{2.4}$$

$$\text{Precision} = \frac{\#TP}{\#TP + \#FP} \tag{2.5}$$

$$\text{Recall} = \frac{\#TP}{\#TP + \#FN} \tag{2.6}$$

Sensitivity (also known as the true positive rate) is the ratio of the number of correct positive examples to the number classified as positive, while specificity is the same ratio for negative examples. Precision is the ratio of correct positive examples to the number of actual positive examples, while recall is the ratio of the number of correct positive examples out of those that were classified as positive, which is the same as sensitivity. If you look at the chart again you can see that sensitivity and specificity sum the columns for the denominator, while precision and recall sum the first column and the first row, and so miss out some information about how well the learner does on the negative examples.

Together, either of these pairs of measures gives more information than just the accuracy. If you consider precision and recall, then you can see that they are to some extent inversely related, in that if the number of false positives increases (meaning that the algorithm is using a broader definition of that class), then the number of false negatives often decreases, and vice versa. They can be combined to give a single measure, the $F_1$ measure, which can be written in terms of precision and recall as:

$$F_1 = 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{2.7}$$

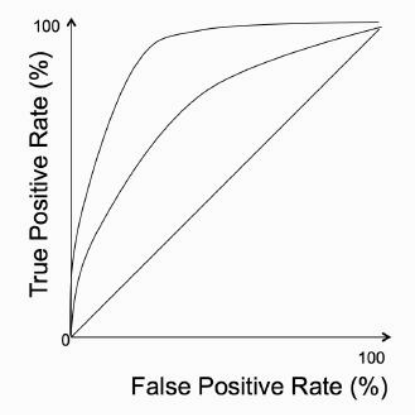


FIGURE 2.8 An example of an ROC curve. The diagonal line represents exactly chance, so anything above the line is better than chance, and the further from the line, the better. Of the two curves shown, the one that is further away from the diagonal line would represent a more accurate method.

and in terms of the numbers of false positives, etc. (from which it can be seen that it computes the mean of the false examples) as:

$$F_1 = \frac{\#TP}{\#TP + (\#FN + \#FP)/2}. \tag{2.8}$$

### 2.2.5 The Receiver Operator Characteristic (ROC) Curve

Since we can use these measures to evaluate a particular classifier, we can also compare classifiers – either the same classifier with different learning parameters, or completely different classifiers. In this case, the Receiver Operator Characteristic curve (almost always known just as the ROC curve) is useful. This is a plot of the percentage of true positives on the $y$ axis against false positives on the $x$ axis; an example is shown in Figure 2.8. A single run of a classifier produces a single point on the ROC plot, and a perfect classifier would be a point at $(0, 1)$ (100% true positives, 0% false positives), while the anti-classifier that got everything wrong would be at (1,0); so the closer to the top-left-hand corner the result of a classifier is, the better the classifier has performed. Any classifier that sits on the diagonal line from (0,0) to (1,1) behaves exactly at the chance level (assuming that the positive and negative classes are equally common) and so presumably a lot of learning effort is wasted since a fair coin would do just as well.

In order to compare classifiers, or choices of parameters settings for the same classifier, you could just compute the point that is furthest from the 'chance' line along the diagonal. However, it is normal to compute the area under the curve (AUC) instead. If you only have one point for each classifier, the curve is the trapezoid that runs from (0,0) up to the point and then from there to (1,1). If there are more points (based on more runs of the classifier, such as trained and/or tested on different datasets), then they are just included in order along the diagonal line.

The key to getting a curve rather than a point on the ROC curve is to use cross-validation. If you use 10-fold cross-validation, then you have 10 classifiers, with 10 different

test sets, and you also have the 'ground truth' labels. The true labels can be used to produce a ranked list of the different cross-validation-trained results, which can be used to specify a curve through the 10 datapoints on the ROC curve that correspond to the results of this classifier. By producing an ROC curve for each classifier it is possible to compare their results.

### 2.2.6 Unbalanced Datasets

Note that for the accuracy we have implicitly assumed that there are the same number of positive and negative examples in the dataset (which is known as a balanced dataset). However, this is often not true (this can potentially cause problems for the learners as well, as we shall see later in the book). In the case where it is not, we can compute the balanced accuracy as the sum of sensitivity and specificity divided by 2. However, a more correct measure is Matthew's Correlation Coefficient, which is computed as:

$$MCC = \frac{\#TP \times \#TN - \#FP \times \#FN}{\sqrt{(\#TP + \#FP)(\#TP + \#FN)(\#TN + \#FP)(\#TN + \#FN)}} \tag{2.9}$$

If any of the brackets in the denominator are 0, then the whole of the denominator is set to 1. This provides a balanced accuracy computation.

As a final note on these methods of evaluation, if there are more than two classes and it is useful to distinguish the different types of error, then the calculations get a little more complicated, since instead of one set of false positives and one set of false negatives, you have some for each class. In this case, specificity and recall are not the same. However, it is possible to create a set of results, where you use one class as the positives and everything else as the negatives, and repeat this for each of the different classes.

### 2.2.7 Measurement Precision

There is a different way to evaluate the accuracy of a learning system, which unfortunately also uses the word precision, although with a different meaning. The concept here is to treat the machine learning algorithm as a measurement system. We feed in inputs and look at the outputs that we get. Even before comparing them to the target values, we can measure something about the algorithm: if we feed in a set of similar inputs, then we would expect to get similar outputs for them. This measure of the variability of the algorithm is also known as precision, and it tells us how repeatable the predictions that the algorithm makes are. It might be useful to think of precision as being something like the variance of a probability distribution: it tells you how much spread around the mean to expect.

The point is that just because an algorithm is precise it does not mean that it is accurate – it can be precisely wrong if it always gives the wrong prediction. One measure of how well the algorithm's predictions match reality is known as trueness, and it can be defined as the average distance between the correct output and the prediction. Trueness doesn't usually make much sense for classification problems unless there is some concept of certain classes being similar to each other. Figure 2.9 illustrates the idea of trueness and precision in the traditional way: as a darts game, with four examples with varying trueness and precision for the three darts thrown by a player.

This section has considered the endpoint of machine learning, looking at the outputs, and thinking about what we need to do with the input data in terms of having multiple datasets, etc. In the next section we return to the starting point and consider how we can start analysing a dataset by dealing with probabilities.

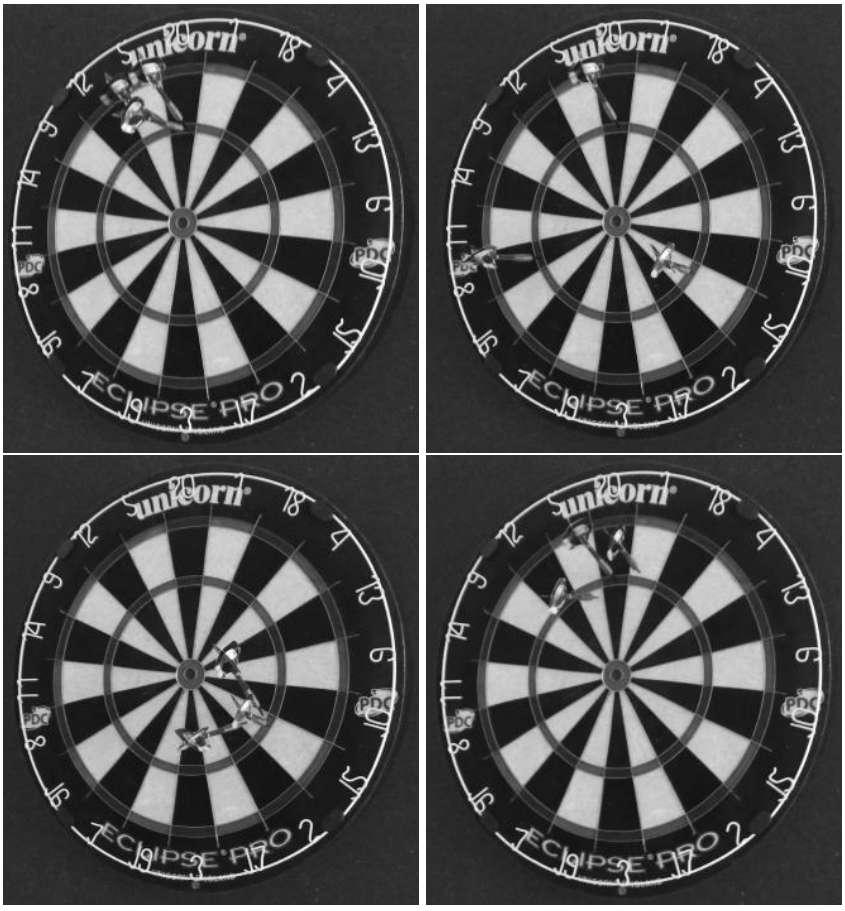


FIGURE 2.9 Assuming that the player was aiming for the highest-scoring triple 20 in darts (the segments each score the number they are labelled with, the narrow band on the outside of the circle scores double and the narrow band halfway in scores triple; the outer and inner 'bullseye' at the centre score 25 and 50, respectively), these four pictures show different outcomes. *Top left:* very accurate: high precision and trueness, *top right:* low precision, but good trueness, *bottom left:* high precision, but low trueness, and *bottom right:* reasonable trueness and precision, but the actual outputs are not very good. *(Thanks to Stefan Nowicki, whose dartboard was used for these pictures.)*

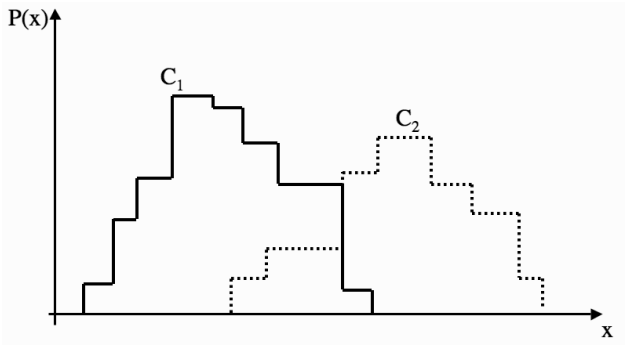


FIGURE 2.10 A histogram of feature values ($x$) against their probability for two classes.

## 2.3 TURNING DATA INTO PROBABILITIES

Take a look at the plot in Figure 2.10. It shows the measurements of some feature $x$ for two classes, $C_1$ and $C_2$. Members of class $C_2$ tend to have larger values of feature $x$ than members of class $C_1$, but there is some overlap between the two classes. The correct class is fairly easy to predict at the extremes of the range, but what to do in the middle is unclear. Suppose that we are trying to classify writing of the letters 'a' and 'b' based on their height (as shown in Figure 2.11). Most people write their 'a's smaller than their 'b's, but not everybody. However, in this example, we have a secret weapon. We know that in English text, the letter 'a' is much more common than the letter 'b' (we called this an unbalanced dataset earlier). If we see a letter that is either an 'a' or a 'b' in normal writing, then there is a 75% chance that it is an 'a.' We are using prior knowledge to estimate the probability that the letter is an 'a': in this example, $P(C_1) = 0.75, P(C_2) = 0.25$. If we weren't allowed to see the letter at all, and just had to classify it, then if we picked 'a' every time, we'd be right 75% of the time.

However, when we are asked to make a classification we are also given the value of $x$. It would be pretty silly to just use the value of $P(C_1)$ and ignore the value of $x$ if it might help! In fact, we are given a training set of values of $x$ and the class that each exemplar belongs to. This lets us calculate the value of $P(C_1)$ (we just count how many times out of the total the class was $C_1$ and divide by the total number of examples), and also another useful measurement: the conditional probability of $C_1$ given that $x$ has value $X$: $P(C_1|X)$ . The conditional probability tells us how likely it is that the class is $C_1$ given that the value of $x$ is $X$. So in Figure 2.10 the value of $P(C_1|X)$ will be much larger for small values of $X$ than for large values. Clearly, this is exactly what we want to calculate in order to perform classification. The question is how to get to this conditional probability, since we can't read it directly from the histogram.

The first thing that we need to do to get these values is to quantise the measurement $x$, which just means that we put it into one of a discrete set of values $\{X\}$, such as the bins in a histogram. This is exactly what is plotted in Figure 2.10. Now, if we have lots of examples of the two classes, and the histogram bins that their measurements fall into, we can compute $P(C_i, X_j)$, which is the joint probability, and tells us how often a measurement of $C_i$ fell into histogram bin $X_j$. We do this by looking in histogram bin $X_j$, counting the number of examples of class $C_i$ that are in it, and dividing by the total number of examples (of any class).

We can also define $P(X_j|C_i)$, which is a different conditional probability, and tells us

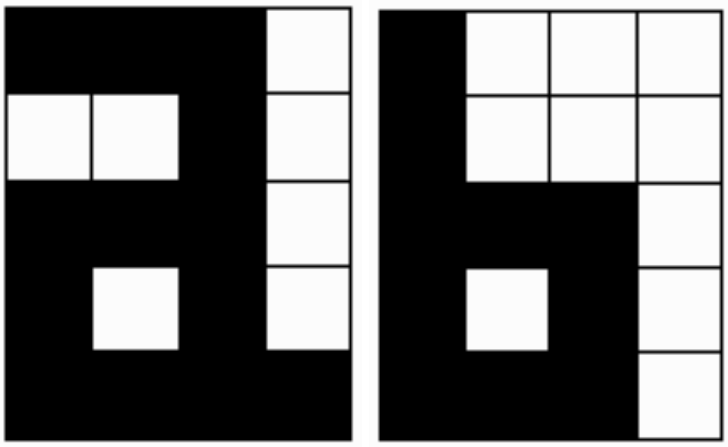

FIGURE 2.11 The letters 'a' and 'b' in pixel form.

how often (in the training set) there is a measurement of $X_j$ given that the example is a member of class $C_i$. Again, we can just get this information from the histogram by counting the number of examples of class $C_i$ in histogram bin $X_j$ and dividing by the number of examples of that class there are (in any bin). Hopefully, this has just been revision for you from a statistics course at some stage; if not, and you don't follow it, get hold of any introductory probability book.

So we have now worked out two things from our training data: the joint probability $P(C_i, X_j)$ and the conditional probability $P(X_j|C_i)$. Since we actually want to compute $P(C_i|X_j)$ we need to know how to link these things together. As some of you may already know, the answer is Bayes' rule, which is what we are now going to derive. There is a link between the joint probability and the conditional probability. It is:

$$P(C_i, X_j) = P(X_j|C_i)P(C_i), \tag{2.10}$$

or equivalently:

$$P(C_i, X_j) = P(C_i|X_j)P(X_j). \tag{2.11}$$

Clearly, the right-hand side of these two equations must be equal to each other, since they are both equal to $P(C_i, X_j)$, and so with one division we can write:

$$P(C_i|X_j) = \frac{P(X_j|C_i)P(C_i)}{P(X_j)}. \tag{2.12}$$

This is Bayes' rule. If you don't already know it, learn it: it is the most important equation in machine learning. It relates the posterior probability $P(C_i|X_j)$ with the prior probability $P(C_i)$ and class-conditional probability $P(X_j|C_i)$. The denominator (the term on the bottom of the fraction) acts to normalise everything, so that all the probabilities sum to 1. It might not be clear how to compute this term. However, if we notice that any observation $X_k$ has to belong to some class $C_i$, then we can marginalise over the classes to compute:

$$P(X_k) = \sum_i P(X_k|C_i)P(C_i). \tag{2.13}$$

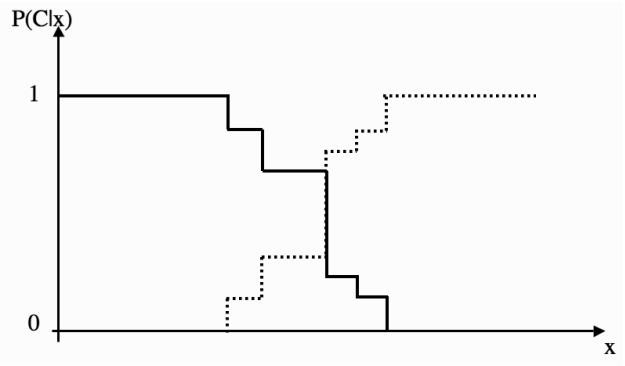


FIGURE 2.12 The posterior probabilities of the two classes $C_1$ and $C_2$ for feature $x$.

The reason why Bayes' rule is so important is that it lets us obtain the posterior probability—which is what we actually want—by calculating things that are much easier to compute. We can estimate the prior probabilities by looking at how often each class appears in our training set, and we can get the class-conditional probabilities from the histogram of the values of the feature for the training set. We can use the posterior probability (Figure 2.12) to assign each new observation to one of the classes by picking the class $C_i$ where:

$$P(C_i|\mathbf{x}) > P(C_j|\mathbf{x}) \quad \forall \; i \neq j, \tag{2.14}$$

where $\mathbf{x}$ is a vector of feature values instead of just one feature. This is known as the maximum a posteriori or MAP hypothesis, and it gives us a way to choose which class to choose as the output one. The question is whether this is the right thing to do. There has been quite a lot of research in both the statistical and machine learning literatures into what is the right question to ask about our data to perform classification, but we are going to skate over it very lightly.

The MAP question is what is the most likely class given the training data? Suppose that there are three possible output classes, and for a particular input the posterior probabilities of the classes are $P(C_1|\mathbf{x}) = 0.35$, $P(C_2|\mathbf{x}) = 0.45$, $P(C_3|\mathbf{x}) = 0.2$. The MAP hypothesis therefore tells us that this input is in class $C_2$, because that is the class with the highest posterior probability. Now suppose that, based on the class that the data is in, we want to do something. If the class is $C_1$ or $C_3$ then we do action 1, and if the class is $C_2$ then we do action 2. As an example, suppose that the inputs are the results of a blood test, the three classes are different possible diseases, and the output is whether or not to treat with a particular antibiotic. The MAP method has told us that the output is $C_2$, and so we will not treat the disease. But what is the probability that it does not belong to class $C_2$, and so should have been treated with the antibiotic? It is $1 - P(C_2) = 0.55$. So the MAP prediction seems to be wrong: we should treat with antibiotic, because overall it is more likely. This method where we take into account the final outcomes of all of the classes is called the Bayes' Optimal Classification. It minimises the probability of misclassification, rather than maximising the posterior probability.

### 2.3.1 Minimising Risk

In the medical example we just saw it made sense to classify based on minimising the probability of misclassification. We can also consider the risk that is involved in the misclassification. The risk from misclassifying someone as unhealthy when they are healthy is usually smaller than the other way around, but not necessarily always: there are plenty of treatments that have nasty side effects, and you wouldn't want to suffer from those if you didn't have the disease. In cases like this we can create a loss matrix that specifies the risk involved in classifying an example of class $C_i$ as class $C_j$. It looks like the confusion matrix we saw in Section 2.2, except that a loss matrix always contains zeros on the leading diagonal since there should never be a loss from getting the classification correct! Once we have the loss matrix, we just extend our classifier to minimise risk by multiplying each case by the relevant loss number.

### 2.3.2 The Naïve Bayes' Classifier

We're now going to return to performing classification, without worrying about the outcomes, so that we are back to calculating the MAP outcome, Equation (2.14). We can compute this exactly as described above, and it will work fine. However, suppose that the vector of feature values had many elements, so that there were lots of different features that were measured. How would this affect the classifier? We are trying to estimate $P(\mathbf{X}_j|C_i) = P(X_j^1, X_j^2, \ldots X_j^n|C_i)$ (where the superscripts index the elements of the vector) by looking at the histogram of all of our training data. As the dimensionality of $\mathbf{X}$ increases (as $n$ gets larger), the amount of data in each bin of the histogram shrinks. This is the curse of dimensionality again (Section 2.1.2), and means that we need much more data as the dimensionality increases.

There is one simplifying assumption that we can make. We can assume that the elements of the feature vector are conditionally independent of each other, given the classification. So given the class $C_i$, the values of the different features do not affect each other. This is the naïveté in the name of the classifier, since it often doesn't make much sense—it tells us that the features are independent of each other. If we were to try to classify coins it would say that the weight and the diameter of the coin are independent of each other, which clearly isn't true. However, it does mean that the probability of getting the string of feature values $P(X_j^1 = a_1, X_j^2 = a_2, \ldots, X_j^n = a_n|C_i)$ is just equal to the product of multiplying together all of the individual probabilities:

$$P(X_j^1 = a_1|C_i) \times P(X_j^2 = a_2|C_i) \times \ldots \times P(X_j^n = a_n|C_i) = \prod_k P(X_j^k = a_k|C_i), \quad (2.15)$$

which is much easier to compute, and reduces the severity of the curse of dimensionality. So the classifier rule for the naïve Bayes' classifier is to select the class $C_i$ for which the following computation is the maximum:

$$P(C_i) \prod_k P(X_j^k = a_k|C_i). \quad (2.16)$$

This is clearly a great simplification over evaluating the full probability, so it might come as a surprise that the naïve Bayes' classifier has been shown to have comparable results to other classification methods in certain domains. Where the simplification is true, so that the features are conditionally independent of each other, the naïve Bayes' classifier produces exactly the MAP classification.

In Chapter 12 on learning with trees, particularly Section 12.4, there is an example concerned with what to do in the evening based on whether you have an assignment deadline and what is happening. The data, shown below, consists of a set of prior examples from the last few days.

| Deadline? | Is there a party? | Lazy? | Activity |
|---|---|---|---|
| Urgent | Yes | Yes | Party |
| Urgent | No | Yes | Study |
| Near | Yes | Yes | Party |
| None | Yes | No | Party |
| None | No | Yes | Pub |
| None | Yes | No | Party |
| Near | No | No | Study |
| Near | No | Yes | TV |
| Near | Yes | Yes | Party |
| Urgent | No | No | Study |

In Chapter 12 we will see the results of a decision tree learning about this data, but here we will use the naïve Bayes' classifier. We feed in the current values for the feature variables (deadline, whether there is a party, etc.) and ask the classifier to compute the probabilities of each of the four possible things that you might do in the evening based on the data in the training set. Then we pick the most likely class. Note that the probabilities will be very small. This is one of the problems with the Bayes' classifier: since we are multiplying lots of probabilities, which are all less than one, the numbers get very small.

Suppose that you have deadlines looming, but none of them are particularly urgent, that there is no party on, and that you are currently lazy. Then the classifier needs to evaluate:

- P(Party) × P(Near | Party) × P(No Party | Party) × P(Lazy | Party)
- P(Study) × P(Near | Study) × P(No Party | Study) × P(Lazy | Study)
- P(Pub) × P(Near | Pub) × P(No Party | Pub) × P(Lazy | Pub)
- P(TV) × P(Near | TV) × P(No Party | TV) × P(Lazy | TV)

Using the data above these evaluate to:

$$\begin{aligned} P(\text{Party}|\text{near (not urgent) deadline, no party, lazy}) &= \frac{5}{10} \times \frac{2}{5} \times \frac{0}{5} \times \frac{3}{5} \\ &= 0 \qquad (2.17) \\ P(\text{Study}|\text{near (not urgent) deadline, no party, lazy}) &= \frac{3}{10} \times \frac{1}{3} \times \frac{3}{3} \times \frac{1}{3} \\ &= \frac{1}{30} \qquad (2.18) \\ P(\text{Pub}|\text{near (not urgent) deadline, no party, lazy}) &= \frac{1}{10} \times \frac{0}{1} \times \frac{1}{1} \times \frac{1}{1} \\ &= 0 \qquad (2.19) \\ P(\text{TV}|\text{near (not urgent) deadline, no party, lazy}) &= \frac{1}{10} \times \frac{1}{1} \times \frac{1}{1} \times \frac{1}{1} \\ &= \frac{1}{10} \qquad (2.20) \end{aligned}$$

So based on this you will be watching TV tonight.

## 2.4 SOME BASIC STATISTICS

This section will provide a quick summary of a few important statistical concepts. You may well already know about them, but just in case we'll go over them, highlighting the points that are important for machine learning. Any basic statistics book will give considerably more detailed information.

### 2.4.1 Averages

We'll start as basic as can be, with the two numbers that can be used to characterise a dataset: the mean and the variance. The mean is easy, it is the most commonly used average of a set of data, and is the value that is found by adding up all the points in the dataset and dividing by the number of points. There are two other averages that are used: the median and the mode. The median is the middle value, so the most common way to find it is to sort the dataset according to size and then find the point that is in the middle (of course, if there is an even number of datapoints then there is no exact middle, so people typically take the value halfway between the two points that are closest to the middle). There is a faster algorithm for computing the median based on a randomised algorithm that is described in most textbooks on algorithms. The mode is the most common value, so it just requires counting how many times each element appears and picking the most frequent one. We will also need to develop the idea of variance within a dataset, and of probability distributions.

### 2.4.2 Variance and Covariance

If we are given a set of random numbers, then we already know how to compute the mean of the set, together with the median. However, there are other useful statistics that can be computed, one of which is the expectation. The name expectation shows the gambling roots of most probability theory, since it describes the amount of money you can expect to win. It consists of multiplying together the payoff for each possibility with the probability of that thing happening, and then adding them all together. So if you are approached in the street by somebody selling raffle tickets for $1 and they tell you that there is a prize of $100,000 and they are selling 200,000 tickets, then you can work out the expected value of your ticket as:

$$E = -1 \times \frac{199,999}{200,000} + 99,999 \times \frac{1}{200,000} = -0.5, \tag{2.21}$$

where the -1 is the price of your ticket, which does not win 199,999 times out of 200,000 and the 99,999 is the prize minus the cost of your ticket. Note that the expected value is not a real value: you will never actually get 50 cents back, no matter what happens. If we just compute the expected value of a set of numbers, then we end up with the mean value.

The variance of the set of numbers is a measure of how spread out the values are. It is computed as the sum of the squared distances between each element in the set and the expected value of the set (the mean, $\boldsymbol{\mu}$):

$$\text{var}(\{\mathbf{x}_i\}) = \sigma^2(\{\mathbf{x}_i\}) = E((\{\mathbf{x}_i\} - \boldsymbol{\mu})^2) = \sum_{i=1}^{N} (\mathbf{x}_i - \boldsymbol{\mu})^2. \tag{2.22}$$

The square root of the variance, $\sigma$, is known as the standard deviation. The variance looks at the variation in one variable compared to its mean. We can generalise this to look at how two variables vary together, which is known as the covariance. It is a measure of how dependent the two variables are (in the statistical sense). It is computed by:

$$\text{cov}(\{\mathbf{x}_i\}, \{\mathbf{y}_i\}) = E(\{\mathbf{x}_i\} - \boldsymbol{\mu})E(\{\mathbf{y}_i\} - \boldsymbol{\nu}), \tag{2.23}$$

where $\boldsymbol{\nu}$ is the mean of set $\{\mathbf{y}_i\}$. If two variables are independent, then the covariance is 0 (the variables are then known as uncorrelated), while if they both increase and decrease at the same time, then the covariance is positive, and if one goes up while the other goes down, then the covariance is negative.

The covariance can be used to look at the correlation between all pairs of variables within a set of data. We need to compute the covariance of each pair, and these are then put together into what is imaginatively known as the covariance matrix. It can be written as:

$$\boldsymbol{\Sigma} = \begin{pmatrix} E[(\mathbf{x}_1 - \boldsymbol{\mu}_1)(\mathbf{x}_1 - \boldsymbol{\mu}_1)] & E[(\mathbf{x}_1 - \boldsymbol{\mu}_1)(\mathbf{x}_2 - \boldsymbol{\mu}_2)] & \dots & E[(\mathbf{x}_1 - \boldsymbol{\mu}_1)(\mathbf{x}_n - \mu_n)] \\ E[(\mathbf{x}_2 - \boldsymbol{\mu}_2)(\mathbf{x}_1 - \boldsymbol{\mu}_1)] & E[(\mathbf{x}_2 - \boldsymbol{\mu}_2)(\mathbf{x}_2 - \boldsymbol{\mu}_2)] & \dots & E[(\mathbf{x}_2 - \boldsymbol{\mu}_2)(\mathbf{x}_n - \boldsymbol{\mu}_n)] \\ \dots & \dots & \dots & \dots \\ E[(\mathbf{x}_n - \boldsymbol{\mu}_n)(\mathbf{x}_1 - \boldsymbol{\mu}_1)] & E[(\mathbf{x}_n - \boldsymbol{\mu}_n)(\mathbf{x}_2 - \boldsymbol{\mu}_2)] & \dots & E[(\mathbf{x}_n - \boldsymbol{\mu}_n)(\mathbf{x}_n - \boldsymbol{\mu}_n)] \end{pmatrix} \tag{2.24}$$

where $\mathbf{x}_i$ is a column vector describing the elements of the $i$th variable, and $\boldsymbol{\mu}_i$ is their mean. Note that the matrix is square, that the elements on the leading diagonal of the matrix are equal to the variances, and that it is symmetric since $\text{cov}(\mathbf{x}_i, \mathbf{x}_j) = \text{cov}(\mathbf{x}_j, \mathbf{x}_i)$. Equation (2.24) can also be written in matrix form as $\boldsymbol{\Sigma} = E[(\mathbf{X} - E[\mathbf{X}])(\mathbf{X} - E[\mathbf{X}])^T]$, recalling that the mean of a variable $\mathbf{X}$ is $E(\mathbf{X})$.

We will see in Chapter 6 that the covariance matrix has other uses, but for now we will think about what it tells us about a dataset. In essence, it says how the data varies along each data dimension. This is useful if we want to think about distances again. Suppose I gave you the two datasets shown in Figure 2.13 and the test point (labelled by the large 'X' in the figures) and asked you if the 'X' was part of the data. For the figure on the left you would probably say yes, while for the figure on the right you would say no, even though the two points are the same distance from the centre of the data. The reason for this is that as well as looking at the mean, you've also looked at where the test point lies in relation to the spread of the actual datapoints. If the data is tightly controlled then the test point has to be close to the mean, while if the data is very spread out, then the distance of the test point from the mean does not matter as much. We can use this to construct a distance measure that takes this into account. It is called the Mahalanobis distance after the person who described it in 1936, and is written as:

$$D_M(\mathbf{x}) = \sqrt{(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})}, \tag{2.25}$$

where $\mathbf{x}$ is the data arranged as a column vector, $\boldsymbol{\mu}$ is column vector representing the mean, and $\boldsymbol{\Sigma}^{-1}$ is the inverse of the covariance matrix. If we set the covariance matrix to the identity matrix, then the Mahalanobis distance reduces to the Euclidean distance.

Computing the Mahalanobis distance requires some fairly heavy computational machinery in computing the covariance matrix and then its inverse. Fortunately these are very easy to do in NumPy. There is a function that estimates the covariance matrix of a dataset (`np.cov(x)` for data matrix `x`) and the inverse is called `np.linalg.inv(x)`. The inverse does not have to exist in all cases, of course.

We are now going to consider a probability distribution, which describes the probabilities of something occurring over the range of possible feature values. There are lots of probability distributions that are common enough to have names, but there is one that is much better known than any other, because it occurs so often; therefore, that is the only one we will worry about here.

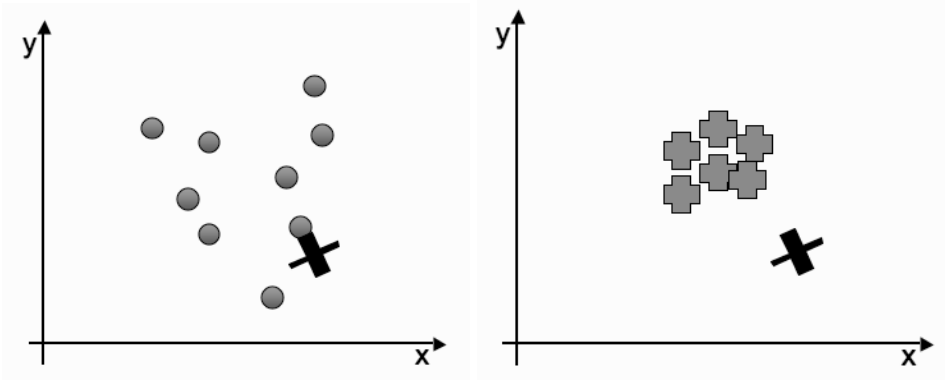


FIGURE 2.13 Two different datasets and a test point.

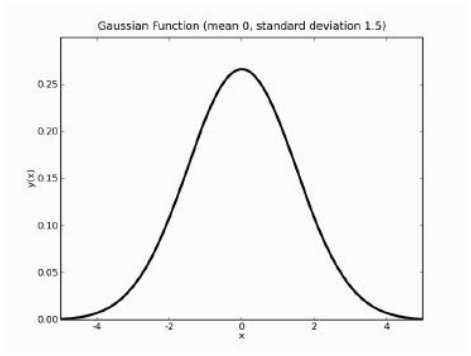


FIGURE 2.14 Plot of the one-dimensional Gaussian curve.

### 2.4.3 The Gaussian

The probability distribution that is most well known (indeed, the only one that many people know, or even need to know) is the Gaussian or normal distribution. In one dimension it has the familiar 'bell-shaped' curve shown in Figure 2.14, and its equation in one dimension is:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right), \tag{2.26}$$

where $\mu$ is the mean and $\sigma$ the standard deviation. The Gaussian distribution turns up in many problems because of the Central Limit Theorem, which says that lots of small random numbers will add up to something Gaussian. In higher dimensions it looks like:

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right), \tag{2.27}$$

where $\boldsymbol{\Sigma}$ is the $n \times n$ covariance matrix (with $|\boldsymbol{\Sigma}|$ being its determinant and $\boldsymbol{\Sigma}^{-1}$ being its inverse). Figure 2.15 shows the appearance in two dimensions of three different cases: when the covariance matrix is the identity; when there are only numbers on the leading diagonal of the matrix; and the general case. The first case is known as a spherical covariance matrix, and has only 1 parameter. The second and third cases define ellipses in two dimensions, either aligned with the axes (with $n$ parameters) or more generally, with $n^2$ parameters.

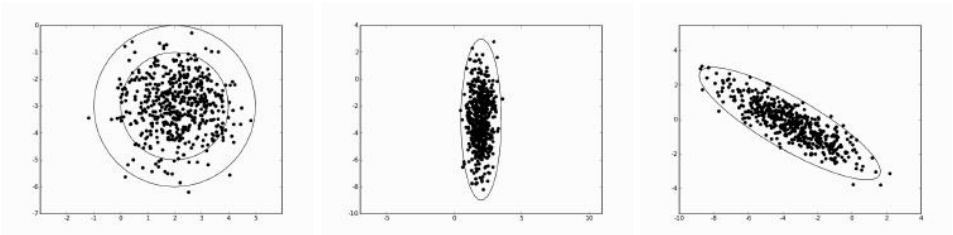

FIGURE 2.15 The two-dimensional Gaussian when (*left*) the covariance matrix is the identity, (*centre*) the covariance matrix has elements on the leading diagonal only, and (*right*) the general case.

## 2.5 THE BIAS-VARIANCE TRADEOFF

To round off this chapter, we use the statistical ideas of the previous section to look again at the idea of how to evaluate the amount of learning that can be performed, from a theoretical perspective.

Whenever we train any type of machine learning algorithm we are making some choices about a model to use, and fitting the parameters of that model. The more degrees of freedom the algorithm has, the more complicated the model that can be fitted. We have already seen that more complicated models have inherent dangers such as overfitting, and requiring more training data, and we have seen the need for validation data to ensure that the model does not overfit. There is another way to understand this idea that more complex models do not necessarily result in better results. Some people call it the bias-variance dilemma rather than a tradeoff, but this seems to be over-dramatising things a little.

In fact, it is a very simple idea. A model can be bad for two different reasons. Either it is not accurate and doesn't match the data well, or it is not very precise and there is a lot of variation in the results. The first of these is known as the bias, while the second is the statistical variance. More complex classifiers will tend to improve the bias, but the cost of this is higher variance, while making the model more specific by reducing the variance will increase the bias. Just like the Heisenberg Uncertainty Principle in quantum physics, there is a fundamental law at work behind the scenes that says that we can't have everything at once. As an example, consider the difference between a straight line fit to some data and a high degree polynomial, which can go precisely though the datapoints. The straight line has no variance at all, but high bias since it is a bad fit to the data in general. The spline can fit the training data to arbitrary accuracy, but the variance will increase. Note that the variance probably increases by rather less than the bias decreases, since we expect that the spline will give a better fit. Some models are definitely better than others, but choosing the complexity of the model is important for getting good results.

The most common way to compute the error between the targets and the predicted outputs is to sum up the squares of the difference between the two (the reason for squaring them is that if we don't, and just add up the differences, and if we had one example where the target was bigger than the prediction, and one where it was smaller by the same amount, then they would sum to zero). When looking at this sum-of-squares error function we can split it up into separate pieces that represent the bias and the variance. Suppose that the function that we are trying to approximate is $y = f(\mathbf{x}) + \epsilon$, where $\epsilon$ is the noise, which is assumed to be Gaussian with 0 mean and variance $\sigma^2$. We use our machine learning algorithm to fit

a hypothesis $h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$ (where $\mathbf{w}$ is the weight vector from Section 2.1) to the data in order to minimise the sum-of-squares error $\sum_i (y_i - h(\mathbf{x}_i))^2$.

In order to decide whether or not our method is successful we need to consider it on independent data, so we consider a new input $\mathbf{x}^*$ and compute the expected value of the sum-of squares error, which we will assume is a random variable. Remember that $E[x] = \bar{x}$, the mean value. We are now going to do some algebraic manipulation, mostly based on the fact that (where Z is just some random variable):

$$\begin{aligned} E[(Z - \bar{Z})^2] &= E[Z^2 - 2Z\bar{Z} + \bar{Z}^2] \\ &= E[Z^2] - 2E[Z]\bar{Z} + \bar{Z}^2 \\ &= E[Z^2] - 2\bar{Z}\bar{Z} + \bar{Z}^2 \\ &= E[Z^2] - \bar{Z}^2. \end{aligned} \tag{2.28}$$

Using this, we can compute the expectation of the sum-of-squares error of a new datapoint:

$$\begin{aligned} E[(y^* - h(\mathbf{x}^*))^2] &= E[y^{*2} - 2y^* h(\mathbf{x}^*) + h(\mathbf{x}^*)^2] \\ &= E[y^{*2}] - 2E[y^* h(\mathbf{x}^*)] + E[h(\mathbf{x}^*)^2] \\ &= E[(y^{*2} - f(\mathbf{x}^*))^2] + f(\mathbf{x}^*)^2 + E[(h(\mathbf{x}^* - \bar{h}(\mathbf{x}^*))^2] \\ &+ \bar{h}(\mathbf{x}^*)^2 - 2f(\mathbf{x}^*)\bar{h}(\mathbf{x}^*) \\ &= E[(y^{*2} - f(\mathbf{x}^*))^2] + E[(h(\mathbf{x}^*) - \bar{h}(\mathbf{x}^*))^2] + (f(\mathbf{x}^*) + \bar{h}(\mathbf{x}^*))^2 \\ &= \text{noise}^2 + \text{variance} + \text{bias}^2. \end{aligned} \tag{2.29}$$

The first of the three terms on the right of the equation is beyond our control. It is the irreducible error and is the variance of the test data. The second term is variance, and the third is the square of the bias. The variance tells us how much $\mathbf{x}^*$ changes depending on the particular training set that was used, while the bias tells us about the average error of $h(\mathbf{x}^*)$. It is possible to exchange bias and variance, so that you can have a model with low bias (meaning that on average the outputs are current), but high variance (meaning that the answers wibble around all over the place) or vice versa, but you can't make them both zero – for each model there is a tradeoff between them. However, for any particular model and dataset there is some reasonable set of parameters that will give the best results for the bias and variance together, and part of the challenge of model fitting is to find this point.

This tradeoff is a useful way to see what machine learning is doing in general, but it is time now to go and see what we can actually do with some real machine learning algorithms, starting with neural networks.

## FURTHER READING

Any standard statistics textbook gives more detail about the basic probability and statistics introduced here, but for an alternative take from the point of view of machine learning, see:

- Sections 1.2 and 1.4 of C.M. Bishop. *Pattern Recognition and Machine Learning.* Springer, Berlin, Germany, 2006.

For more on the bias-variance tradeoff, see:

- Sections 7.2 and 7.3 of T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition, Springer, Berlin, Germany, 2008.

There are two books on semi-supervised learning that can be used to get an overview of the area:

- O. Chapelle, B. Schölkopf, and A. Zien. *Semi-supervised learning*. MIT Press, Cambridge, MA, USA, 2006.
- X. Zhu and A.B. Goldberg. *Introduction to Semi-Supervised Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning, 2009.

## PRACTICE QUESTIONS

**Problem 2.1** Use Bayes' rule to solve the following problem: At a party you meet a person who claims to have been to the same school as you. You vaguely recognise them, but can't remember properly, so decide to work out how likely it is, given that:

- 1 in 2 of the people you vaguely recognise went to school with you
- 1 in 10 of the people at the party went to school with you
- 1 in 5 people at the party you vaguely recognise

**Problem 2.2** Consider how using the risk calculation in Section 2.3.1 would change the naïve Bayes classifier.