# Machine Learning Basics 3

Machine learning is a subfield of artificial intelligence and cognitive science. In artificial intelligence, it is divided into three main branches: *supervised learning*, *unsupervised learning* and *reinforcement learning*. Deep learning is a special approach in machine learning which covers all three branches and seeks also to extend them to address other problems in artificial intelligence which are not usually included in machine learning such as knowledge representation, reasoning, planning, etc. In this book, we will cover supervised and unsupervised learning.

In this chapter, we will be providing the general machine learning basics. These are not part of deep learning, but prerequisites that have been carefully chosen to enable a quick and easy grasp of the elementary concepts needed for deep learning. This is far from a complete treatment, and for a more comprehensive treatment we refer the reader to [1] or any other classical machine learning textbook. The reader interested in the GOFAI approach to knowledge representation and reasoning should consult [2]. The first part of this chapter is devoted to supervised learning and its terminology, while the last part is about unsupervised learning. We will not be covering reinforcement learning and we refer the reader to [3] for a comprehensive treatment.

## 3.1 Elementary Classification Problem

Supervised learning is just classification. The trick is that a vast amount of problems can be seen as classification problems, for example, the problem of recognizing a vehicle in an image can be seen as classifying the image in one of the two classes: ‘has vehicle’ or ‘does not have vehicle’. Same goes for predictions: if we need to

S. Skansi, *Introduction to Deep Learning*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-73004-2_3

make a portfolio of penny stocks, we can reformulate it to be a classification problem of the form: 'winner! will rise 400% or more' or 'nay, pass'.

Of course the trick is to make a classifier that is good enough. We have two options, either selecting by hand with some property or combination of properties (e.g. is the stock bottoming and making an RSI divergence and trading on a low high for the past two days) or we can remain agnostic about the properties we need and simply say 'look, I have 5000 examples of good ones and 5000 examples of bad ones, feed it to an algorithm and let it decide whether the 10001st is more similar to the good ones or the bad ones in terms of the properties it has'. The latter is the quintessential machine learning approach. The former is known as *knowledge engineering* or *expert system engineering* or (historical term) *hacking*. We will focus on the machine learning approach here.

Let us see what 'classification' means. Imagine that we have two classes of animals, say 'dogs' and 'non-dogs'. In Fig. 3.1, each dog is marked with an $X$ and all 'non-dogs' (you can think of them as 'cats') are marked with an $O$. We have two properties for them, their length and their weight. Each particular animal has the two properties associated with it and together they form a *datapoint* (a point in space where the axes are the properties). In machine learning, properties are called *features*. The animal can have a *label* or *target* which says what it is: the label might be 'dog'/'non-dog' or simply '1'/'0'. Notice that if we have the problem of multiclass classification (e.g. 'dog', 'cat' and 'ocelot'), we can first perform a 'dog'/'non-dog' classification and then on the 'non-dog' datapoints perform a 'cat'/'non-cat' classification. But this is rather cumbersome and we will develop techniques for multiclass classification which can do it right away without the need to transform it in $n - 1$ binary classifications.

Returning to our Fig. 3.1, imagine that we have three properties, the third being height. Then, we would need a 3D coordinate system or space. In general, if we have $n$ properties, we would need an $n$-dimensional system. This might seem hard to imagine, but notice what is happening in the 2D versus 3D case and then generalize it: look at the two animals which have the 2D coordinates (38, 7) (it is the overlapping

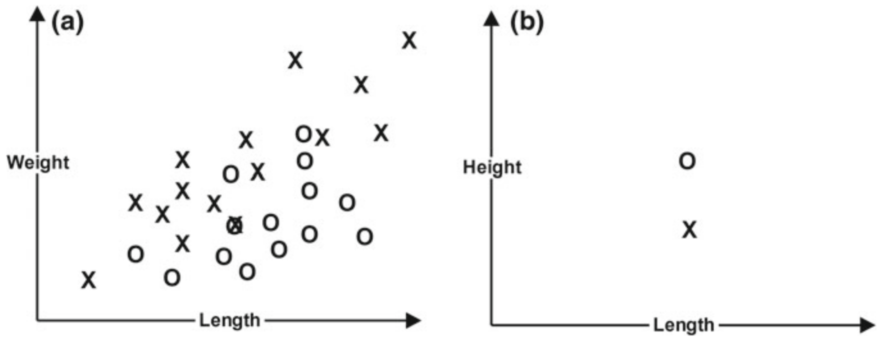

**Fig. 3.1** Adding a new dimension

$X$ and $O$ in Fig. 3.1a). We will never be able to distinguish them, and if a new animal were to have this length and weight we would not be able to conclude what it is.

But take a look at the 'top view' in Fig. 3.1b where we have added an axis $z$: if we were to know that its height (coordinate $z$) is 20 for one and 30 for the another, we could now easily separate them in this 3D space, but we would need a plane instead of a line if we wanted to draw a boundary between them (and this boundary drawing is actually the essence of classification). The point is that adding a new feature and expanding our graph to a new dimension offers us new ways to separate what was very hard or even impossible in a lower number of dimensions. This is a good intuition to keep while imagining 37-dimensional space: it is the expansion of 36-dimensional space with one extra property that will enable us (hopefully) to better distinguish what we could not distinguish in 36-dimensional space. In a 4D space or higher, this plane is which divides cats and dogs the so-called a *hyperplane* which is one of the most important concepts in machine learning. Once we have the hyperplane which separates the two classes in an $n$-dimensional space, we know for a new unlabelled datapoint what (probably) is just by looking whether it falls in the 'dog' side or the 'non-dog' side.

Now, the hard part is to draw a good hyperplane. Let us return to the 2D world where we have just a line (but we will keep calling it 'hyperplane' to inculcate the terminology) and look at some examples. $X$s and $O$s represent dogs and cats (labelled datapoints) and little squares represent new unlabelled datapoints. Notice that we have all the properties for these new datapoints, we are just missing a label and we have to find it. We even know how to find it: see on which side of the hyperplane the datapoint is and then add the label which is the label of that side of the hyperplane.[1] Now, we only need to find out how to define the hyperplane. We have one fundamental choice: should we ignore the labelled datapoints and draw the hyperplane by some other method, or should we try to draw the hyperplane so that it *fits* the existing labelled datapoints nicely? The former approach seems to be the epitome of irrationality, while the latter is the machine learning approach.

Let us comment on the different hyperplanes drawn in Fig. 3.2. Hyperplane A is more or less useless. It has a certain appeal since it does separate the datapoints in a manner that on the 'dog' side there are more dogs than non-dogs and on the 'non-dog' side there are more non-dogs. But it seems that we could have done this with no data at all. Hyperplane B is similar, but it has an interesting feature, namely that on the 'non-dog' side all datapoints are non-dogs. If a new datapoint falls here, we would be very confident that it is a cat. On the other side, things are not good. But if we recast this problem in a marketing setting where $O$s represent people who will most probably buy a product, then a hyperplane like B would provide a very

[1] You may wonder how a side gets a label, and this procedure is different for the various machine learning algorithms and has a number o peculiarities, but for now you may just think that the side will get the label which the majority of datapoints on that side have. This will usually be true, but is not an elegant definition. One case where this is not true is the case where you have only one dog and two cats overlapping (in 2D space) it and four other cats. Most classifiers will place the dog and the two cats in the category 'dog'. Cases like this are rare, but they may be quite meaningful.

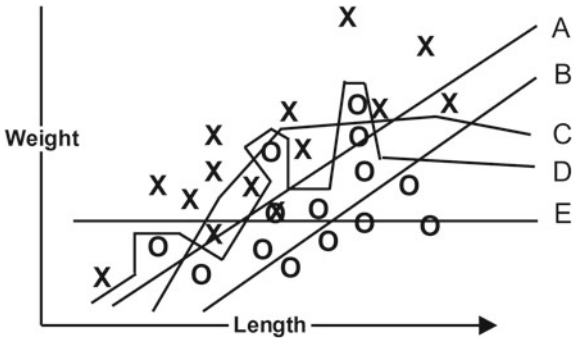

**Fig. 3.2** Different hyperplanes

useful separation. Hyperplane E is even worse than hyperplane A, but to define it we just need a threshold on the weight like $weight > 5$. Here, we could quite easily combine it with other parameters and find a better separation by purely logical ways (no arithmetical operations, just relations $<$, $>$ and $=$ and logical connectives $\wedge$, $\vee$, $\neg$). This could offer us the insight on *what the hyperplane means*, since we would know exactly how it behaves and manually tweak it. If we use machine learning for delicate matters (e.g. predicting failures for nuclear reactors), we want to be able to understand the *why*. This is the basis of *decision tree learning* [4], which is a very useful first model when tackling an unknown dataset.[2]

Hyperplane D seems great—it catches all $X$s on one side and all $O$s on the other. Why not use that? Notice how it went out of its way to catch the middle $O$. We might worry about a hyperplane that provides a perfect fit to the existing data, since there is always some noise[3] in the data, and a new datapoint that falls here might happen to be an $X$. Think of it this way. If there was no $O$ here, would you still justify the same loop? Probably no. If 25% of the overall $O$s were here, would that justify a loop like this? Probably yes. So, there seems to be a fuzzy limit of the number of $O$s we want to see to make such a loop justified. The point is that we want the classifier to be good for new instances, and a classifier that works in 100% of the old cases is probably learning noise along with the important and necessary information from the datapoints. Hyperplane C is a reasonable separation which is quite good and seems to be less concerned with precision than hyperplane C. It is not perfect, but it seems to be capturing a rather general trend in the data.

There is, however, a dose of simplicity in hyperplanes A, B and particularly E we would love to have. Let us see if we can make it happen. What if we use the features we have to create a new one? We have seen we could add a new one like height, but could we just try to build something with what we have? Let us try to plot on the axis $z$ a new feature $\frac{length}{weight}$ (Fig. 3.3, top view). Now, we see that we can

[2]A *dataset* is simply a set of datapoints, some labelled some unlabelled.

[3]Noise is just a name for the random oscillations that are present in the data. They are imperfections that happen and we do not want to learn to predict noise but the elements that are actually relevant to what we want.

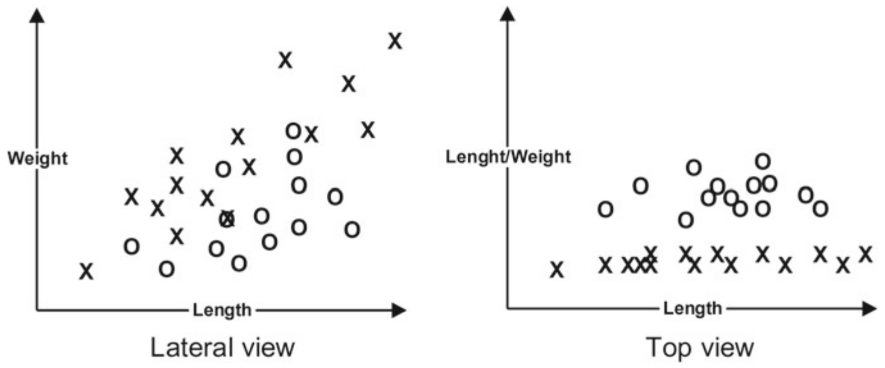

**Fig. 3.3** Feature engineering

actually separate the two classes by a simple straight plane in 3D. When it is possible to separate[4] two classes in an $n$-dimensional space with a 'straight' hyperplane, we say that the classes are *linearly separable*. Usually, one can find a feature which is then added as a new dimension which makes two classes (almost) linearly separable. We can manually add features in which case it is called *feature engineering*, but we would like our algorithms to do it automatically. Machine learning algorithms work by exploiting this idea and they automate the process: they have a linear separator and then they try to find features such that when they are added the classes become linearly separable. Deep learning is no exception, and it is one of most powerful ways to find features automatically. Even though later deep learning will do this for us, to understand deep learning it is important to understand the manual process.

So far we have explored features that are numerical, like height, weight and length. They are specific in two ways. First, order matters: 1 is before 3, 3 is before 14 and we can derive that 1 is before 14. The use of 'before' instead of 'less than' is deliberate. The second thing is that we can add and multiply them. A different kind of feature is an *ordinal feature*. Here, we have the first property of the numerical features 'before' but not the second. Think of the ending positions in a race: the fact that someone is second, someone is third and someone is fourth does not mean that the distance between the second and third is the same as between third and fourth, but the order still holds (second comes before third, and third comes before fourth). If we do not have that either, we are using *categorical features*. Here, we have just the names of the categories and nothing can be inferred from them. An example would be the dog's colour. There are no 'middles' or orders in them, just categories.

Categorical features are very common. Machine learning algorithms cannot accept categorical features as they are and they must be converted. We take the initial table with the categorical feature 'Colour':

[4]It does not have to a perfect separation, a good separation will do.

| Length | Weight | Colour | Label |
|---|---|---|---|
| 34 | 7 | Black | Dog |
| 59 | 15 | White | Dog |
| 54 | 17 | Brown | Dog |
| 78 | 28 | White | Dog |
| … | … | … | … |

And convert it so that we expand the columns with the initial category names and allow only binary values in those columns which indicate which one of the colours the given dog has. This is called *one-hot encoding*, and it increases the dimensionality[5] of the data but now a machine learning algorithm[6] can process the categorical data. The modified table[7] is

| Length | Weight | Brown | Black | White | Label |
|---|---|---|---|---|---|
| 34 | 7 | 0 | 1 | 0 | Dog |
| 59 | 15 | 0 | 0 | 1 | Dog |
| 54 | 17 | 1 | 0 | 0 | Dog |
| 78 | 28 | 0 | 0 | 1 | Dog |
| … | … | … | … | … | … |

We conclude this section by giving a brief description of all supervised machine learning algorithms in terms of input and output. Every *supervised* machine learning algorithm receives a set of training datapoints and labels (they are row vectors). In this phase, the algorithm creates a hyperplane by adjusting its internal parameters. This phase is called the training phase: it receives as inputs row vectors with corresponding labels (called *training samples*) and does not give any output. Instead, in the training phase, the algorithm simply adjusts its internal parameters (and by doing so creates the hyperplane). The next phase is called the predicting phase. In this phase, the trained algorithm takes in a number of row vectors but this time without labels and creates the labels with the hyperplane (depending on which side of the hyperplane the row vectors end up). The row vectors themselves are simply rows from a table like the one above, so the row vector which corresponds to the training sample in the third line is simply (54, 17, 1, 0, 0, *Dog*). If it were a row vector for which we need to predict a label, it would look the same except it would not have the 'Dog' tag in the end.[8]

[5]Think about how one-hot encoding can boost the understanding of $n$-dimensional space.

[6]Deep learning is no exception.

[7]Notice that to do one-hot encoding, it needs to make two passes over the data: the first collects the names of the new columns, then we create the columns, and then we make another pass over the data to fill them.

[8]Strictly speaking, these vectors would not look exactly the same: the training sample would be (54,17,1,0,0, Dog), which is a row vector of length 6, and the row vector for which we want to

## 3.2 Evaluating Classification Results

In the previous section, we have explored the basics of classification and we left the hard part (producing the hyperplane) largely untouched. We will address this in the next section. In this section, we will assume we have a working classifier and we want to see how well it behaves. Take a look at Fig. 3.4.

This image illustrates a classifier named $C$ for classifying $X$s. This is the task for this classifier and it is important to keep this in mind at all times. The black line is the hyperplane, and the grey region is what $C$ considers to be the region of $X$. From the perspective of $C$, everything inside the grey region *is* $X$, while everything outside is not an $X$. We have marked the individual datapoints with $X$ or $O$ depending whether they are in reality an $X$ or $O$. We can see right away that the reality differs from what $C$ thinks and this is the usual scenario when we have and empirical classification task. Intuitively, we see that the hyperplane makes sense, but we want to define objective classification metrics which can tell us how good a classifier is and, if we have two or more, which classifier is the best.

We can now define the concepts of *true positive*, *false positive*, *true negative* and *false negative*. A true positive is a datapoint for which the classifier says it is an $X$ and it truly is an $X$. A false positive is a datapoint for which the classifier thinks it is an $X$ but it is an $O$. A true negative is a datapoint for which the classifier thinks it is *not* and $X$ and in fact it is not, and a false negative is a datapoint for which the classifier thinks it is not an $X$ but in fact it is. In Fig. 3.4, there are five true positives ($X$s in the grey), one false positive (the $O$ in the grey), six true negatives (the $O$s in the white) and two false negatives (the $X$s in the white). Remember, the grey area is the area where the classifier $C$ thinks all are $X$s and the white area is what the classifier thinks all are $O$s.

The first and most fundamental classification metric is *accuracy*. Accuracy simply tells us how good is the classifier at sorting $X$s and $O$s. In other words, it is the

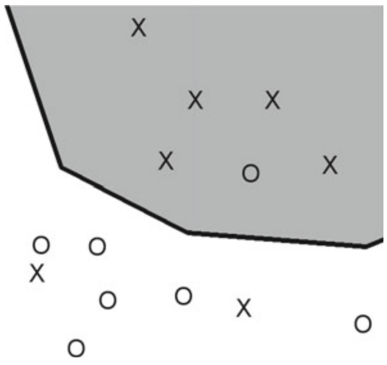
**Fig. 3.4** A classifier $C$ for classifying $X$s

predict the label would have to be of length 5 (without the last component which is the label), e.g. (47,15,0,0,1).

number of true positives, added to the number of true negatives and divided by the total number of datapoints. In our case, this would be $\frac{5+6}{14} = 0.785714\ldots$ but we will be rounding off to four decimal points.[9]

We might be interested in how good is a classifier at avoiding false alarms. The metric used to calculate this is called *precision*. The precision of a classifier on a dataset is calculated by $\frac{truePositives}{truePositives+falsePositives} = \frac{5}{5+1} = 0.8333$. If we are concerned about missing out and we want to catch as many true $X$s we can, we need a different metric called *recall* to measure our success. The recall is calculated by taking $\frac{truePositives}{truePositives+falseNegatives} = \frac{5}{5+2} = 0.7142$.

There is a standard way to display the number of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) in a more visual way and this method is called a *confusion matrix*. For a two-class classification (also known as *binary classification*), the confusion matrix is a $2 \times 2$ table of the form:

| | Classifier says YES | Classifier says NO |
|---|---|---|
| In reality YES | Number of true positives | Number of false negatives |
| In reality NO | Number of false negatives | Number of true negatives |

Once we have a confusion matrix, precision, recall, accuracy and any other evaluation, metric can be calculated directly from it.

The values for all classifier evaluation metrics range from 0 to 1 and can be interpreted as probabilities. Note that there are trivial modifications that can make either the precision or recall reach 100% (but not both at the same time). If we want the precision to be 1, we can simply make a classifier that selects no datapoint, i.e. for each datapoint it should say 'O'. The opposite works for recall: just select all datapoints as $X$s, and recall will be 1. This is why we need all three metrics to get a meaningful insight on how good a classifier is and how to compare two classifiers.

Now that we know about evaluation metrics, let us turn to the question of evaluating the classifier performance from a procedural point of view. When faced with a classification task, as noted earlier we have a classification algorithm and a training set. We train the algorithm on the training set and now we are ready to use it for prediction. But where is the evaluation part? The usual strategy is not to use the whole training set for training, but keep a part of it for testing. This is usually 10%, but it can be more or less than that.[10] The 10% we held out and did not train on it is called the *test set*. In the test set, we separate the labels from the other features, so that we have row vectors of the same form we would be getting when predicting. When we have a trained model on the 90% (the training set), we use it to classify the test set, and we compare the classification results with the labels. In this way, we get the necessary information for calculating the precision, recall, and accuracy. This is

[9]If we will be needing more we will keep more decimals, but in this book we will usually round off to four.

[10]It is mostly a matter of choice, there is no objective way of determining how much to split.

called *splitting the dataset in training and testing sets* or simply the *train–test split*. The test set is designed to be a controlled simulation of how well will the classifier behave. This approach is sometimes called *out-of-sample* validation to distinguish it from *out-of-time* validation where the 10% of the data are not chosen randomly from all datapoints, but a time period spanning around 10% of the datapoints is chosen. Out-of-time validation is generally not recommended since there might be seasonal trends in the data which would seriously cripple the evaluation.

## 3.3 A Simple Classifier: Naive Bayes

In this section, we sketch the simplest classifier we will explore in this book, called the *naive Bayes classifier*. The naive Bayes classifier has been used from at least 1961 [5], but, due to its simplicity, it is hard to pinpoint where research on the applications of Bayes' theorem ends and the research on the naive Bayes classifier begins.

The naive Bayes classifier is based on Bayes' theorem which we saw earlier in Chap. 2 (this accounts for the 'Bayes' in the name), and it makes and additional assumption that all features are conditionally independent from each other (this is why there is 'Naive' in the name). This means that each feature carries 'its own weight' in terms of predictive power: there is no piggy-backing or synergy of features going on. We will rename the variables in the Bayes theorem to give it a more 'machine learning feel':

$$\mathbb{P}(t|f) = \frac{\mathbb{P}(f|t)\mathbb{P}(t)}{\mathbb{P}(f)},$$

where $\mathbb{P}(t)$ is the prior probability[11] of a given target value (i.e. the class label), $\mathbb{P}(f)$ is the prior probability of a feature, $\mathbb{P}(f|t)$ is the probability of the feature $f$ given the target $t$, and, of course, $\mathbb{P}(t|f)$ is the probability of the target $t$ given only the feature $f$ which is what we want to find.

Recall from Chap. 2 that we can convert Bayes' theorem to accommodate for a ($n$-dimensional) vector of features, and in that case we have the following formula:

$$\mathbb{P}(t|f_{all}) = \frac{\mathbb{P}(f_1|t) \cdot \mathbb{P}(f_2|t) \cdot \ldots \cdot \mathbb{P}(f_n|t) \cdot \mathbb{P}(t)}{\mathbb{P}(f_{all})}$$

Let us see a very simple example to demonstrate how the naive Bayes classifier works and how it draws its hyperplane. Imagine that we have the following table detailing visits to a webpage:

We first need to convert this into a table with counts (called a *frequency table*, similar to one-hot, but not exactly the same):

Now, we can calculate some basic prior probabilities. The probability of 'yes' is $\frac{9}{13} = 0.6923$. The probability of 'no' is $\frac{4}{13} = 0.3076$. The probability of 'morning'

[11]The prior probability is just a matter of counting. If you have a dataset with 20 datapoints and in some feature there are five values of 'New Vegas' while the others (15 of them) are 'Core region', the prior probability $\mathbb{P}(New\ Vegas) = 0.25$.

| Time | Buy |
|---|---|
| morning | no |
| afternoon | yes |
| evening | yes |
| morning | yes |
| morning | yes |
| afternoon | yes |
| evening | no |
| evening | yes |
| morning | no |
| afternoon | no |
| afternoon | yes |
| afternoon | yes |
| morning | yes |

| Time | yes | no | TOTAL |
|---|---|---|---|
| morning | 3 | 2 | 5 |
| afternoon | 4 | 1 | 5 |
| evening | 2 | 1 | 3 |
| TOTAL | 9 | 4 | 13 |

is $\frac{5}{13} = 0.3846$. The probability of 'afternoon' is $\frac{5}{13} = 0.3846$. The probability of 'evening' is $\frac{3}{13} = 0.2307$. Ok, that takes care of all the probabilities which we can calculate just by counting from the dataset (the so-called 'priors' we addressed in Sect. 2.3 of Chap. 2). We will be needing one more thing but we will get to it.

Imagine now we are given a new case for which we do not know the target label and we must predict it. This new case is the row vector $(morning)$[12] and we want to know whether it is a 'yes' or a 'no', so we need to calculate

$$\mathbb{P}(yes|morning) = \frac{\mathbb{P}(morning|yes)\mathbb{P}(yes)}{\mathbb{P}(morning)}$$

We can plug in the priors $\mathbb{P}(yes) = 0.6923$ and $\mathbb{P}(morning) = 0.3846$ we calculated above. Now, we only need to calculate $\mathbb{P}(morning|yes)$, which is the percentage of times the 'morning' occurs if we restrict ourselves to the rows which have 'yes', which is present 9 times, and out of these, three have also a 'yes', so we have $\mathbb{P}(morning|yes) = \frac{3}{9} = 0.3333$. Taking it all to Bayes' theorem, we have

$$\mathbb{P}(yes|morning) = \frac{\mathbb{P}(morning|yes) \cdot \mathbb{P}(yes)}{\mathbb{P}(morning)} = \frac{0.3333 \cdot 0.6923}{0.3846} = 0.5999$$

[12] If we were to have $n$ features, this would be an $n$-dimensional row vector such as $(x_1, x_2, \ldots, x_n)$, but now we have only one feature so we have a 1D row vector of the form $(x_1)$. A 1D vector is *exactly the same* as the scalar $x_1$ but we keep referring to it as a vector to delineate that in the general case it would be an $n$-dimensional *vector*.

We also know that $\mathbb{P}(no|morning) = 1 - \mathbb{P}(yes|morning) = 0.4$. This means that the datapoint gets the label 'yes', since the value is over 0.5 (we have two classes). In general, if we were to have $n$ classes, $\frac{1}{n}$ is the value over which the probability would have to be.

The diligent reader could say that we could have calculated $\mathbb{P}(yes|morning)$ directly from the table as we did with $\mathbb{P}(morning|yes)$, and this is true. The problem is that we can do it by counting from the table only if there is a single feature, so for the case of multiple features we would have to use calculation we actually used (with the expanded formula for multiple features).

Naive Bayes is a simple algorithm, but it is still very useful for large datasets. In fact, if we adopt a probabilistic view of machine learning and claim that all machine learning algorithms actually learn only $\mathbb{P}(y|\mathbf{x})$, we could say that naive Bayes is *the* simplest machine learning algorithm, since it has only the bare necessities to make the 'flip' from $\mathbb{P}(f|t)$ to $\mathbb{P}(t|f)$ work (from counting to predicting). This is a specific (probabilistic) view of machine learning, but it is compatible with the deep learning mindset, so feel free to adopt it as a pet.

One important thing to remember is that naive Bayes makes the conditional independence assumption.[13] So it cannot handle any dependencies in the features. Sometimes, we might want to be able to model sequences like this, e.g. when the order of the feature matters (we will see this come into play for language modelling or for sequences of events in time), and naive Bayes is unable to do this. Later in the book, we will present deep learning models fully capable of handling this. Before continuing on, notice that the naive Bayes classifier had to draw a hyperplane to be able to classify the new datapoints. Suppose we had a binary classification at hand. Then, naive Bayes expanded the space by one dimension (so the row vectors are augmented to include this value), and that dimension accepts values between 0 and 1. In this dimension, the hyperplane is visible and it passes through the value 0.5.

## 3.4 A Simple Neural Network: Logistic Regression

Supervised learning is usually divided into two types of learning. The first one is classification, where we have to predict the class. We have seen that already with naive Bayes, we will see it again countless times in this book. The second one is regression where we predict a value, and we will not be exploring regression in this book.[14] In this section, we explore *logistic regression* which is *not* a regression algorithm but a classification algorithm. The reason behind this is that it is considered a regression model in statistics and the machine learning community just adopted it and began using it as a classifier.

[13]That is, the assumption that features are conditionally independent given the target.

[14]Regression problems can be simulated with classification. An example would be if we had to find the proper value between 0 and 1, and we had to round it in two decimals, then we could treat it as a 100-class classification problem. The opposite also holds, and we have actually seen this in the naive Bayes section, where we had to pick a threshold over which we would consider it a 1 and below which it would be a 0.

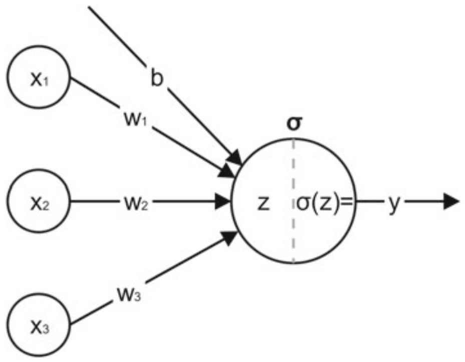

**Fig. 3.5** Schematic view of logistic regression

Logistic regression was first introduced in 1958 by D. R. Cox [6], and a considerable amount of research was done both on logistic regression and using logistic regression. Logistic regression is mainly used today for two reasons. First, it gives an interpretation of the relative importance of features, which is nice to have if we wish to build an intuition on a given dataset.[15] The second reason, which is much more important to us, is that the logistic regression is actually a one-neuron neural network.[16]

By understanding logistic regression, we are taking a first and important step towards neural networks and deep learning. Since logistic regression is a supervised learning algorithm, we will have to have the target values for training included in the row vectors for the training set. Imagine that we have three training cases, $\mathbf{x}_A = (0.2, 0.5, 1, 1)$ , $\mathbf{x}_B = (0.4, 0.01, 0.5, 0)$ and $\mathbf{x}_C = (0.3, 1.1, 0.8, 0)$. Logistic regression has a much input neurons as it has features in the row vectors,[17] which is in our case 3.

You can see a schematic representation of logistic regression in Fig. 3.5. As for the calculation part, the logistic regression can be divided into two equations:

$$z = b + w_1x_1 + w_2x_2 + w_3x_3,$$

which calculates the *logit* (also known as the weighted sum) and the *logistic* or *sigmoid* function:

$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$

[15]Afterwards, we may do a bit of feature engineering and use an all-together different model. This is important when we do not have an understanding of the data we use which is often the case in industry.

[16]We will see later that logistic regression has more than one neuron, since each component of the input vector will have to have an input neuron, but it has 'one' neuron in the sense of having a single 'workhorse' neuron.

[17]If the training set consists of $n$-dimensional row vectors, then there are exactly $n - 1$ features—the last one is the target or label.

If we join them and tidy up a bit, we have simply

$$y = \sigma(b + w_1 x_1 + w_2 x_2 + w_3 x_3)$$

Now, let us comment on these equations. The first equation shows how to calculate the logit from the inputs. The inputs in deep learning are always denoted by $x$, the output of the neuron is always denoted by $y$ and the logit is denoted by $z$ or sometimes $a$. The equations above make use of all the notational abuse which is common in the machine learning community, so be sure to understand why the symbols are employed as they are.

To calculate the logit, we need (asides from the inputs) the weights $w$ and the bias $b$. If you look at the equations, you will notice that everything except the bias and weights is either an input or calculated. The elements which are not given as inputs or are constants like $e$ are called *parameters*. For now, the parameters are the weights and biases, and the point of logistic regression is to *learn* a good vector of weights and a good bias to achieve good classification. This is *the* only learning in logistic regression (and deep learning): finding a good set of weights.

But what are the weights and biases? The weights control how much of each feature from the input we should let in. You can think about them as if they represent percentages. They are not limited to the interval between 0 and 1, but this is a good intuition to have. For weights over 1, you could think of them as 'amplifications'. The bias is a bit more tricky. Historically,[18] it has been called threshold and it behaved a bit differently. The idea was that the logit would simply calculate the weighted sum of the inputs, and if it was above the threshold, the neuron would output a 1, otherwise a 0. The 1 and 0 part was replaced by our equation for $\sigma(z)$, which does not output a sharp 0 or 1, but instead it ranges from 0 to 1. You can see the different plots on Fig. 3.6. Later, in Chap. 4, we will see how to incorporate the bias as one of the weights. For now, it is enough to know that the bias can be absorbed as one of

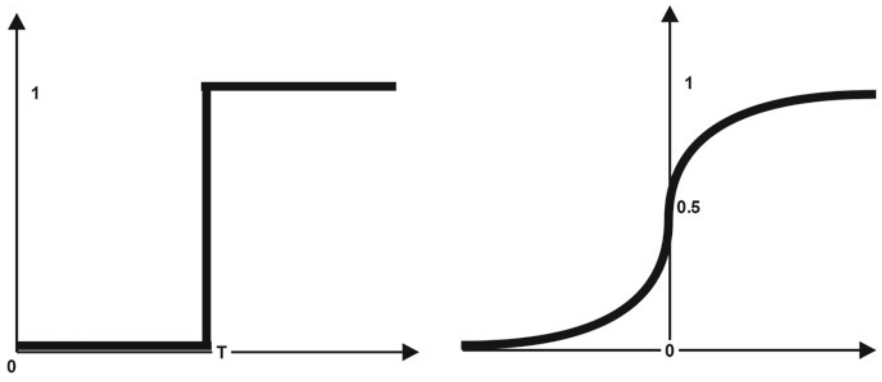

**Fig. 3.6** Historic and actual neuron activation functions

[18]Mathematically, the bias is useful to make an offset called the intercept.

the weights so we can forget about the bias knowing it will be taken care of and it will become one of the weights.

Let us make a calculation based on our inputs which will explain the mechanics of logistic regression. We will need a starting value for the weights and bias, and we usually produce this at random. This is done from a gaussian random variable, but to keep things simple, we will generate a set of weights and bias by taking random values between 0 and 1. Now, we would need to pass the input row vectors through one-hot encoding and normalize them, but suppose they already have been one-hot encoded and normalized. So we have $\mathbf{x}_A = (0.2, 0.5, 0.91, 1)$, $\mathbf{x}_B = (0.4, 0.01, 0.5, 0)$ and $\mathbf{x}_C = (0.3, 1.1, 0.8, 0)$ and assume that the randomly generated weight vector is $\mathbf{w} = (0.1, 0.35, 0.7)$ and the bias is $b = 0.66$. Now we turn to our equations, and put in the first input:

$$y_A = \sigma(0.66 + 0.1 \cdot 0.2 + 0.35 \cdot 0.5 + 0.7 \cdot 0.91) = \sigma(1.492) = \frac{1}{1 + e^{-1.492}} = 0.8163$$

We note the result 0.8163 and the actual label 1. Now we do the same for the second input:

$$y_B = \sigma(0.66 + 0.1 \cdot 0.4 + 0.35 \cdot 0.01 + 0.7 \cdot 0.5) = \sigma(1.0535) = \frac{1}{1 + e^{-1.0535}} = 0.7414$$

Noting again the result 0.7414 and label 0. And now we do it for the last input row vector:

$$y_C = \sigma(0.66 + 0.1 \cdot 0.3 + 0.35 \cdot 1.1 + 0.7 \cdot 0.8) = \sigma(1.635) = \frac{1}{1 + e^{-1.635}} = 0.8368$$

Noting again the result 0.8368 and the label 0. It seems quite clear that we did good on the first, but failed to classify the second and third input correctly. Now, we should update the weights somehow, but to do that we need to calculate how lousy we were at classifying. For measuring this, we will be needing an *error function* and we will be using the *sum of squared error* or SSE[19]:

$$E = \frac{1}{2} \sum_n (t^{(n)} - y^{(n)})^2$$

The $t$s are targets or labels, and the $y$s are the actual outputs of the model. The weird exponents $(t^{(n)})$ are just indices which range across training samples, so $(t^{(k)})$ would be the target for the $k$th training row vector. You will see in a moment why

[19]There are other error functions that can be used, but the SSE is one of the simplest.

do we need such weird notation now and a bit later how to dispense with it. Let us calculate our SSE:

$$E = \frac{1}{2}\sum_n (t^{(n)} - y^{(n)})^2 = \tag{3.1}$$

$$= \frac{1}{2}((1 - 0.8163)^2 + (0 - 0.7414)^2 + (0 - 0.8368)^2) = \tag{3.2}$$

$$= \frac{0.0337 + 0.5496 + 0.7002}{2} = \tag{3.3}$$

$$= 0.64175 \tag{3.4}$$

We now update the **w** and $b$ by using magic, and get $\mathbf{w} = (0.1, 0.36, 0.3)$ and $b = 0.25$. Later (in Chap. 4), we will see it is actually done by something called the *general weight update rule*. This completes one cycle of weight adjustment. This is colloquially called an *epoch*, but we will redefine this term later in Chap. 4 to make it more precise. Let us recalculate the outputs and the new SSE to see whether the new set of weights is better:

$$y_A^{new} = \sigma(0.25 + 0.1 \cdot 0.2 + 0.36 \cdot 0.5 + 0.3 \cdot 0.91) = \sigma(0.723) = \frac{1}{1 + e^{-0.723}} = 0.6732 \tag{3.5}$$

$$y_B^{new} = \sigma(0.25 + 0.1 \cdot 0.4 + 0.36 \cdot 0.01 + 0.3 \cdot 0.5) = \sigma(0.4436) = \frac{1}{1 + e^{-0.4436}} = 0.6091 \tag{3.6}$$

$$y_C^{new} = \sigma(0.25 + 0.1 \cdot 0.3 + 0.36 \cdot 1.1 + 0.3 \cdot 0.8) = \sigma(0.916) = \frac{1}{1 + e^{-1.635}} = 0.7142 \tag{3.7}$$

$$E^{new} = \frac{1}{2}((1 - 0.6732)^2 + (0 - 0.6091)^2 + (0 - 0.7142)^2) = \tag{3.8}$$

$$= \frac{0.1067 + 0.371 + 0.51}{2} = \tag{3.9}$$

$$= 0.4938 \tag{3.10}$$

We can see clearly that the overall error has decreased. We can continue this procedure a number of times, and the error will decrease, until at one point it will stop decreasing and stabilize. On rare occasions, it might even exhibit chaotic behaviour. This is the essence of logistic regression, and the very core of deep learning—everything we do will be an upgrade or modification of this.

Let us turn our attention to data representation. So far we have used an expanded view of the process so that we may see clearly everything, but let us see how we can make the procedure more compact and computationally faster. Notice that even though a dataset is a set (and the order does not matter), it might make a bit of sense to put $\mathbf{x}_A$, $\mathbf{x}_B$ and $\mathbf{x}_C$ in a vector, since we will be using them one by one (the vector would then simulate a queue or stack). But since they also share the same structures (same features in the same place in each row vector), we might opt for a matrix to represent the whole training set. This is important in the computational sense as well since most deep learning libraries have somewhere in the background C, and arrays (the programming equivalent of matrices) are a native data structure in C, and computation on them is incredibly fast.

So what we want to do is first turn the $n$ $d$-dimensional input vectors into and input matrix of the size $n \times d$. In our case, this is a $3 \times 3$ matrix:

$$\mathbf{x} = \begin{bmatrix} 0.2 & 0.5 & 0.91 \\ 0.4 & 0.01 & 0.5 \\ 0.3 & 1.1 & 0.8 \end{bmatrix}$$

We will be keeping the targets (labels) in a separate vector, and we have to be extremely careful not to shuffle neither the target vector nor the dataset matrix from this point onwards, since the order of the matrix rows and vector components is the only thing that can join them again. The target vector in our case is $\mathbf{t} = (1, 0, 0)$.

Let us turn our attention to the weights. The bias is a bit of a bother, so we can turn it in one of the weights. To do this, we have to add a single column of 1's as the first column of the input matrix. Notice that this will not be an approximation, but will capture *exactly* the calculation we need to perform. As for the weights, we will be needing as many weights as there are inputs. Also, if we have more than one workhorse neuron, we would need to have that many times the weights, e.g. if we have 5 inputs (5-dimensional input row vectors) and 3 workhorse neurons, we would need $5 \times 3$ weights. This $5 \times 3$ is deliberate, since we would be using a $5 \times 3$ matrix[20] to store it in, since then we could do all the calculations needed for the logit with a simple matrix multiplication. This illustrates something that could be called 'the general deep learning strategy for fast computation': try to do as much work as you can with matrix (and vector) multiplication and transpositions.

Returning to our example, we have three inputs and we add the column of 1's in front of the inputs to make room for the bias in the weight matrix. The new input matrix is now a $3 \times 4$ matrix:

$$\mathbf{x} = \begin{bmatrix} 1 & 0.2 & 0.5 & 0.91 \\ 1 & 0.4 & 0.01 & 0.5 \\ 1 & 0.3 & 1.1 & 0.8 \end{bmatrix}$$

Now we can define the weight matrix. It is a $4 \times 1$ matrix consisting of the bias followed by weight:

$$\mathbf{w} = \begin{bmatrix} 0.66 \\ 0.1 \\ 0.35 \\ 0.7 \end{bmatrix}$$

[20]Recall that this is not the same as a $3 \times 5$ matrix.

This matrix can be equivalently represented as $(0.66, 0.1, 0.35, 0.7)^\top$, but we will use the matrix form for now. Now, to calculate the logit we do simple matrix multiplication of the two matrices, with which we will get a $3 \times 1$ matrix in which every row (there is a single value in every row) will represent the logit for each training case (compare this with the previous calculation):

$$\mathbf{z} = \mathbf{xw} = \begin{bmatrix} 1 & 0.2 & 0.5 & 0.91 \\ 1 & 0.4 & 0.01 & 0.5 \\ 1 & 0.3 & 1.1 & 0.8 \end{bmatrix} \cdot \begin{bmatrix} 0.66 \\ 0.1 \\ 0.35 \\ 0.7 \end{bmatrix} = \tag{3.11}$$

$$= \begin{bmatrix} 1 \cdot 0.66 + 0.2 \cdot 0.1 + 0.5 \cdot 0.35 + 0.91 \cdot 0.7 \\ 1 \cdot 0.66 + 0.4 \cdot 0.1 + 0.01 \cdot 0.35 + 0.5 \cdot 0.7 \\ 1 \cdot 0.66 + 0.3 \cdot 0.1 + 1.1 \cdot 0.35 + 0.8 \cdot 0.7 \end{bmatrix} = \tag{3.12}$$

$$= \begin{bmatrix} 1.492 \\ 1.0535 \\ 1.635 \end{bmatrix} \tag{3.13}$$

Now we must only apply the logistic function $\sigma$ to $\mathbf{z}$. This is done by simply applying the function to each element of the matrix:

$$\sigma(\mathbf{z}) = \begin{bmatrix} \sigma(1.492) \\ \sigma(1.0535) \\ \sigma(1.635) \end{bmatrix} = \begin{bmatrix} 0.8163 \\ 0.7414 \\ 0.8368 \end{bmatrix}$$

We add a final remark. The logistic function is the main component of the logistic regression. But if we treat the logistic regression as a simple neural network, we are not committed to the logistic function. In this view, the logistic function is a *nonlinearity*,[21] i.e. it is the component which enables complex behaviour (especially when we expand the model beyond a single workhorse neuron of the classic logistic regression). There are many types of nonlinearity, and they all have a slightly different behaviour. The logistic regression ranges between 0 and 1. Another common nonlinearity is the *hyperbolic tangent* or $tanh$, which we will denote by $\tau$ to enforce a bit of notational consistency. The $\tau$ nonlinearity ranges between $-1$ and 1, and has a similar shape like the logistic function. It is calculated by

$$\tau(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{3.14}$$

The choice of which activation function to use in neural networks is a matter of preference, and it is often guided by the results one obtains using them. If, we use the hyperbolic tangent in logistic regression instead of the logistic function, it will still work nicely, but technically that is not logistic regression anymore. Neural networks, on the other hand, are still neural networks regardless of which nonlinearity we use.

[21] In the older literature, this is sometimes called *activation function*.

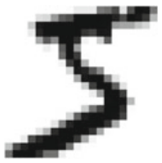
**Fig. 3.7** A single MNIST datapoint

## 3.5 Introducing the MNIST Dataset

The MNIST dataset is a modification of the National Institute of Standards and Technology of the United States dataset consisting of handwritten digits. The original datasets are described in [7] and the MNIST (*modified NIST*) is a modification of the Special Database 1 and Special Database 3 of the original dataset compiled by Yann LeCun, Corinna Cortes and Christopher J. C. Burges. The MNIST dataset was first used in the paper [8]. Geoffrey Hinton called MNIST 'the fruit fly of machine learning'[22] since a lot of research in machine learning was performed on it and it is quite versatile for a number of simple tasks. Today, MNIST is available from a variety of sources, but the 'cleanest' source is probably Kaggle where the data is kept in a simple CSV file,[23] accessible by any software with ease. In Fig. 3.7 (image taken from [9]), we can see an example of a MNIST digit.

MNIST images are 28 by 28 pixels in greyscale, so the value for each pixel ranges between 0 (white) and 255 (black). This is different from the usual greyscale where 0 is black and 255 is white, but the community thought it might make more sense since it can be stored in less space this way, but this is a minor point today for a dataset of the size of MNIST.

There is one issue here, to which we will return at the very end of the book. The problem is that all currently available supervised machine learning algorithms can only process vectors as inputs: no matrices, graphs, trees, etc. This means that whatever we are trying to do, we have to find a way to put in vector form and transform *all* of our inputs in $n$-dimensional vectors. The MNIST dataset consists of 28 by 28 images, so, in essence, the inputs are matrices. Since they are all of the same size, we can transform them in 784-dimensional vectors.[24] We could do this by simply 'reading' them as we would a written page: left to right, after the row of pixels ends, move to the leftmost part of the next line and continue again. By doing this, we have transformed a $28 \times 28$ matrix into a 784-dimensional vector. This is a rather simple transformation (note that it only works if all input samples are of the same size), and if we want to learn graphs and trees, we have to have a vector representation of them. We will return to this as an open problem at the very end of this book.

There is one additional point we want to make here. MNIST consists of greyscale images. What could we do if it was RGB? Recall that an RGB image consists of three

[22]See http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec1.pdf.
[23]Available at https://www.kaggle.com/c/digit-recognizer/data.
[24]The interested reader may look up the details in Chap. 4 of [10].

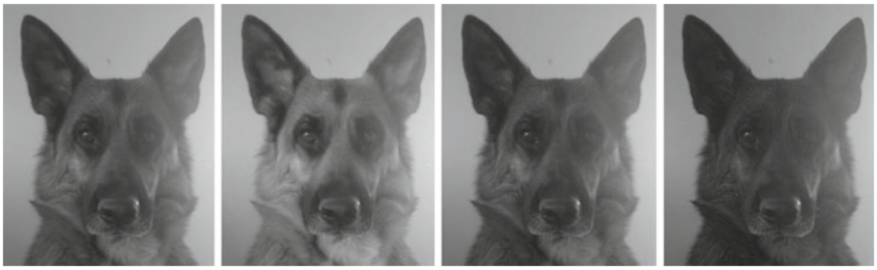

**Fig. 3.8** Greyscale for all colours, red channel, green channel and blue channel

component ‘images’ called *channels*: red, green and blue. They are joined to form the complete (colour) image. We could print these in colour (each pixel of the red channel would have a value from 0 to 255 to denote how much red is in it), but we have actually converted the colours to greyscale without noticing (see Fig. 3.8). It might seem weird to represent the red channel as grey, but that is *exactly* what a computer does. The *name* of the channel image is ‘red’ but the values in pixels are between 0 and 255, which is, *computationally speaking*, grey. This is because an RGB pixel is simply three values from 0 to 255. The first one is *called* ‘red’, but computationally it is red just because it is in the first place. There is no intrinsic ‘redness’ or qualia in it. If we were to display the pixels without providing the other two components, 0 will be interpreted as black and 255 as white, making it a greyscale. In other words, an RGB image would have a pixel with the value (34, 67, 234), but if we separate a channel by taking only the red component 34 we would get a greyscale. To get the ‘redness’ in the display, we must state it as (34, 0, 0) and keep it as an RGB image. And the same goes for green and blue. Returning to our initial question, if we were processing RGB images would have several options:

- Average the components to produce and average greyscale representation (this is the usual way to create greyscale images from RGB).
- Separate the channels and form three different datasets and train three classifiers. When predicting, we take the average of their result as the final result. This is an example of a *committee* of classifiers.
- Separate the channels in distinct images, shuffle them and train a single classifier on all of them. This approach would be essentially *dataset augmentation*.
- Separate the channels in distinct images, train three instances of the same classifier on each (same size and parameters), and then use a fourth classifier to make the final call. This is the approach that leads to *convolutional neural networks* which we will explore in detail in Chap. 6.

Each of these approaches has its merit, and depending on the problem at hand, any of them can be a good choice. You can consider other options, deep learning has an exploratory element to it, and an unorthodox method which contributes to accuracy will be appreciated.

## 3.6 Learning Without Labels: K-Means

We now turn our attention to two algorithms for unsupervised learning, the *K-means* and the *PCA*. We will briefly address PCA in the next section (especially the intuition behind it), but we will be returning to it in Chap. 9 where we will be giving the technical details. PCA represents a branch of unsupervised learning called *distributed representations*, and it is one of the most important topics in deep learning today, and PCA is the simplest algorithm for building distributed representations.[25] Another branch of unsupervised learning which is conceptually simpler is called *clustering*. The goal of clustering is to assign all datapoints to clusters which (hopefully) capture their similarity in $n$-dimensional space. K-means is the simplest clustering algorithm, and we will use it to illustrate how a clustering algorithm works.[26]

But before we proceed to K-means, let us comment briefly what is unsupervised learning. *Unsupervised learning is learning without labels or targets*. Since unsupervised learning is usually the last of the three areas to be defined (supervised and reinforcement being the other two), there is a tendency to put everything which is not supervised or reinforcement learning in unsupervised learning. This is a very broad definition, but it is very interesting, since it begs the cognitive question of how we learn without feedback, and is learning without feedback actually learning or is it a different phenomenon? By exploring unsupervised learning, we are dwelling deep in cognitive modelling and this makes this an exciting and colourful area.

Let us demonstrate how K-means works. K-means is a clustering algorithm, which means it will produce clusters of data. Producing clusters actually means assigning a cluster name to all datapoints so that similar datapoints share the same cluster name. The usual cluster names are ‘1’, ‘2’, ‘3’, etc. Assume we have two features so that we work in 2D space. In unsupervised learning, we do not have a training and testing set, but all datapoints we have are ‘training’ datapoints, and we build the clusters (which will define the hyperplane) from them. The input row vectors do not have a label; they consist only of features.

The K-means algorithm takes as an input the number of centroids to be used. Each centroid will define a cluster. At the very start of the algorithm, the centroids are placed in a random location in the datapoint vector space. K-means has two phases, one called ‘assign’ and the another ‘minimize’ forming a cycle, and it repeats this cycle a number of times.[27] During the assign phase, each datapoint is assigned to the nearest centroid in terms of Euclidean distance. During the ‘minimize’ phase, centroids are moved in a direction that minimizes the sum of the distance of all datapoints assigned to it.[28] This completes a cycle. The next cycle begins by dis-

[25]But PCA itself is not that simple to understand.

[26]K-means (also called the Lloyd-Forgy algorithm) was first proposed by independently by S. P. Lloyd in [16] and E. W. Forgy in [17].

[27]Usually, in a predefined number of times, there are other tactics as well.

[28]Imagine that a centroid is pinned down and connected to all its datapoints with rubber bands, and then you unpin it from the surface. It will move so that the rubber bands are less tense in total (even though individual rubber bands may become *more* tense).

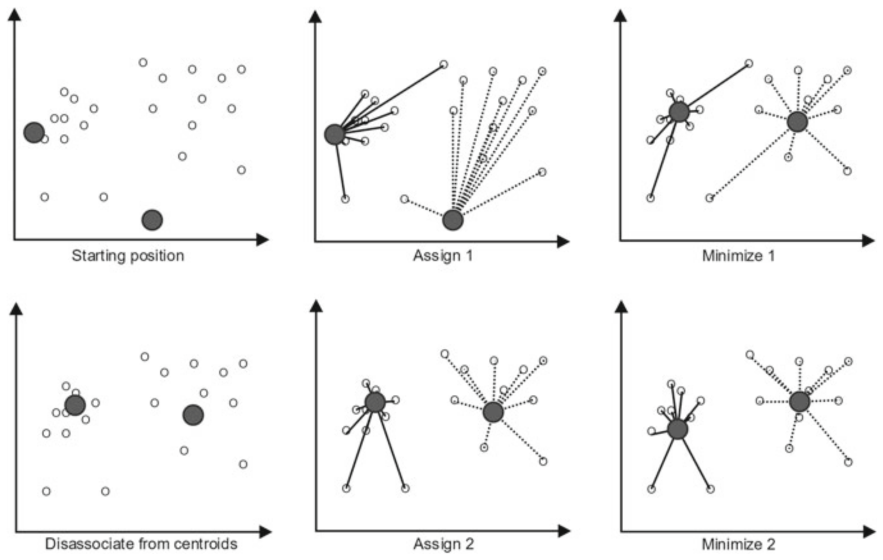

**Fig. 3.9** Two complete cycles of K-means with two centroids

associating all datapoints from centroids. Centroids stay where they are, but a new assignment phase begins, which may make a different assignment than the previous one. You can see this in Fig. 3.9. After the end of the cycles, we have a hyperplane ready: when we get a new datapoint, it will be assigned to the closest centroid. In other words, it will get the name of the closest centroid as a label.

In the usual setting, we do not have labels when using clustering (and we do not need them for unsupervised learning). The evaluation metrics we discussed in the previous sections are useless without labels since we cannot calculate the true positives, false positives, true negatives and false negatives. It can happen that we have access to labels but prefer to use clustering, or that we will obtain the true labels at a later time. In such case, we may evaluate the results of clustering as if they were classification results, and this is called *external evaluation of clustering*. A detailed exposition of using classification evaluation metrics for the external evaluation of clustering is given in [11].

But sometimes we do not have any labels and we must work without them. In such cases, we can use a class of evaluation metrics called *internal evaluation of clustering*. There are several evaluation metrics, but the Dunn coefficient [12] is the most popular. The main idea is to measure how dense the clusters are in $n$-dimensional space. So for each cluster[29] $C$ the Dunn coefficient is calculated by

$$D_C = \frac{min\{d(i, j)|i, j \in Centroids\}}{d^{in}(C)} \tag{3.15}$$

[29]Recall that a cluster in K-means is a region around a centroid separated by the hyperplane.

Here, $d(i, j)$ is the Euclidean distance between centroids $i$ and $j$ and $d^{in}(C)$ is the intra-cluster distance which is taken to be the distance:

$$d^{in}(C) = max\{d(x, y)|x, y \in C\}, \tag{3.16}$$

where $C$ is the cluster for which we calculate the Dunn coefficient. The Dunn coefficient is calculated for each cluster and the quality of each cluster can be assessed by it. The Dunn coefficient can be used to evaluate different clusterings by taking the average of the Dunn coefficients for each cluster in both clusterings[30] and then comparing them.

## 3.7 Learning Different Representations: PCA

The data we used so far has *local representations*. If the value of a feature named 'Height' is 180, then that piece of information about that datapoint (we could even say 'that property of the entity') exists only there. A different column 'Weight' contains no information on height. Such representations of the properties of the entities that we are describing as features of a datapoint are called *local representations*. Notice that the fact that the object has some height does put a constraint on weight. This is not a hard constraint but more of an 'epistemic shortcut': if we know that the person is 180 cm tall, then they will probably have around 80 kg. Individual persons may vary, but in general we could make a relatively decent guess of the person's weight just by knowing their height. This phenomenon is called *correlation* and it is a tricky phenomenon. If two features are highly correlated, they are very hard to tell apart. Ideally, we would want to find a transformation of the data which has weird features, but that are not correlated. In this representation, we would have a feature 'Argh' which captures the underlying component[31] by which we were able to deduce the weight from the height, and leave 'Haght' and 'Waght' as the part which is left in 'Height' and 'Weight' after 'Argh' was removed from them. Such representations are called *distributed representations*.

Building distributed representations by hand is hard, and yet this is the essence of what artificial neural networks do. Every layer builds its own distributed representation and this facilitates learning (this is perhaps the very essence of deep learning—learning many layers of distributed representations). We will show the simplest method of building a meaningful distributed representation, but we will write all the mathematical details of it only in Chap. 9. It is quite hard, and this is why we want deep learning to build such things for us. This method of building distributed representations is called the *principal component analysis* or PCA for short. In this chapter, we will provide a bird's-eye view of PCA and we will give all

[30] We have to use the same number of centroids in both clusterings for this to work.

[31] These features are known as *latent variables* in statistics.

the details in Chap. 9.[32] PCA has the following form:

$$Z = XQ, \tag{3.17}$$

where $X$ is the input matrix, $Z$ is the transformed matrix and $Q$ is the 'tool-matrix' with which we do the transformation. If $X$ is an $n \times d$ matrix, $Z$ should also be $n \times d$. This gives us our first information about $Q$: it has to be a $d \times d$ matrix for the multiplication to work. We will show how to find the appropriate $Q$ in Chap. 9. In the remainder of this section, we will introduce the intuition behind PCA as a whole and some of the elements needed to build $Q$. We will also describe in detail what do we want PCA to do and for what we want to be able to use it.

In general terms, PCA is used to preprocess the data. This means that it has to transform the data before the data is fed in a classifier, to make it more digestible. PCA is helpful for preprocessing in a couple of ways. We have seen above that we will use it to build distributed representations of data to eliminate correlation. We will also be able to use PCA for dimensionality reduction. We have seen how dimensions can expand with one-hot encoding and manual feature engineering. When we make distributed representations with artificial features such as 'Argh', 'Haght' and 'Waght', we would like to be able to order them in terms of informativity, so that we can discard the uninformative features. Informativity is just variance[33]: if a feature varies more, it carries more information.[34] This is something we want our $Z$ to be like: the feature that has the most variance should be in the first column, the one with the second most variance in the second column of $Z$ and so on.

To illustrate how the variance can change with simple transformations, see Fig. 3.10 where we have a simple case of six 2D datapoints. The part A of Fig. 3.10 illustrates the starting position. Note that the variance along the $x$ coordinate is relatively small: the projections of the datapoints on the $x$ axis are tightly packed together. The variance along the $y$ axis is better, and the $y$ coordinates are further apart. But we can do even better. Take a look at the part B of Fig. 3.10: we have obtained this by rotating the coordinate system a bit. Notice that all data stays the same and we are changing our representation of the data, i.e. the axes (which correspond to features). The new 'coordinate system' is actually, mathematically speaking, just a different basis for the points in this 2D vector space. You are not changing the points (i.e. 2D vectors), but the 'coordinate system' they live in. You are actually not even changing the coordinate system, but simply the basis of the vector space. The question of how to do this mathematically is actually the same as asking how to find a matrix $Q$ such that it behaves in this way, and we will answer this in Chap. 9. Along the axes, we have plotted the distance between the first and last datapoint coordinates, which may be seen as a 'graphical proxy' for variance. In the B part of

[32]One of the reasons for this is that we have not yet developed all the tools we need to write out the details now.

[33]See Chap. 2.

[34]And if a feature is always the same, it has a variance of 0 and it carries no information useful for drawing the hyperplane.

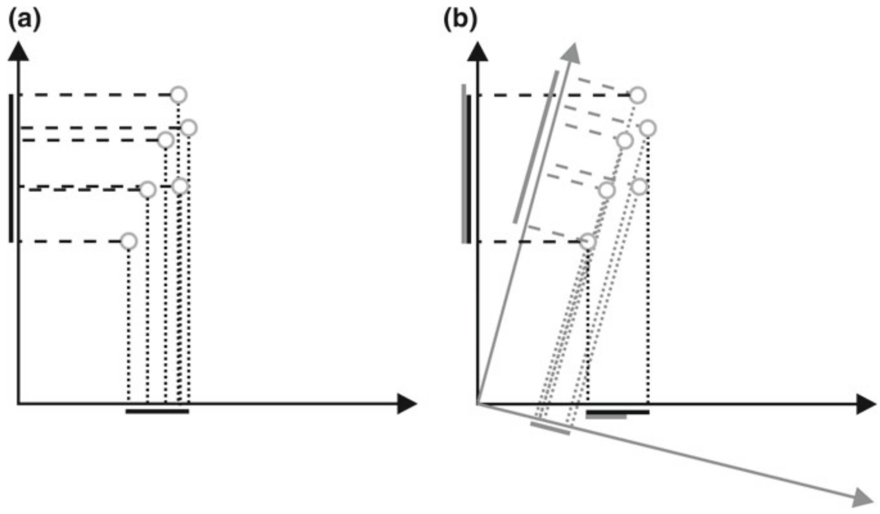

**Fig. 3.10** Variance under rotation of the coordinate system

Fig. 3.10, we have compared the black (original coordinate system) with the grey (transformed) variance side-by-side (next to the black coordinate system). Notice that the variance along the $y$ axis (the axis which had more variance in the original system) has increased, while the variance on the $x$ axis (the axis which had less variance in the original system) has actually decreased.

Before continuing, let us make a final remark about PCA and preprocessing. One of the most fundamental problems with any kind of data is that it is noisy. Noise can be defined as everything except relevant information. If our dataset has enough training samples, then it should have non-random information and random noise. They are usually mixed up in features. But if we can build a distributed representation, this means we can extract as separate features the parts which have more variance and part which have less variance; we could assume that noise (which is random) has low variance (it is 'equally random' everywhere), whereas information has high variance. Suppose we have used PCA on a 20-dimensional input matrix. Then, we can keep the first 10 new features and by doing so we have eliminated a lot of noise (low variance features) by eliminating only a little bit of information (since they are low variance features—not 'no variance' features).

PCA has been around a long time. It was first discovered by Karl Pearson of the University College London [13] in 1901. Since then variants of the PCA went by many names, and often there were subtle differences. The details of the relations between various variants of the PCA are interesting, but unfortunately they would require a whole book to explore, and consequently are beyond the scope of this volume.

## 3.8 Learning Language: The Bag of Words Representation

So far we have addressed numerical features, ordinal features and categorical features. We have seen how to do one-hot encoding for categorical features. We have not addressed a whole field, namely natural language processing. We refer the reader to [14] or [15] for a thorough introduction to natural language processing. In this section, we will see how to process language by using one of the simplest models, the *bag of words*.

Let us first define a couple of terms for natural language processing. A *corpus* is a whole collection of texts we have. A corpus can be decomposed into *fragments*. Fragments can be single sentences, paragraphs or multi-page documents. Basically, a fragment is something we wish to treat as a training sample. If we are analysing clinical documents, each patient admission document might be one fragment; if we are analysing all PhD theses from a major university, each 200-page thesis is one fragment; if we are analysing sentiment on social media, each user comment is one fragment; and so on. A bag of words model is made by turning each word from the *corpus* in a feature and in each row, under that word, counting how many times the word occurs in that *fragment*. Clearly, the order of the words is lost by creating a bag of words.

The bag of words model is one of the main ways to convert language in features to be fed to a machine learning algorithm, and only deep learning has good alternatives to it as we shall see in Chaps. 6, 7 and 8. Other machine learning methods use the bag of words or variations[35] almost exclusively, and for many language processing tasks, the bag of words is a great language model even in deep learning. Let us see how the bag of words works in a simple social media dataset[36]:

| User | Comment | Likes |
|---|---|---|
| S. A | you dont know | 22 |
| F. F | as if you know | 13 |
| S. A | i know what i know | 9 |
| P. H | i know | 43 |

We need to convert the column ‘Comment’ into a bag of words. The columns ‘User’ and ‘Likes’ are left as they are for now. To create a bag of words from the comments, we need to make two passes. The first just collects all the words that occur and turns them into features (i.e. collects the unique words and creates the columns from them) and the second writes in the actual values:

[35] An example of an expansion of the basic bag of words model is a bag of $n$-grams. An $n$-gram is a $n$-tuple consisting of $n$ words that occur next to each other. If we have a sentence ‘I will go now’, the set of its 2-grams will be $\{(`I', `will'), (`will', `go'), (`go', `now')\}$.

[36] For most language processing tasks, especially tasks requiring the use of data collected from social media, it makes sense to convert all text to lowercase first and get rid of all commas apostrophes and non-alphanumerics, which we have already done here.

| User | you | dont | know | as | if | i | what | Likes |
|---|---|---|---|---|---|---|---|---|
| S. A | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 22 |
| F. F | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 13 |
| S. A | 0 | 0 | 2 | 0 | 0 | 2 | 1 | 9 |
| P. H | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 43 |

Now, we have the bag of words of the column 'Comment' and we need to do one-hot encoding on the column 'User' before being able to feed the dataset in a machine learning algorithm. We do this as we have explained earlier and get the final input matrix:

| S. A | F. F | P. H | you | dont | know | as | if | i | what | Likes |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 22 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 13 |
| 1 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 2 | 1 | 9 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 43 |

This example shows the difference between one-hot encoding and the bag of words. In one-hot, each row has only 1 or 0 and, moreover, it must have exactly one 1. This means that it can be represented rather compactly by noting only the column number where it is 1. Take the fourth example in the upper column: we know everything for the one-hot part by simply noting '3' as the column number, which takes less space than writing '0,0,1'. The bag of words is different. Here, we take the word count for each fragment, which can be more than 1. Also, we need to use the bag of words on the entire dataset which means that we have to encode the training and test set together. This means that words that occur only in the test set will have 0 in the whole training set. Also, note that since most classifiers require that all samples have the same dimensionality (and feature names), when we will use the algorithm to predict, we will have to toss away any new word which is not in the trained model to be able to feed the data to the algorithm.

What they both have in common is that they expand the dimensions considerably and almost everywhere they will have the value 0. When we encode data like this we say, we have a *sparse encoding*. This means that a lot of features will be meaningless and that we want our classifier to dismiss them as soon as possible. We will see later how some techniques like PCA and $L_1$ regularization can be useful when confronted with a dataset which is sparsely encoded. Also, notice how we use the expansions of dimensions of the space to try to capture 'semantics' by counting words.

## References

1. R. Tibshirani, T. Hastie, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd edn. (Springer, New York, 2016)
2. F. van Harmelen, V. Lifschitz, B. Porter, *Handbook of Knowledge Representation* (Elsevier Science, New York, 2008)
3. R.S. Sutton, A.G. Barto, *Reinforcement Learning: An Introduction* (MIT Press, Cambridge, 1998)
4. J.R. Quinlan, Induction of decision trees. Mach. Learn. **1**, 81–106 (1986)
5. M.E. Maron, Automatic indexing: an experimental inquiry. J. ACM **8**(3), 404–417 (1961)
6. D.R. Cox, The regression analysis of binary sequences (with discussion). J. Roy. Stat. Soc. B (Methodol.) **20**(2), 215–242 (1958)
7. P.J. Grother, NIST special database 19: handprinted forms and characters database (1995)
8. Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
9. M.A. Nielsen, *Neural Networks and Deep Learning* (Determination Press, 2015)
10. P.N. Klein, *Coding the Matrix* (Newtonian Press, London, 2013)
11. I. Färber, S. Günnemann, H.P. Kriegel, P. Kroöger, E. Müller, E. Schubert, T. Seidl, A. Zimek. On using class-labels in evaluation of clusterings, in *MultiClust: Discovering, Summarizing, and Using Multiple Clusterings*, ed. by X.Z. Fern, I. Davidson, J. Dy (ACM SIGKDD, 2010)
12. J. Dunn, Well separated clusters and optimal fuzzy partitions. J. Cybern. **4**(1), 95–104 (1974)
13. K. Pearson, On lines and planes of closest fit to systems of points in space. Phil. Mag. **2**(11), 559–572 (1901)
14. C. Manning, H. Schütze, *Foundations of Statistical Natural Language Processing* (MIT Press, Cambridge, 1999)
15. D. Jurafsky, J. Martin, *Speech and Language Processing* (Prentice Hall, New Jersey, 2008)
16. S. P. Lloyd, Least squares quantization in PCM. *IEEE Transactions on Information Theory*, **28**(2), 129–137 (1982)
17. E. W. Forgy, Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *Biometrics*, **21**(3), 768–769 (1965)