1 *Introduction*

1.1 Machine learning: what and why?

We are drowning in information and starving for knowledge. — John Naisbitt.

We are entering the era of **big data**. For example, there are about 1 trillion web pages[1]; one hour of video is uploaded to YouTube every second, amounting to 10 years of content every day[2]; the genomes of 1000s of people, each of which has a length of 3.8×10^9 base pairs, have been sequenced by various labs; Walmart handles more than 1M transactions per hour and has databases containing more than 2.5 petabytes (2.5×10^{15}) of information (Cukier 2010); and so on.

This deluge of data calls for automated methods of data analysis, which is what **machine learning** provides. In particular, we define machine learning as a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty (such as planning how to collect more data!).

This books adopts the view that the best way to solve such problems is to use the tools of probability theory. Probability theory can be applied to any problem involving uncertainty. In machine learning, uncertainty comes in many forms: what is the best prediction about the future given some past data? what is the best model to explain some data? what measurement should I perform next? etc. The probabilistic approach to machine learning is closely related to the field of statistics, but differs slightly in terms of its emphasis and terminology[3].

We will describe a wide variety of probabilistic models, suitable for a wide variety of data and tasks. We will also describe a wide variety of algorithms for learning and using such models. The goal is not to develop a cook book of ad hoc techiques, but instead to present a unified view of the field through the lens of probabilistic modeling and inference. Although we will pay attention to computational efficiency, details on how to scale these methods to truly massive datasets are better described in other books, such as (Rajaraman and Ullman 2011; Bekkerman et al. 2011).

1. http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html
2. Source: http://www.youtube.com/t/press_statistics.
3. Rob Tibshirani, a statistician at Stanford university, has created an amusing comparison between machine learning and statistics, available at http://www-stat.stanford.edu/~tibs/stat315a/glossary.pdf.

It should be noted, however, that even when one has an apparently massive data set, the effective number of data points for certain cases of interest might be quite small. In fact, data across a variety of domains exhibits a property known as the **long tail**, which means that a few things (e.g., words) are very common, but most things are quite rare (see Section 2.4.6 for details). For example, 20% of Google searches each day have never been seen before[4]. This means that the core statistical issues that we discuss in this book, concerning generalizing from relatively small samples sizes, are still very relevant even in the big data era.

1.1.1 Types of machine learning

Machine learning is usually divided into two main types. In the **predictive** or **supervised learning** approach, the goal is to learn a mapping from inputs $\mathbf{x}$ to outputs y, given a labeled set of input-output pairs $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$. Here $\mathcal{D}$ is called the **training set**, and N is the number of training examples.

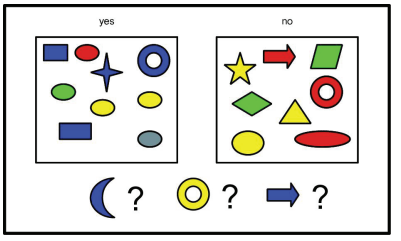
In the simplest setting, each training input $\mathbf{x}_i$ is a D-dimensional vector of numbers, representing, say, the height and weight of a person. These are called **features**, **attributes** or **covariates**. In general, however, $\mathbf{x}_i$ could be a complex structured object, such as an image, a sentence, an email message, a time series, a molecular shape, a graph, etc.

Similarly the form of the output or **response variable** can in principle be anything, but most methods assume that y_i is a **categorical** or **nominal** variable from some finite set, $y_i \in \{1, \ldots, C\}$ (such as male or female), or that y_i is a real-valued scalar (such as income level). When y_i is categorical, the problem is known as **classification** or **pattern recognition**, and when y_i is real-valued, the problem is known as **regression**. Another variant, known as **ordinal regression**, occurs where label space $\mathcal{Y}$ has some natural ordering, such as grades A–F.

The second main type of machine learning is the **descriptive** or **unsupervised learning** approach. Here we are only given inputs, $\mathcal{D} = \{\mathbf{x}_i\}_{i=1}^N$, and the goal is to find "interesting patterns" in the data. This is sometimes called **knowledge discovery**. This is a much less well-defined problem, since we are not told what kinds of patterns to look for, and there is no obvious error metric to use (unlike supervised learning, where we can compare our prediction of y for a given $\mathbf{x}$ to the observed value).

There is a third type of machine learning, known as **reinforcement learning**, which is somewhat less commonly used. This is useful for learning how to act or behave when given occasional reward or punishment signals. (For example, consider how a baby learns to walk.) Unfortunately, RL is beyond the scope of this book, although we do discuss decision theory in Section 5.7, which is the basis of RL. See e.g., (Kaelbling et al. 1996; Sutton and Barto 1998; Russell and Norvig 2010; Szepesvari 2010; Wiering and van Otterlo 2012) for more information on RL.

4. http://certifiedknowledge.org/blog/are-search-queries-becoming-even-more-unique-statistics-from-google.



(a)

D features (attributes)

N cases

Color	Shape	Size (cm)	Label
Blue	Square	10	1
Red	Ellipse	2.4	1
Red	Ellipse	20.7	0

(b)

Figure 1.1 Left: Some labeled training examples of colored shapes, along with 3 unlabeled test cases. Right: Representing the training data as an $N \times D$ design matrix. Row i represents the feature vector $\mathbf{x}_i$. The last column is the label, $y_i \in \{0, 1\}$. Based on a figure by Leslie Kaelbling.

1.2 Supervised learning

We begin our investigation of machine learning by discussing supervised learning, which is the form of ML most widely used in practice.

1.2.1 Classification

In this section, we discuss classification. Here the goal is to learn a mapping from inputs $\mathbf{x}$ to outputs y, where $y \in \{1, \ldots, C\}$, with C being the number of classes. If $C = 2$, this is called **binary classification** (in which case we often assume $y \in \{0, 1\}$); if $C > 2$, this is called **multiclass classification**. If the class labels are not mutually exclusive (e.g., somebody may be classified as tall and strong), we call it **multi-label classification**, but this is best viewed as predicting multiple related binary class labels (a so-called **multiple output model**). When we use the term "classification", we will mean multiclass classification with a single output, unless we state otherwise.

One way to formalize the problem is as **function approximation**. We assume $y = f(\mathbf{x})$ for some unknown function f, and the goal of learning is to estimate the function f given a labeled training set, and then to make predictions using $\hat{y} = \hat{f}(\mathbf{x})$. (We use the hat symbol to denote an estimate.) Our main goal is to make predictions on novel inputs, meaning ones that we have not seen before (this is called **generalization**), since predicting the response on the training set is easy (we can just look up the answer).

1.2.1.1 Example

As a simple toy example of classification, consider the problem illustrated in Figure 1.1(a). We have two classes of object which correspond to labels 0 and 1. The inputs are colored shapes. These have been described by a set of D features or attributes, which are stored in an $N \times D$ design matrix $\mathbf{X}$, shown in Figure 1.1(b). The input features $\mathbf{x}$ can be discrete, continuous or a combination of the two. In addition to the inputs, we have a vector of training labels $\mathbf{y}$.

In Figure 1.1, the test cases are a blue crescent, a yellow circle and a blue arrow. None of these have been seen before. Thus we are required to **generalize** beyond the training set. A

reasonable guess is that blue crescent should be $y = 1$, since all blue shapes are labeled 1 in the training set. The yellow circle is harder to classify, since some yellow things are labeled $y = 1$ and some are labeled $y = 0$, and some circles are labeled $y = 1$ and some $y = 0$. Consequently it is not clear what the right label should be in the case of the yellow circle. Similarly, the correct label for the blue arrow is unclear.

1.2.1.2 The need for probabilistic predictions

To handle ambiguous cases, such as the yellow circle above, it is desirable to return a probability. The reader is assumed to already have some familiarity with basic concepts in probability. If not, please consult Chapter 2 for a refresher, if necessary.

We will denote the probability distribution over possible labels, given the input vector $\mathbf{x}$ and training set $\mathcal{D}$ by $p(y|\mathbf{x}, \mathcal{D})$. In general, this represents a vector of length C. (If there are just two classes, it is sufficient to return the single number $p(y = 1|\mathbf{x}, \mathcal{D})$, since $p(y = 1|\mathbf{x}, \mathcal{D}) + p(y = 0|\mathbf{x}, \mathcal{D}) = 1$.) In our notation, we make explicit that the probability is conditional on the test input $\mathbf{x}$, as well as the training set $\mathcal{D}$, by putting these terms on the right hand side of the conditioning bar $|$. We are also implicitly conditioning on the form of model that we use to make predictions. When choosing between different models, we will make this assumption explicit by writing $p(y|\mathbf{x}, \mathcal{D}, M)$, where M denotes the model. However, if the model is clear from context, we will drop M from our notation for brevity.

Given a probabilistic output, we can always compute our "best guess" as to the "true label" using

$$\hat{y} = \hat{f}(\mathbf{x}) = \operatorname*{argmax}_{c=1}^{C} p(y = c|\mathbf{x}, \mathcal{D}) \tag{1.1}$$

This corresponds to the most probable class label, and is called the **mode** of the distribution $p(y|\mathbf{x}, \mathcal{D})$; it is also known as a **MAP estimate** (MAP stands for **maximum a posteriori**). Using the most probable label makes intuitive sense, but we will give a more formal justification for this procedure in Section 5.7.

Now consider a case such as the yellow circle, where $p(\hat{y}|\mathbf{x}, \mathcal{D})$ is far from 1.0. In such a case we are not very confident of our answer, so it might be better to say "I don't know" instead of returning an answer that we don't really trust. This is particularly important in domains such as medicine and finance where we may be risk averse, as we explain in Section 5.7. Another application where it is important to assess risk is when playing TV game shows, such as Jeopardy. In this game, contestants have to solve various word puzzles and answer a variety of trivia questions, but if they answer incorrectly, they lose money. In 2011, IBM unveiled a computer system called Watson which beat the top human Jeopardy champion. Watson uses a variety of interesting techniques (Ferrucci et al. 2010), but the most pertinent one for our present purposes is that it contains a module that estimates how confident it is of its answer. The system only chooses to "buzz in" its answer if sufficiently confident it is correct. Similarly, Google has a system known as SmartASS (ad selection system) that predicts the probability you will click on an ad based on your search history and other user and ad-specific features (Metz 2010). This probability is known as the **click-through rate** or **CTR**, and can be used to maximize expected profit. We will discuss some of the basic principles behind systems such as SmartASS later in this book.

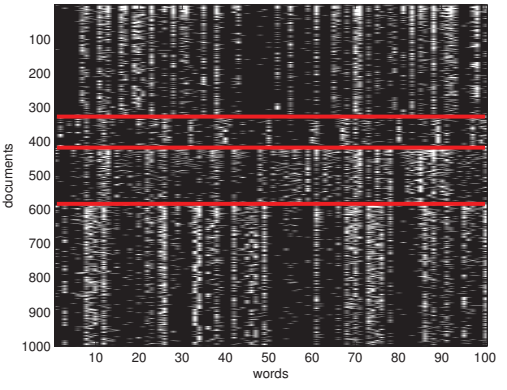


Figure 1.2 Subset of size 16242 x 100 of the 20-newsgroups data. We only show 1000 rows, for clarity. Each row is a document (represented as a bag-of-words bit vector), each column is a word. The red lines separate the 4 classes, which are (in descending order) comp, rec, sci, talk (these are the titles of USENET groups). We can see that there are subsets of words whose presence or absence is indicative of the class. The data is available from `http://cs.nyu.edu/~roweis/data.html`. Figure generated by `newsgroupsVisualize`.

1.2.1.3 Real-world applications

Classification is probably the most widely used form of machine learning, and has been used to solve many interesting and often difficult real-world problems. We have already mentioned some important applciations. We give a few more examples below.

Document classification and email spam filtering

In **document classification**, the goal is to classify a document, such as a web page or email message, into one of C classes, that is, to compute $p(y = c|\mathbf{x}, \mathcal{D})$, where $\mathbf{x}$ is some representation of the text. A special case of this is **email spam filtering**, where the classes are spam $y = 1$ or ham $y = 0$.

Most classifiers assume that the input vector $\mathbf{x}$ has a fixed size. A common way to represent variable-length documents in feature-vector format is to use a **bag of words** representation. This is explained in detail in Section 3.4.4.1, but the basic idea is to define $x_{ij} = 1$ iff word j occurs in document i. If we apply this transformation to every document in our data set, we get a binary document $\times$ word co-occurrence matrix: see Figure 1.2 for an example. Essentially the document classification problem has been reduced to one that looks for subtle changes in the pattern of bits. For example, we may notice that most spam messages have a high probability of containing the words "buy", "cheap", "viagra", etc. In Exercise 8.1 and Exercise 8.2, you will get hands-on experience applying various classification techniques to the spam filtering problem.

(a) (b) (c)

Figure 1.3 Three types of iris flowers: setosa, versicolor and virginica. Source: `http://www.statlab.uni-heidelberg.de/data/iris/` . Used with kind permission of Dennis Kramb and SIGNA.

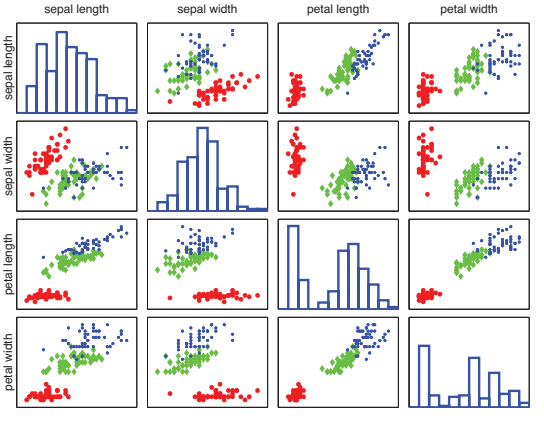


Figure 1.4 Visualization of the Iris data as a pairwise scatter plot. The diagonal plots the marginal histograms of the 4 features. The off diagonals contain scatterplots of all possible pairs of features. Red circle = setosa, green diamond = versicolor, blue star = virginica. Figure generated by `fisheririsDemo`.

Classifying flowers

Figure 1.3 gives another example of classification, due to the statistician Ronald Fisher. The goal is to learn to distinguish three different kinds of iris flower, called setosa, versicolor and virginica. Fortunately, rather than working directly with images, a botanist has already extracted 4 useful features or characteristics: sepal length and width, and petal length and width. (Such **feature extraction** is an important, but difficult, task. Most machine learning methods use features chosen by some human. Later we will discuss some methods that can learn good features from the data.) If we make a **scatter plot** of the iris data, as in Figure 1.4, we see that it is easy to distinguish setosas (red circles) from the other two classes by just checking if their petal length

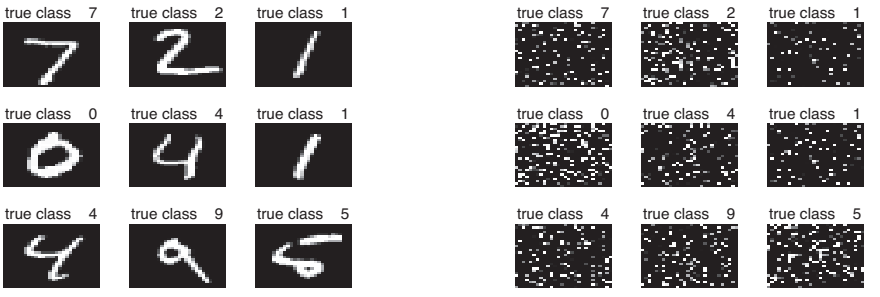


Figure 1.5 (a) First 9 test MNIST gray-scale images. (b) Same as (a), but with the features permuted randomly. Classification performance is identical on both versions of the data (assuming the training data is permuted in an identical way). Figure generated by `shuffledDigitsDemo`.

or width is below some threshold. However, distinguishing versicolor from virginica is slightly harder; any decision will need to be based on at least two features. (It is always a good idea to perform **exploratory data analysis**, such as plotting the data, before applying a machine learning method.)

Image classification and handwriting recognition

Now consider the harder problem of classifying images directly, where a human has not preprocessed the data. We might want to classify the image as a whole, e.g., is it an indoors or outdoors scene? is it a horizontal or vertical photo? does it contain a dog or not? This is called **image classification**.

In the special case that the images consist of isolated handwritten letters and digits, for example, in a postal or ZIP code on a letter, we can use classification to perform **handwriting recognition**. A standard dataset used in this area is known as **MNIST**, which stands for "Modified National Institute of Standards"[5]. (The term "modified" is used because the images have been preprocessed to ensure the digits are mostly in the center of the image.) This dataset contains 60,000 training images and 10,000 test images of the digits 0 to 9, as written by various people. The images are size 28×28 and have grayscale values in the range $0 : 255$. See Figure 1.5(a) for some example images.

Many generic classification methods ignore any structure in the input features, such as spatial layout. Consequently, they can also just as easily handle data that looks like Figure 1.5(b), which is the same data except we have randomly permuted the order of all the features. (You will verify this in Exercise 1.1.) This flexibility is both a blessing (since the methods are general purpose) and a curse (since the methods ignore an obviously useful source of information). We will discuss methods for exploiting structure in the input features later in the book.

5. Available from `http://yann.lecun.com/exdb/mnist/`.

(a) (b)

Figure 1.6 Example of face detection. (a) Input image (Murphy family, photo taken 5 August 2010). Used with kind permission of Bernard Diedrich of Sherwood Studios. (b) Output of classifier, which detected 5 faces at different poses. This was produced using the online demo at `http://demo.pittpatt.com/`. The classifier was trained on 1000s of manually labeled images of faces and non-faces, and then was applied to a dense set of overlapping patches in the test image. Only the patches whose probability of containing a face was sufficiently high were returned. Used with kind permission of Pittpatt.com

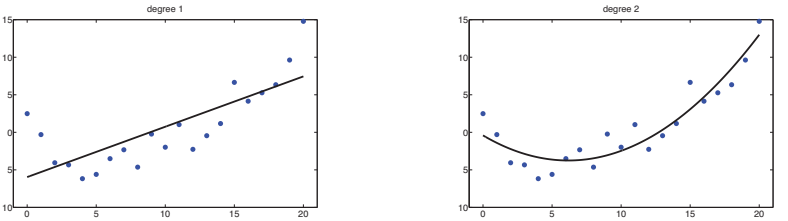
Face detection and recognition

A harder problem is to find objects within an image; this is called **object detection** or **object localization**. An important special case of this is **face detection**. One approach to this problem is to divide the image into many small overlapping patches at different locations, scales and orientations, and to classify each such patch based on whether it contains face-like texture or not. This is called a **sliding window detector**. The system then returns those locations where the probability of face is sufficiently high. See Figure 1.6 for an example. Such face detection systems are built-in to most modern digital cameras; the locations of the detected faces are used to determine the center of the auto-focus. Another application is automatically blurring out faces in Google's StreetView system.

Having found the faces, one can then proceed to perform **face recognition**, which means estimating the identity of the person (see Figure 1.10(a)). In this case, the number of class labels might be very large. Also, the features one should use are likely to be different than in the face detection problem: for recognition, subtle differences between faces such as hairstyle may be important for determining identity, but for detection, it is important to be **invariant** to such details, and to just focus on the differences between faces and non-faces. For more information about visual object detection, see e.g., (Szeliski 2010).

1.2.2 Regression

Regression is just like classification except the response variable is continuous. Figure 1.7 shows a simple example: we have a single real-valued input $x_i \in \mathbb{R}$, and a single real-valued response $y_i \in \mathbb{R}$. We consider fitting two models to the data: a straight line and a quadratic function. (We explain how to fit such models below.) Various extensions of this basic problem can arise, such as having high-dimensional inputs, outliers, non-smooth responses, etc. We will discuss ways to handle such problems later in the book.



(a) (b)

Figure 1.7 (a) Linear regression on some 1d data. (b) Same data with polynomial regression (degree 2). Figure generated by `linregPolyVsDegree`.

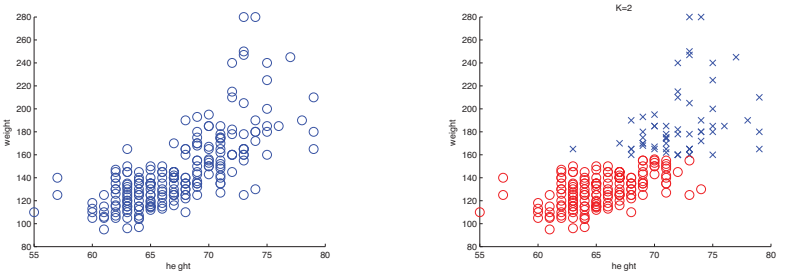
Here are some examples of real-world regression problems.

- Predict tomorrow's stock market price given current market conditions and other possible side information.
- Predict the age of a viewer watching a given video on YouTube.
- Predict the location in 3d space of a robot arm end effector, given control signals (torques) sent to its various motors.
- Predict the amount of prostate specific antigen (PSA) in the body as a function of a number of different clinical measurements.
- Predict the temperature at any location inside a building using weather data, time, door sensors, etc.

1.3 Unsupervised learning

We now consider **unsupervised learning**, where we are just given output data, without any inputs. The goal is to discover "interesting structure" in the data; this is sometimes called **knowledge discovery**. Unlike supervised learning, we are not told what the desired output is for each input. Instead, we will formalize our task as one of **density estimation**, that is, we want to build models of the form $p(\mathbf{x}_i|\boldsymbol{\theta})$. There are two differences from the supervised case. First, we have written $p(\mathbf{x}_i|\boldsymbol{\theta})$ instead of $p(y_i|\mathbf{x}_i, \boldsymbol{\theta})$; that is, supervised learning is conditional density estimation, whereas unsupervised learning is unconditional density estimation. Second, $\mathbf{x}_i$ is a vector of features, so we need to create multivariate probability models. By contrast, in supervised learning, y_i is usually just a single variable that we are trying to predict. This means that for most supervised learning problems, we can use univariate probability models (with input-dependent parameters), which significantly simplifies the problem. (We will discuss multi-output classification in Chapter 19, where we will see that it also involves multivariate probability models.)

Unsupervised learning is arguably more typical of human and animal learning. It is also more widely applicable than supervised learning, since it does not require a human expert to



(a) (b)

Figure 1.8 (a) The height and weight of some people. (b) A possible clustering using $K = 2$ clusters. Figure generated by `kmeansHeightWeight`.

manually label the data. Labeled data is not only expensive to acquire[6], but it also contains relatively little information, certainly not enough to reliably estimate the parameters of complex models. Geoff Hinton, who is a famous professor of ML at the University of Toronto, has said:

> When we're learning to see, nobody's telling us what the right answers are — we just look. Every so often, your mother says "that's a dog", but that's very little information. You'd be lucky if you got a few bits of information — even one bit per second — that way. The brain's visual system has 10^{14} neural connections. And you only live for 10^9 seconds. So it's no use learning one bit per second. You need more like 10^5 bits per second. And there's only one place you can get that much information: from the input itself. — Geoffrey Hinton, 1996 (quoted in (Gorder 2006)).

Below we describe some canonical examples of unsupervised learning.

1.3.1 Discovering clusters

As a canonical example of unsupervised learning, consider the problem of **clustering** data into groups. For example, Figure 1.8(a) plots some 2d data, representing the height and weight of a group of 210 people. It seems that there might be various clusters, or subgroups, although it is not clear how many. Let K denote the number of clusters. Our first goal is to estimate the distribution over the number of clusters, $p(K|\mathcal{D})$; this tells us if there are subpopulations within the data. For simplicity, we often approximate the distribution $p(K|\mathcal{D})$ by its mode, $K^* = \arg\max_K p(K|\mathcal{D})$. In the supervised case, we were told that there are two classes (male and female), but in the unsupervised case, we are free to choose as many or few clusters as we like. Picking a model of the "right" complexity is called model selection, and will be discussed in detail below.

Our second goal is to estimate which cluster each point belongs to. Let $z_i \in \{1, \ldots, K\}$ represent the cluster to which data point i is assigned. (z_i is an example of a **hidden** or

6. The advent of **crowd sourcing** web sites such as Mechanical Turk, (`https://www.mturk.com/mturk/welcome`), which outsource data processing tasks to humans all over the world, has reduced the cost of labeling data. Nevertheless, the amount of unlabeled data is still orders of magnitude larger than the amount of labeled data.

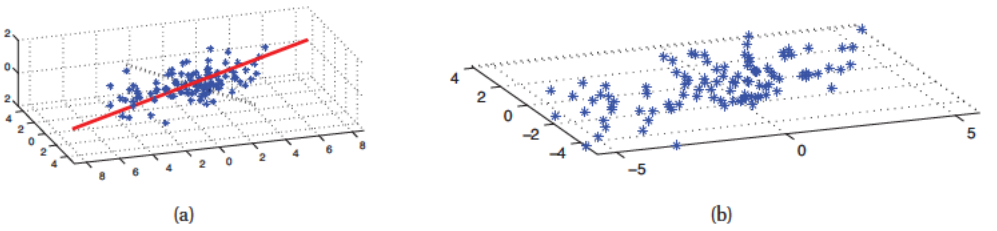


Figure 1.9 (a) A set of points that live on a 2d linear subspace embedded in 3d. The solid red line is the first principal component direction. The dotted black line is the second PC direction. (b) 2D representation of the data. Figure generated by `pcaDemo3d`.

latent variable, since it is never observed in the training set.) We can infer which cluster each data point belongs to by computing $z_i^* = \text{argmax}_k \, p(z_i = k|\mathbf{x}_i, \mathcal{D})$. This is illustrated in Figure 1.8(b), where we use different colors to indicate the assignments, assuming $K = 2$.

In this book, we focus on **model based clustering**, which means we fit a probabilistic model to the data, rather than running some ad hoc algorithm. The advantages of the model-based approach are that one can compare different kinds of models in an objective way (in terms of the likelihood they assign to the data), we can combine them together into larger systems, etc.

Here are some real world applications of clustering.

- In astronomy, the **autoclass** system (Cheeseman et al. 1988) discovered a new type of star, based on clustering astrophysical measurements.
- In e-commerce, it is common to cluster users into groups, based on their purchasing or web-surfing behavior, and then to send customized targeted advertising to each group (see e.g., (Berkhin 2006)).
- In biology, it is common to cluster flow-cytometry data into groups, to discover different sub-populations of cells (see e.g., (Lo et al. 2009)).

1.3.2 Discovering latent factors

When dealing with high dimensional data, it is often useful to reduce the dimensionality by projecting the data to a lower dimensional subspace which captures the "essence" of the data. This is called **dimensionality reduction**. A simple example is shown in Figure 1.9, where we project some 3d data down to a 2d plane. The 2d approximation is quite good, since most points lie close to this subspace. Reducing to 1d would involve projecting points onto the red line in Figure 1.9(a); this would be a rather poor approximation. (We will make this notion precise in Chapter 12.)

The motivation behind this technique is that although the data may appear high dimensional, there may only be a small number of degrees of variability, corresponding to **latent factors**. For example, when modeling the appearance of face images, there may only be a few underlying latent factors which describe most of the variability, such as lighting, pose, identity, etc, as illustrated in Figure 1.10.

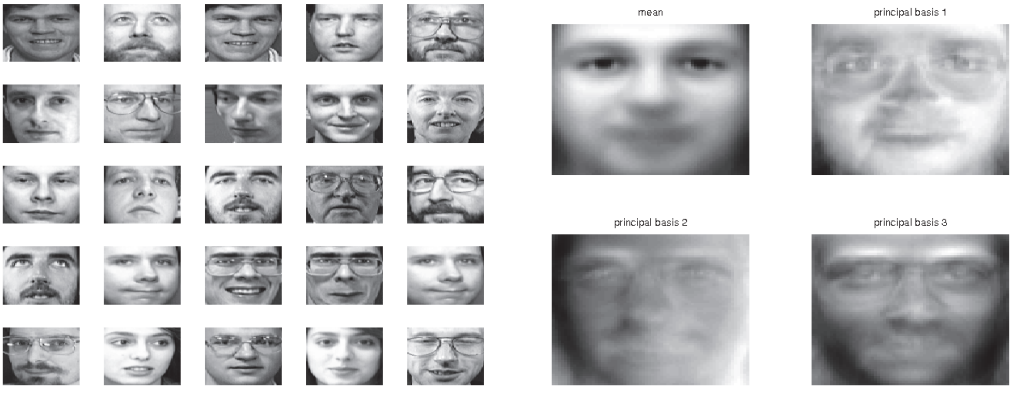


Figure 1.10 a) 25 randomly chosen 64×64 pixel images from the Olivetti face database. (b) The mean and the first three principal component basis vectors (eigenfaces). Figure generated by `pcaImageDemo`.

When used as input to other statistical models, such low dimensional representations often result in better predictive accuracy, because they focus on the "essence" of the object, filtering out inessential features. Also, low dimensional representations are useful for enabling fast nearest neighbor searches and two dimensional projections are very useful for **visualizing** high dimensional data.

The most common approach to dimensionality reduction is called **principal components analysis** or **PCA**. This can be thought of as an unsupervised version of (multi-output) linear regression, where we observe the high-dimensional response $\mathbf{y}$, but not the low-dimensional "cause" $\mathbf{z}$. Thus the model has the form $\mathbf{z} \rightarrow \mathbf{y}$; we have to "invert the arrow", and infer the latent low-dimensional $\mathbf{z}$ from the observed high-dimensional $\mathbf{y}$. See Section 12.1 for details.

Dimensionality reduction, and PCA in particular, has been applied in many different areas. Some examples include the following:

- In biology, it is common to use PCA to interpret gene microarray data, to account for the fact that each measurement is usually the result of many genes which are correlated in their behavior by the fact that they belong to different biological pathways.
- In natural language processing, it is common to use a variant of PCA called latent semantic analysis for document retrieval (see Section 27.2.2).
- In signal processing (e.g., of acoustic or neural signals), it is common to use ICA (which is a variant of PCA) to separate signals into their different sources (see Section 12.6).
- In computer graphics, it is common to project motion capture data to a low dimensional space, and use it to create animations. See Section 15.5 for one way to tackle such problems.

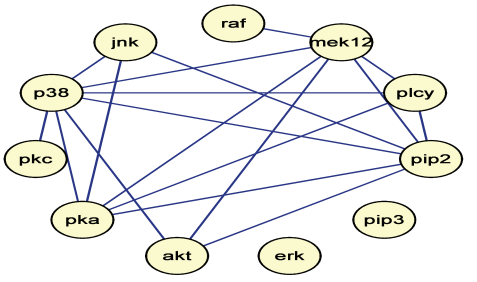


Figure 1.11 A sparse undirected Gaussian graphical model learned using graphical lasso (Section 26.7.2) applied to some flow cytometry data (from (Sachs et al. 2005)), which measures the phosphorylation status of 11 proteins. Figure generated by `ggmLassoDemo`.

1.3.3 Discovering graph structure

Sometimes we measure a set of correlated variables, and we would like to discover which ones are most correlated with which others. This can be represented by a graph G, in which nodes represent variables, and edges represent direct dependence between variables (we will make this precise in Chapter 10, when we discuss graphical models). We can then learn this graph structure from data, i.e., we compute $\hat{G} = \operatorname{argmax} p(G|\mathcal{D})$.

As with unsupervised learning in general, there are two main applications for learning sparse graphs: to discover new knowledge, and to get better joint probability density estimators. We now give somes example of each.

- Much of the motivation for learning sparse graphical models comes from the systems biology community. For example, suppose we measure the phosphorylation status of some proteins in a cell (Sachs et al. 2005). Figure 1.11 gives an example of a graph structure that was learned from this data (using methods discussed in Section 26.7.2). As another example, Smith et al. (2006) showed that one can recover the neural "wiring diagram" of a certain kind of bird from time-series EEG data. The recovered structure closely matched the known functional connectivity of this part of the bird brain.

- In some cases, we are not interested in interpreting the graph structure, we just want to use it to model correlations and to make predictions. One example of this is in financial portfolio management, where accurate models of the covariance between large numbers of different stocks is important. Carvalho and West (2007) show that by learning a sparse graph, and then using this as the basis of a trading strategy, it is possible to outperform (i.e., make more money than) methods that do not exploit sparse graphs. Another example is predicting traffic jams on the freeway. Horvitz et al. (2005) describe a deployed system called JamBayes for predicting traffic flow in the Seattle area; predictions are made using a graphical model whose structure was learned from data.

(a) (b)

Figure 1.12 (a) A noisy image with an occluder. (b) An estimate of the underlying pixel intensities, based on a pairwise MRF model. Source: Figure 8 of (Felzenszwalb and Huttenlocher 2006). Used with kind permission of Pedro Felzenszwalb.

1.3.4 Matrix completion

Sometimes we have missing data, that is, variables whose values are unknown. For example, we might have conducted a survey, and some people might not have answered certain questions. Or we might have various sensors, some of which fail. The corresponding design matrix will then have "holes" in it; these missing entries are often represented by **NaN**, which stands for "not a number". The goal of **imputation** is to infer plausible values for the missing entries. This is sometimes called **matrix completion**. Below we give some example applications.

1.3.4.1 Image inpainting

An interesting example of an imputation-like task is known as **image inpainting**. The goal is to "fill in" holes (e.g., due to scratches or occlusions) in an image with realistic texture. This is illustrated in Figure 1.12, where we denoise the image, as well as impute the pixels hidden behind the occlusion. This can be tackled by building a joint probability model of the pixels, given a set of clean images, and then inferring the unknown variables (pixels) given the known variables (pixels). This is somewhat like masket basket analysis, except the data is real-valued and spatially structured, so the kinds of probability models we use are quite different. See Sections 19.6.2.7 and 13.8.4 for some possible choices.

1.3.4.2 Collaborative filtering

Another interesting example of an imputation-like task is known as **collaborative filtering**. A common example of this concerns predicting which movies people will want to watch based on how they, and other people, have rated movies which they have already seen. The key idea is that the prediction is not based on features of the movie or user (although it could be), but merely on a ratings matrix. More precisely, we have a matrix $\mathbf{X}$ where $X(m, u)$ is the rating

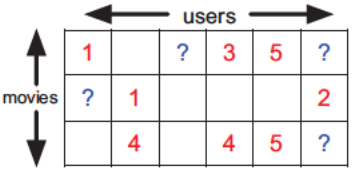


Figure 1.13 Example of movie-rating data. Training data is in red, test data is denoted by ?, empty cells are unknown.

(say an integer between 1 and 5, where 1 is dislike and 5 is like) by user u of movie m. Note that most of the entries in $\mathbf{X}$ will be missing or unknown, since most users will not have rated most movies. Hence we only observe a tiny subset of the $\mathbf{X}$ matrix, and we want to predict a different subset. In particular, for any given user u, we might want to predict which of the unrated movies he/she is most likely to want to watch.

In order to encourage research in this area, the DVD rental company Netflix created a competition, launched in 2006, with a $1M USD prize (see `http://netflixprize.com/`). In particular, they provided a large matrix of ratings, on a scale of 1 to 5, for $\sim 18k$ movies created by $\sim 500k$ users. The full matrix would have $\sim 9 \times 10^9$ entries, but only about 1% of the entries are observed, so the matrix is extremely **sparse**. A subset of these are used for training, and the rest for testing, as shown in Figure 1.13. The goal of the competition was to predict more accurately than Netflix's existing system. On 21 September 2009, the prize was awarded to a team of researchers known as "BellKor's Pragmatic Chaos". Section 27.6.2 discusses some of their methodology. Further details on the teams and their methods can be found at `http://www.netflixprize.com/community/viewtopic.php?id=1537`.

1.3.4.3 Market basket analysis

In commercial data mining, there is much interest in a task called **market basket analysis**. The data consists of a (typically very large but sparse) binary matrix, where each column represents an item or product, and each row represents a transaction. We set $x_{ij} = 1$ if item j was purchased on the i'th transaction. Many items are purchased together (e.g., bread and butter), so there will be correlations amongst the bits. Given a new partially observed bit vector, representing a subset of items that the consumer has bought, the goal is to predict which other bits are likely to turn on, representing other items the consumer might be likely to buy. (Unlike collaborative filtering, we often assume there is no missing data in the training data, since we know the past shopping behavior of each customer.)

This task arises in other domains besides modeling purchasing patterns. For example, similar techniques can be used to model dependencies between files in complex software systems. In this case, the task is to predict, given a subset of files that have been changed, which other ones need to be updated to ensure consistency (see e.g., (Hu et al. 2010)).

It is common to solve such tasks using **frequent itemset mining**, which create association rules (see e.g., (Hastie et al. 2009, sec 14.2) for details). Alternatively, we can adopt a probabilistic approach, and fit a joint density model $p(x_1, \ldots, x_D)$ to the bit vectors, see e.g., (Hu et al.

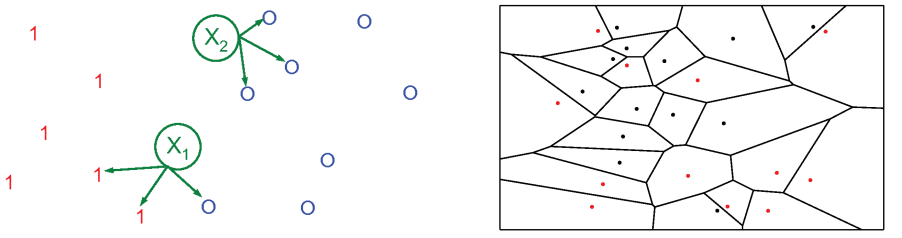


Figure 1.14 (a) Illustration of a K-nearest neighbors classifier in 2d for $K = 3$. The 3 nearest neighbors of test point $\mathbf{x}_1$ have labels 1, 1 and 0, so we predict $p(y = 1|\mathbf{x}_1, \mathcal{D}, K = 3) = 2/3$. The 3 nearest neighbors of test point x_2 have labels 0, 0, and 0, so we predict $p(y = 1|\mathbf{x}_2, \mathcal{D}, K = 3) = 0/3$. (b) Illustration of the Voronoi tesselation induced by 1-NN. Based on Figure 4.13 of (Duda et al. 2001). Figure generated by `knnVoronoi`.

2010). Such models often have better predictive acccuracy than association rules, although they may be less interpretible. This is typical of the difference between data mining and machine learning: in data mining, there is more emphasis on interpretable models, whereas in machine learning, there is more emphasis on accurate models.

1.4 Some basic concepts in machine learning

In this Section, we provide an introduction to some key ideas in machine learning. We will expand on these concepts later in the book, but we introduce them briefly here, to give a flavor of things to come.

1.4.1 Parametric vs non-parametric models

In this book, we will be focussing on probabilistic models of the form $p(y|\mathbf{x})$ or $p(\mathbf{x})$, depending on whether we are interested in supervised or unsupervised learning respectively. There are many ways to define such models, but the most important distinction is this: does the model have a fixed number of parameters, or does the number of parameters grow with the amount of training data? The former is called a **parametric model**, and the latter is called a **non-parametric model**. Parametric models have the advantage of often being faster to use, but the disadvantage of making stronger assumptions about the nature of the data distributions. Non-parametric models are more flexible, but often computationally intractable for large datasets. We will give examples of both kinds of models in the sections below. We focus on supervised learning for simplicity, although much of our discussion also applies to unsupervised learning.

1.4.2 A simple non-parametric classifier: K-nearest neighbors

A simple example of a non-parametric classifier is the K **nearest neighbor** (**KNN**) classifier. This simply "looks at" the K points in the training set that are nearest to the test input $\mathbf{x}$,

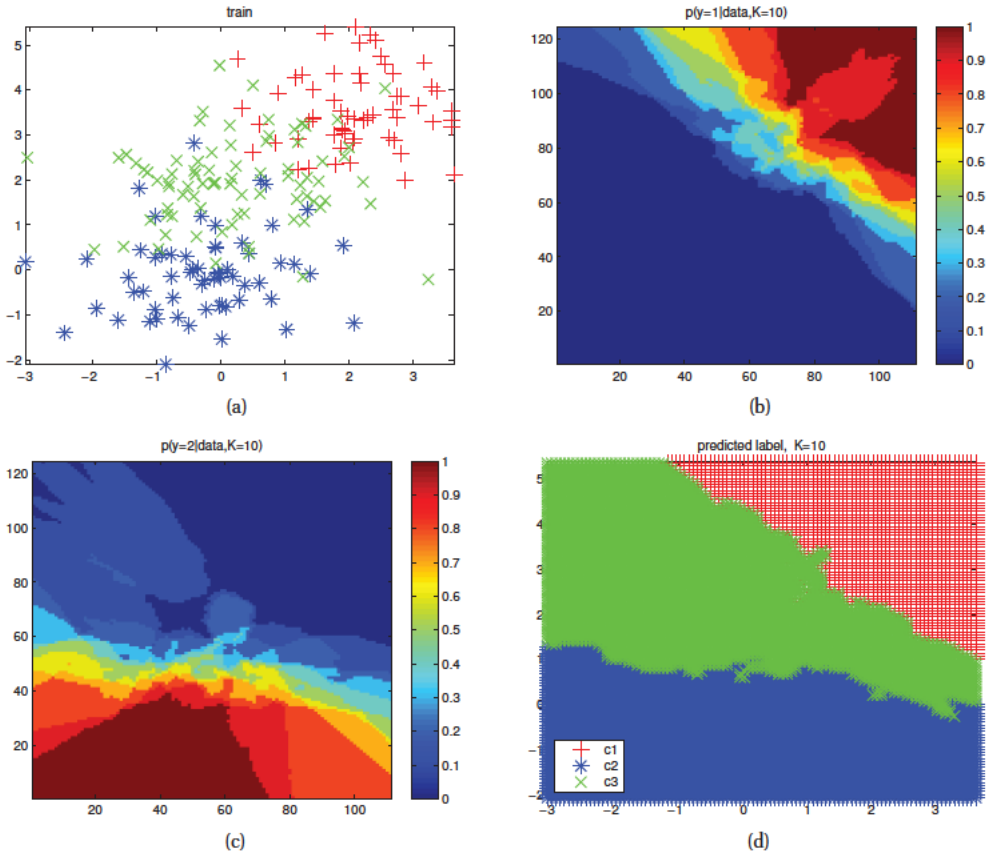


Figure 1.15 (a) Some synthetic 3-class training data in 2d. (b) Probability of class 1 for KNN with $K = 10$. (c) Probability of class 2. (d) MAP estimate of class label. Figure generated by `knnClassifyDemo`.

counts how many members of each class are in this set, and returns that empirical fraction as the estimate, as illustrated in Figure 1.14. More formally,

$$p(y = c|\mathbf{x}, \mathcal{D}, K) = \frac{1}{K} \sum_{i \in N_K(\mathbf{x}, \mathcal{D})} \mathbb{I}(y_i = c) \tag{1.2}$$

where $N_K(\mathbf{x}, \mathcal{D})$ are the (indices of the) K nearest points to $\mathbf{x}$ in $\mathcal{D}$ and $\mathbb{I}(e)$ is the **indicator function** defined as follows:

$$\mathbb{I}(e) = \begin{cases} 1 & \text{if } e \text{ is true} \\ 0 & \text{if } e \text{ is false} \end{cases} \tag{1.3}$$

This method is an example of **memory-based learning** or **instance-based learning**. It can be derived from a probabilistic framework as explained in Section 14.7.3. The most common

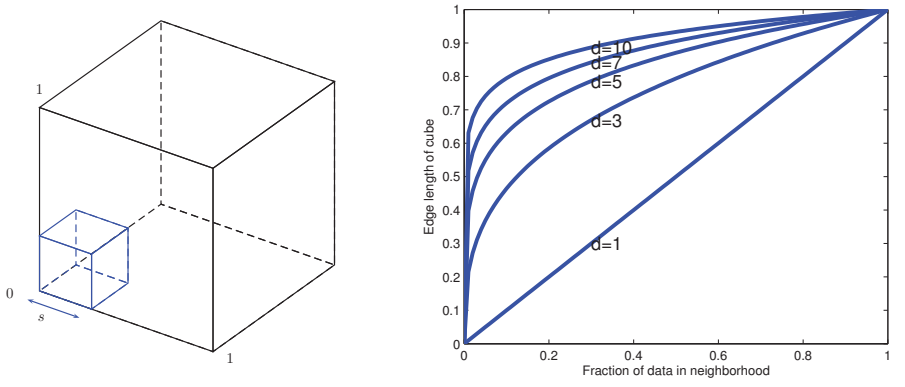


Figure 1.16 Illustration of the curse of dimensionality. (a) We embed a small cube of side s inside a larger unit cube. (b) We plot the edge length of a cube needed to cover a given volume of the unit cube as a function of the number of dimensions. Based on Figure 2.6 from (Hastie et al. 2009). Figure generated by `curseDimensionality`.

distance metric to use is Euclidean distance (which limits the applicability of the technique to data which is real-valued), although other metrics can be used.

Figure 1.15 gives an example of the method in action, where the input is two dimensional, we have three classes, and $K = 10$. (We discuss the effect of K below.) Panel (a) plots the training data. Panel (b) plots $p(y = 1|\mathbf{x}, \mathcal{D})$ where $\mathbf{x}$ is evaluated on a grid of points. Panel (c) plots $p(y = 2|\mathbf{x}, \mathcal{D})$. We do not need to plot $p(y = 3|\mathbf{x}, \mathcal{D})$, since probabilities sum to one. Panel (d) plots the MAP estimate $\hat{y}(\mathbf{x}) = \text{argmax}_c(y = c|\mathbf{x}, \mathcal{D})$.

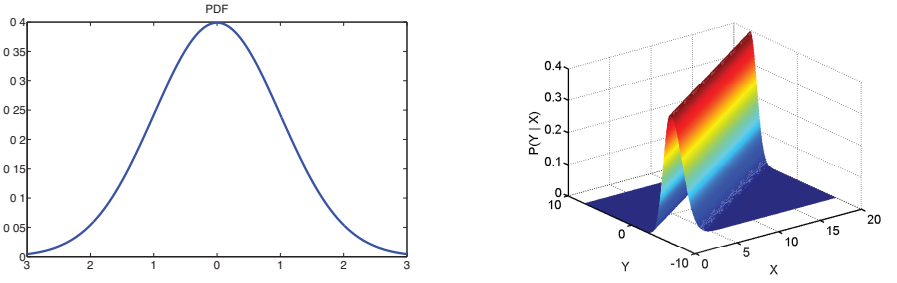
A KNN classifier with $K = 1$ induces a **Voronoi tessellation** of the points (see Figure 1.14(b)). This is a partition of space which associates a region $V(\mathbf{x}_i)$ with each point $\mathbf{x}_i$ in such a way that all points in $V(\mathbf{x}_i)$ are closer to $\mathbf{x}_i$ than to any other point. Within each cell, the predicted label is the label of the corresponding training point.

1.4.3 The curse of dimensionality

The KNN classifier is simple and can work quite well, provided it is given a good distance metric and has enough labeled training data. In fact, it can be shown that the KNN classifier can come within a factor of 2 of the best possible performance if $N \to \infty$ (Cover and Hart 1967).

However, the main problem with KNN classifiers is that they do not work well with high dimensional inputs. The poor performance in high dimensional settings is due to the **curse of dimensionality**.

To explain the curse, we give some examples from (Hastie et al. 2009, p22). Consider applying a KNN classifier to data where the inputs are uniformly distributed in the D-dimensional unit cube. Suppose we estimate the density of class labels around a test point $\mathbf{x}$ by "growing" a hyper-cube around $\mathbf{x}$ until it contains a desired fraction f of the data points. The expected edge length of this cube will be $e_D(f) = f^{1/D}$. If $D = 10$, and we want to base our estimate on 10%



(a) (b)

Figure 1.17 (a) A Gaussian pdf with mean 0 and variance 1. Figure generated by `gaussPlotDemo`. (b) Visualization of the conditional density model $p(y|x, \boldsymbol{\theta}) = \mathcal{N}(y|w_0 + w_1 x, \sigma^2)$. The density falls off exponentially fast as we move away from the regression line. Figure generated by `linregWedgeDemo2`.

of the data, we have $e_{10}(0.1) = 0.8$, so we need to extend the cube 80% along each dimension around $\mathbf{x}$. Even if we only use 1% of the data, we find $e_{10}(0.01) = 0.63$: see Figure 1.16. Since the entire range of the data is only 1 along each dimension, we see that the method is no longer very local, despite the name "nearest neighbor". The trouble with looking at neighbors that are so far away is that they may not be good predictors about the behavior of the input-output function at a given point.

1.4.4 Parametric models for classification and regression

The main way to combat the curse of dimensionality is to make some assumptions about the nature of the data distribution (either $p(y|\mathbf{x})$ for a supervised problem or $p(\mathbf{x})$ for an unsupervised problem). These assumptions, known as **inductive bias**, are often embodied in the form of a **parametric model**, which is a statistical model with a fixed number of parameters. Below we briefly describe two widely used examples; we will revisit these and other models in much greater depth later in the book.

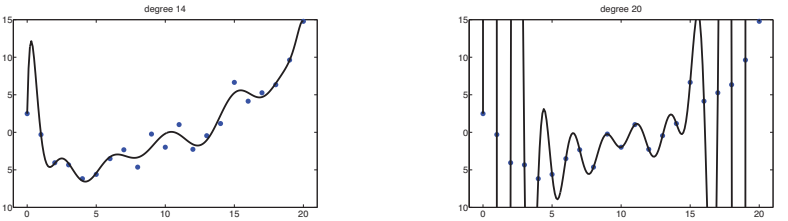
1.4.5 Linear regression

One of the most widely used models for regression is known as **linear regression**. This asserts that the response is a linear function of the inputs. This can be written as follows:

$$y(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + \epsilon = \sum_{j=1}^{D} w_j x_j + \epsilon \tag{1.4}$$

where $\mathbf{w}^T \mathbf{x}$ represents the inner or **scalar product** between the input vector $\mathbf{x}$ and the model's **weight vector** $\mathbf{w}$[7], and ϵ is the **residual error** between our linear predictions and the true response.

7. In statistics, it is more common to denote the regression weights by $\boldsymbol{\beta}$.



(a) (b)

Figure 1.18 Polynomial of degrees 14 and 20 fit by least squares to 21 data points. Figure generated by `linregPolyVsDegree`.

We often assume that ϵ has a **Gaussian**[8] or **normal** distribution. We denote this by $\epsilon \sim \mathcal{N}(\mu, \sigma^2)$, where μ is the mean and σ^2 is the variance (see Chapter 2 for details). When we plot this distribution, we get the well-known **bell curve** shown in Figure 1.17(a).

To make the connection between linear regression and Gaussians more explicit, we can rewrite the model in the following form:

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|\mu(\mathbf{x}), \sigma^2(\mathbf{x})) \tag{1.5}$$

This makes it clear that the model is a conditional probability density. In the simplest case, we assume μ is a linear function of $\mathbf{x}$, so $\mu = \mathbf{w}^T\mathbf{x}$, and that the noise is fixed, $\sigma^2(x) = \sigma^2$. In this case, $\boldsymbol{\theta} = (\mathbf{w}, \sigma^2)$ are the parameters of the model.

For example, suppose the input is 1 dimensional. We can represent the expected response as follows:

$$\mu(\mathbf{x}) = w_0 + w_1 x = \mathbf{w}^T\mathbf{x} \tag{1.6}$$

where w_0 is the intercept or **bias** term, w_1 is the slope, and where we have defined the vector $\mathbf{x} = (1, x)$. (Prepending a constant 1 term to an input vector is a common notational trick which allows us to combine the intercept term with the other terms in the model.) If w_1 is positive, it means we expect the output to increase as the input increases. This is illustrated in 1d in Figure 1.17(b); a more conventional plot, of the mean response vs x, is shown in Figure 1.7(a).

Linear regression can be made to model non-linear relationships by replacing $\mathbf{x}$ with some non-linear function of the inputs, $\boldsymbol{\phi}(\mathbf{x})$. That is, we use

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}), \sigma^2) \tag{1.7}$$

This is known as **basis function expansion**. For example, Figure 1.18 illustrates the case where $\boldsymbol{\phi}(\mathbf{x}) = [1, x, x^2, \ldots, x^d]$, for $d = 14$ and $d = 20$; this is known as **polynomial regression**. We will consider other kinds of basis functions later in the book. In fact, many popular machine learning methods — such as support vector machines, neural networks, classification and regression trees, etc. — can be seen as just different ways of estimating basis functions from data, as we discuss in Chapters 14 and 16.

8. Carl Friedrich Gauss (1777–1855) was a German mathematician and physicist.

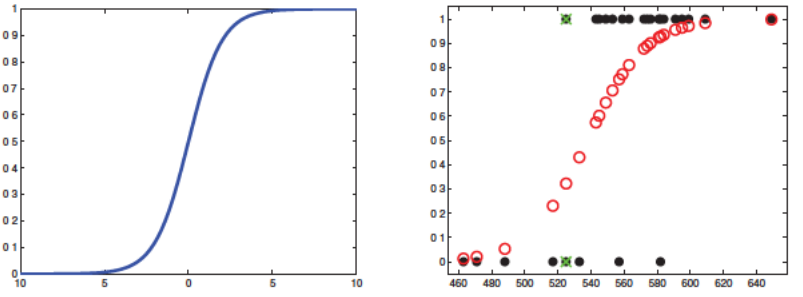


Figure 1.19 (a) The sigmoid or logistic function. We have $\text{sigm}(-\infty) = 0$, $\text{sigm}(0) = 0.5$, and $\text{sigm}(\infty) = 1$. Figure generated by `sigmoidPlot`. (b) Logistic regression for SAT scores. Solid black dots are the data. The open red circles are the predicted probabilities. The green crosses denote two students with the same SAT score of 525 (and hence same input representation x) but with different training labels (one student passed, $y = 1$, the other failed, $y = 0$). Hence this data is not perfectly separable using just the SAT feature. Figure generated by `logregSATdemo`.

1.4.6 Logistic regression

We can generalize linear regression to the (binary) classification setting by making two changes. First we replace the Gaussian distribution for y with a **Bernoulli** distribution[9],which is more appropriate for the case when the response is binary, $y \in \{0, 1\}$. That is, we use

$$p(y|\mathbf{x}, \mathbf{w}) = \text{Ber}(y|\mu(\mathbf{x})) \tag{1.8}$$

where $\mu(\mathbf{x}) = \mathbb{E}[y|\mathbf{x}] = p(y = 1|\mathbf{x})$. Second, we compute a linear combination of the inputs, as before, but then we pass this through a function that ensures $0 \leq \mu(\mathbf{x}) \leq 1$ by defining

$$\mu(\mathbf{x}) = \text{sigm}(\mathbf{w}^T\mathbf{x}) \tag{1.9}$$

where $\text{sigm}(\eta)$ refers to the **sigmoid** function, also known as the **logistic** or **logit** function. This is defined as

$$\text{sigm}(\eta) \triangleq \frac{1}{1 + \exp(-\eta)} = \frac{e^\eta}{e^\eta + 1} \tag{1.10}$$

The term "sigmoid" means S-shaped: see Figure 1.19(a) for a plot. It is also known as a **squashing function**, since it maps the whole real line to $[0, 1]$, which is necessary for the output to be interpreted as a probability.

Putting these two steps together we get

$$p(y|\mathbf{x}, \mathbf{w}) = \text{Ber}(y|\text{sigm}(\mathbf{w}^T\mathbf{x})) \tag{1.11}$$

This is called **logistic regression** due to its similarity to linear regression (although it is a form of classification, not regression!).

9. Daniel Bernoulli (1700–1782) was a Dutch-Swiss mathematician and physicist.

A simple example of logistic regression is shown in Figure 1.19(b), where we plot

$$p(y_i = 1|x_i, \mathbf{w}) = \text{sigm}(w_0 + w_1 x_i) \tag{1.12}$$

where x_i is the SAT[10] score of student i and y_i is whether they passed or failed a class. The solid black dots show the training data, and the red circles plot $p(y = 1|\mathbf{x}_i, \hat{\mathbf{w}})$, where $\hat{\mathbf{w}}$ are the parameters estimated from the training data (we discuss how to compute these estimates in Section 8.3.4).

If we threshold the output probability at 0.5, we can induce a **decision rule** of the form

$$\hat{y}(x) = 1 \iff p(y = 1|\mathbf{x}) > 0.5 \tag{1.13}$$

By looking at Figure 1.19(b), we see that $\text{sigm}(w_0 + w_1 x) = 0.5$ for $x \approx 545 = x^*$. We can imagine drawing a vertical line at $x = x^*$; this is known as a decision boundary. Everything to the left of this line is classified as a 0, and everything to the right of the line is classified as a 1.

We notice that this decision rule has a non-zero error rate even on the training set. This is because the data is not **linearly separable**, i.e., there is no straight line we can draw to separate the 0s from the 1s. We can create models with non-linear decision boundaries using basis function expansion, just as we did with non-linear regression. We will see many examples of this later in the book.

1.4.7 Overfitting

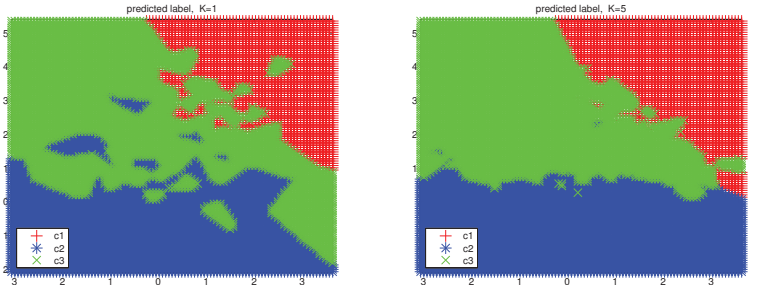
When we fit highly flexible models, we need to be careful that we do not **overfit** the data, that is, we should avoid trying to model every minor variation in the input, since this is more likely to be noise than true signal. This is illustrated in Figure 1.18(b), where we see that using a high degree polynomial results in a curve that is very "wiggly". It is unlikely that the true function has such extreme oscillations. Thus using such a model might result in accurate predictions of future outputs.

As another example, consider the KNN classifier. The value of K can have a large effect on the behavior of this model. When $K = 1$, the method makes no errors on the training set (since we just return the labels of the original training points), but the resulting prediction surface is very "wiggly" (see Figure 1.20(a)). Therefore the method may not work well at predicting future data. In Figure 1.20(b), we see that using $K = 5$ results in a smoother prediction surface, because we are averaging over a larger neighborhood. As K increases, the predictions becomes smoother until, in the limit of $K = N$, we end up predicting the majority label of the whole data set. Below we discuss how to pick the "right" value of K.

1.4.8 Model selection

When we have a variety of models of different complexity (e.g., linear or logistic regression models with different degree polynomials, or KNN classifiers with different values of K), how should we pick the right one? A natural approach is to compute the **misclassification rate** on

10. SAT stands for "Scholastic Aptitude Test". This is a standardized test for college admissions used in the United States (the data in this example is from (Johnson and Albert 1999, p87)).



(a) (b)

Figure 1.20 Prediction surface for KNN on the data in Figure 1.15(a). (a) K=1. (b) K=5. Figure generated by `knnClassifyDemo`.

the training set for each method. This is defined as follows:

$$\text{err}(f, \mathcal{D}) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(f(\mathbf{x}_i) \neq y_i) \tag{1.14}$$

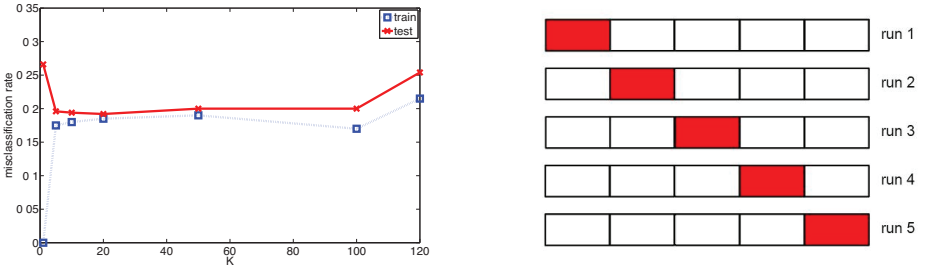
where $f(\mathbf{x})$ is our classifier. In Figure 1.21(a), we plot this error rate vs K for a KNN classifier (dotted blue line). We see that increasing K *increases* our error rate on the training set, because we are over-smoothing. As we said above, we can get minimal error on the training set by using $K = 1$, since this model is just memorizing the data.

However, what we care about is **generalization error**, which is the expected value of the misclassification rate when averaged over future data (see Section 6.3 for details). This can be approximated by computing the misclassification rate on a large independent *test set*, not used during model training. We plot the test error vs K in Figure 1.21(a) in solid red (upper curve). Now we see a **U-shaped curve**: for complex models (small K), the method overfits, and for simple models (big K), the method **underfits**. Therefore, an obvious way to pick K is to pick the value with the minimum error on the test set (in this example, any value between 10 and 100 should be fine).

Unfortunately, when training the model, we don't have access to the test set (by assumption), so we cannot use the test set to pick the model of the right complexity.[11] However, we can create a test set by partitioning the training set into two: the part used for training the model, and a second part, called the **validation set**, used for selecting the model complexity. We then fit all the models on the training set, and evaluate their performance on the validation set, and pick the best. Once we have picked the best, we can refit it to all the available data. If we have a separate test set, we can evaluate performance on this, in order to estimate the accuracy of our method. (We discuss this in more detail in Section 6.5.3.)

Often we use about 80% of the data for the training set, and 20% for the validation set. But if the number of training cases is small, this technique runs into problems, because the model

11. In academic settings, we usually do have access to the test set, but we should not use it for model fitting or model selection, otherwise we will get an unrealistically optimistic estimate of performance of our method. This is one of the "golden rules" of machine learning research.



(a) (b)

Figure 1.21 (a) Misclassification rate vs K in a K-nearest neighbor classifier. On the left, where K is small, the model is complex and hence we overfit. On the right, where K is large, the model is simple and we underfit. Dotted blue line: training set (size 200). Solid red line: test set (size 500). (b) Schematic of 5-fold cross validation. Figure generated by `knnClassifyDemo`.

won't have enough data to train on, and we won't have enough data to make a reliable estimate of the future performance.

A simple but popular solution to this is to use **cross validation** (**CV**). The idea is simple: we split the training data into K **folds**; then, for each fold $k \in \{1, \ldots, K\}$, we train on all the folds but the k'th, and test on the k'th, in a round-robin fashion, as sketched in Figure 1.21(b). We then compute the error averaged over all the folds, and use this as a proxy for the test error. (Note that each point gets predicted only once, although it will be used for training $K-1$ times.) It is common to use $K = 5$; this is called 5-fold CV. If we set $K = N$, then we get a method called **leave-one out cross validation**, or **LOOCV**, since in fold i, we train on all the data cases except for i, and then test on i. Exercise 1.3 asks you to compute the 5-fold CV estimate of the test error vs K, and to compare it to the empirical test error in Figure 1.21(a).

Choosing K for a KNN classifier is a special case of a more general problem known as **model selection**, where we have to choose between models with different degrees of flexibility. Cross-validation is widely used for solving such problems, although we will discuss other approaches later in the book.

1.4.9 No free lunch theorem

> All models are wrong, but some models are useful. — George Box (Box and Draper 1987, p424).[12]

Much of machine learning is concerned with devising different models, and different algorithms to fit them. We can use methods such as cross validation to empirically choose the best method for our particular problem. However, there is no universally best model — this is sometimes called the **no free lunch theorem** (Wolpert 1996). The reason for this is that a set of assumptions that works well in one domain may work poorly in another.

12. George Box is a retired statistics professor at the University of Wisconsin.

As a consequence of the no free lunch theorem, we need to develop many different types of models, to cover the wide variety of data that occurs in the real world. And for each model, there may be many different algorithms we can use to train the model, which make different speed-accuracy-complexity tradeoffs. It is this combination of data, models and algorithms that we will be studying in the subsequent chapters.

Exercises

Exercise 1.1 KNN classifier on shuffled MNIST data

Run `mnist1NNdemo` and verify that the misclassification rate (on the first 1000 test cases) of MNIST of a 1-NN classifier is 3.8%. (If you run it all on all 10,000 test cases, the error rate is 3.09%.) Modify the code so that you first randomly permute the features (columns of the training and test design matrices), as in `shuffledDigitsDemo`, and then apply the classifier. Verify that the error rate is not changed.

Exercise 1.2 Approximate KNN classifiers

Use the Matlab/C++ code at `http://people.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN` to perform approximate nearest neighbor search, and combine it with `mnist1NNdemo` to classify the MNIST data set. How much speedup do you get, and what is the drop (if any) in accuracy?

Exercise 1.3 CV for KNN

Use `knnClassifyDemo` to plot the CV estimate of the misclassification rate on the test set. Compare this to Figure 1.21(a). Discuss the similarities and differences to the test error rate.