# MACHINE LEARNING: A PROBABILISTIC PERSPECTIVE

**Kevin P. Murphy**
University of British Columbia, Canada
http://www.cs.ubc.ca/~murphyk
murphyk@cs.ubc.ca
murphyk@stat.ubc.ca

Draft of November 5, 2011

# Chapter 1

# Introduction

## 1.1 What is machine learning?

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

We are entering the era of **big data**. For example, as of 2008 there are about 1 trillion web pages[1], 20 hours of video are uploaded to YouTube every minute[2], Walmart handles more than 1M transactions per hour and has databases containing more than 2.5 petabytes ($2.5 \times 10^{15}$) of information [Cuk10], etc. This deluge of data calls for automated methods of data analysis, which is what **machine learning** provides. In particular, we can 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!).

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 predictions of $y$ for a given $x$ to the observed values in the training set). However, unsupervised learning is arguably much more interesting than supervised learning, since most human learning is unsupervised.

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. See e.g., [KLM96, SB98, RN10, Sze10b] for more information.

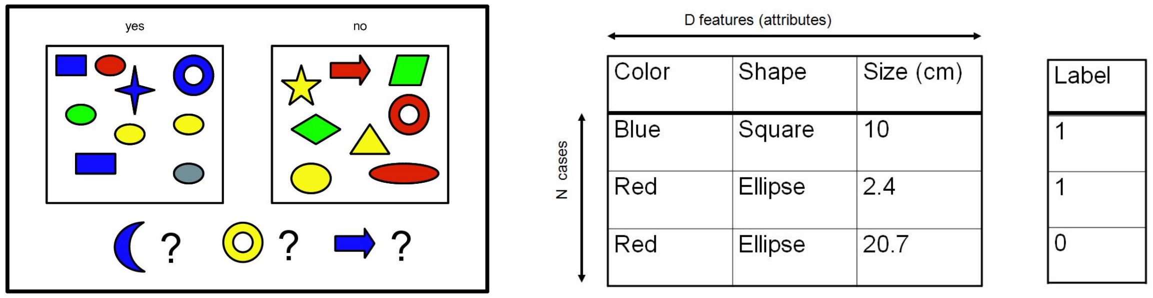
## 1.2 Supervised learning

We begin our investigation of machine learning by discussing supervised learning, which is arguably the most succesful form.

### 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

[1] http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html
[2] http://youtube-global.blogspot.com/2009/05/zoinks-20-hours-of-video-uploaded-every_20.html.



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

(a) (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.

(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 lookup 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 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 8.2.

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 8.2. 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

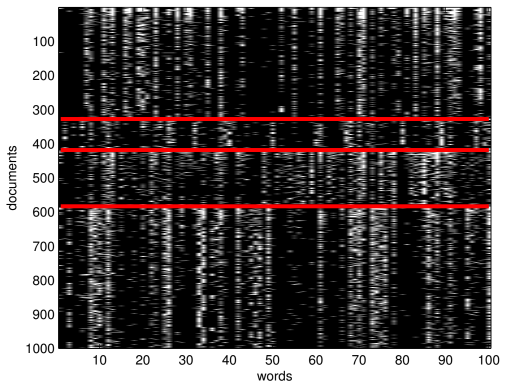


*Figure 1.2:* Subset of size 16242 x 100 of the 20-newsgroups data. We only show 1000 rows, for clarity. Each row is a document (represented as a bag-of-words bit vector), each column is a word. The red lines separate the 4 classes, which are (in descending order) comp, rec, sci, talk (these are the 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`.

lose money. In 2011, IBM unveiled a computer system called **Watson** which beat the top human Jeopary champion. Watson uses a variety of interesting techniques [FBCC+10], 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 [Met10]. 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.

#### 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 5.2.1, 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 7.11.0.7 and Exercise 7.11.0.8, you will get hands-on experience applying various classification techniques to the spam filtering problem.

##### 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 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.

##### 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

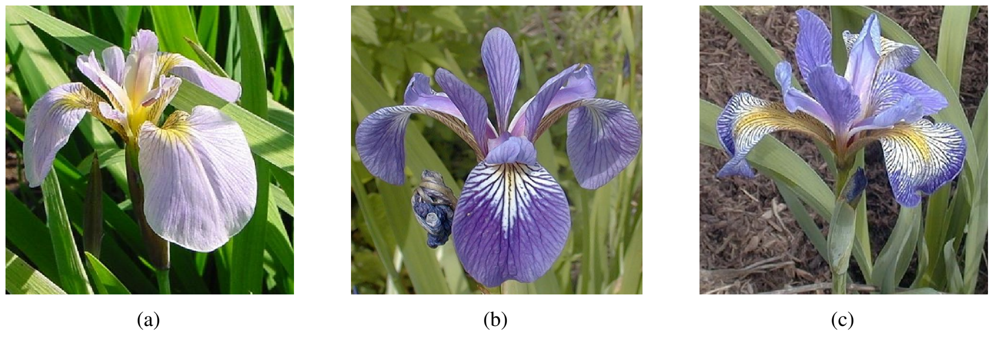


*Figure 1.3:* Three types of iris flowers: setosa, versicolor and virginica. Source: http://www.statlab.uni-heidelberg.de/data/iris/ . Used with kind permission of Dennis Kramb and SIGNA.

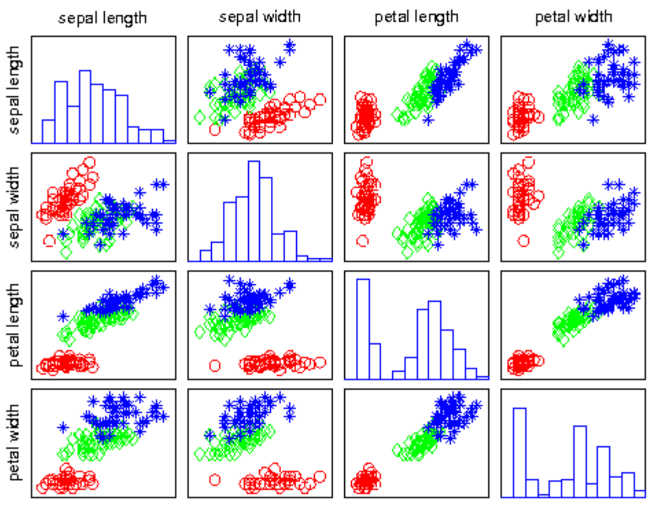


*Figure 1.4:* Visualization of the Iris data as a pairwise scatter plot. The diagonal plots the marginal histograms of the 4 features. The off diagonals contain scatterplots of all possible pairs of features. Red circle = setosa, green diamond = versicolor, blue star = virginica. Figure generated by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

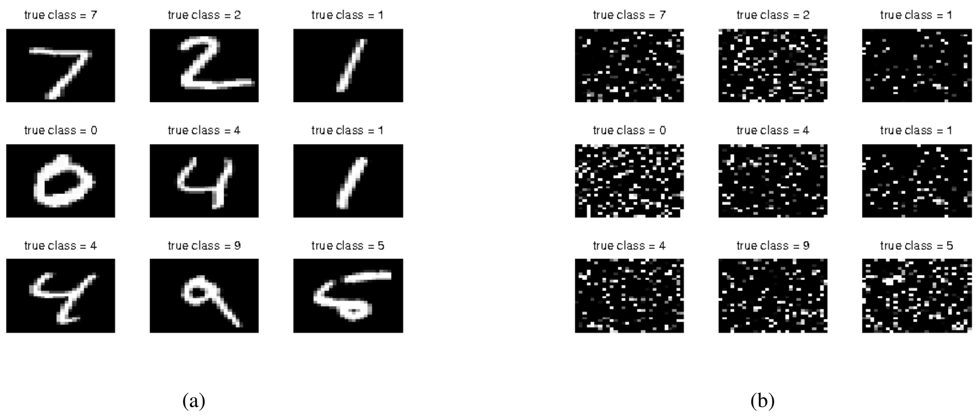


*Figure 1.5:* (a) First 9 test MNIST gray-scale images. (b) Same as (a), but with the features permuted randomly. Classification performance is identical on both versions of the data (assuming the training data is permuted in an identical way). Figure generated by `shuffledDigitsDemo`.

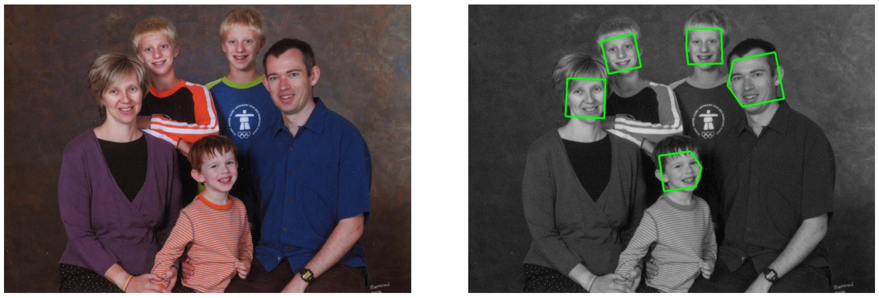


*Figure 1.6:* Example of face detection. (a) Input image (Murphy family, photo taken 5 August 2010 by Bernard Diedrich of Sherwood Studios). (b) Output of classifier, which detected 5 faces at different poses. This was produced using the online demo at `http://demo.pittpatt.com/`. The classifier was 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 (For an example of this technology applied to a video, see `http://www.youtube.com/watch?v=P3FcX2n398E`.)

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".[3] (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 $28 \times 28$ 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 1.5.0.2.) 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.

#### 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.

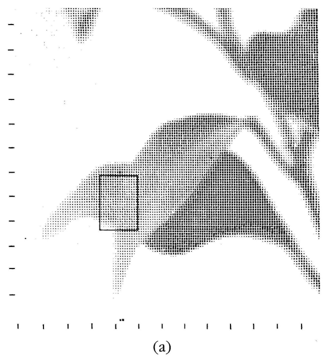
Having found the faces, one can then proceed to perform **face recognition**, which means estimating the identity of the person (see Figure 1.11(a)). In this case, the number of class labels might be very large. Also, the features one should use are likely to be different than in the face detection problem: 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., [Sze10a].

Although detecting and recognizing faces and other visual object categories might look like an easy problem, since people are so good at visual pattern recognition, bear in mind that what the machine sees is just an array of numbers. This is illustrated in Figure 1.7. On the left we see a picture of a flower, with several overlapping leaves, with a sub-image highlighted by a box. On the right we see a representation of the contents of the box, as seen by the computer. It is hard to tell from the numbers that this is a leaf, and detecting the edge between the two leaves is even harder. There is widespead concensus that what the brain is doing is inferring surfaces etc. from more global properties of the image, using a lot of prior knowledge derived from past experience [Mar82, DIPR07].

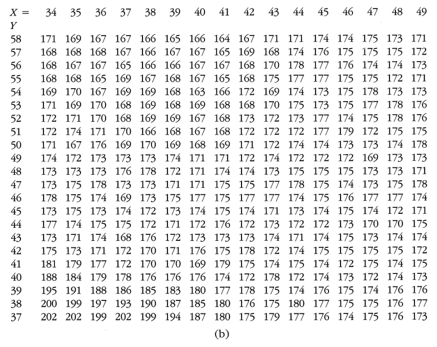
### 1.2.2 Regression

Regression is just like classification except the response variable is continuous. Figure 1.8 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

[3] Available from `http://yann.lecun.com/exdb/mnist/`.



(a)

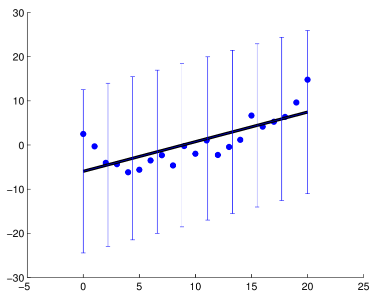


| X = | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Y | | | | | | | | | | | | | | | | |
| 58 | 171 | 169 | 167 | 167 | 166 | 165 | 166 | 164 | 167 | 171 | 171 | 174 | 174 | 175 | 173 | 171 |
| 57 | 168 | 168 | 168 | 167 | 166 | 167 | 167 | 165 | 169 | 168 | 174 | 176 | 175 | 175 | 175 | 172 |
| 56 | 168 | 167 | 167 | 165 | 166 | 166 | 167 | 167 | 168 | 170 | 178 | 177 | 176 | 174 | 174 | 173 |
| 55 | 168 | 168 | 165 | 169 | 167 | 168 | 167 | 165 | 168 | 175 | 177 | 177 | 175 | 175 | 172 | 171 |
| 54 | 169 | 170 | 167 | 169 | 169 | 168 | 163 | 166 | 172 | 169 | 174 | 173 | 175 | 178 | 173 | 173 |
| 53 | 171 | 169 | 170 | 168 | 169 | 168 | 169 | 168 | 168 | 170 | 175 | 173 | 175 | 177 | 178 | 176 |
| 52 | 172 | 171 | 170 | 168 | 169 | 169 | 167 | 168 | 173 | 172 | 173 | 177 | 174 | 175 | 178 | 176 |
| 51 | 172 | 174 | 171 | 170 | 166 | 168 | 167 | 168 | 172 | 172 | 172 | 177 | 179 | 172 | 175 | 175 |
| 50 | 171 | 167 | 176 | 169 | 170 | 169 | 168 | 169 | 171 | 172 | 174 | 174 | 173 | 173 | 174 | 178 |
| 49 | 174 | 172 | 173 | 173 | 173 | 174 | 171 | 171 | 172 | 174 | 172 | 172 | 172 | 169 | 173 | 173 |
| 48 | 173 | 173 | 173 | 176 | 178 | 172 | 171 | 174 | 174 | 173 | 175 | 175 | 175 | 173 | 173 | 171 |
| 47 | 173 | 175 | 178 | 173 | 173 | 171 | 171 | 175 | 175 | 177 | 178 | 175 | 174 | 173 | 175 | 178 |
| 46 | 178 | 175 | 174 | 169 | 173 | 175 | 177 | 175 | 177 | 177 | 174 | 175 | 176 | 177 | 177 | 174 |
| 45 | 173 | 175 | 173 | 174 | 172 | 173 | 174 | 175 | 174 | 171 | 173 | 174 | 175 | 174 | 172 | 171 |
| 44 | 177 | 174 | 175 | 175 | 172 | 171 | 172 | 176 | 172 | 173 | 172 | 172 | 173 | 170 | 170 | 175 |
| 43 | 173 | 171 | 174 | 168 | 176 | 172 | 173 | 173 | 173 | 174 | 171 | 174 | 175 | 173 | 174 | 174 |
| 42 | 175 | 173 | 171 | 172 | 170 | 171 | 176 | 175 | 178 | 172 | 174 | 175 | 175 | 175 | 175 | 172 |
| 41 | 181 | 179 | 177 | 172 | 170 | 170 | 169 | 179 | 175 | 174 | 175 | 174 | 172 | 175 | 174 | 175 |
| 40 | 188 | 184 | 179 | 178 | 176 | 176 | 176 | 174 | 172 | 178 | 172 | 174 | 173 | 172 | 174 | 173 |
| 39 | 195 | 191 | 188 | 186 | 185 | 183 | 180 | 177 | 178 | 175 | 174 | 176 | 175 | 174 | 176 | 176 |
| 38 | 200 | 199 | 197 | 193 | 190 | 187 | 185 | 180 | 176 | 175 | 180 | 177 | 175 | 175 | 176 | 177 |
| 37 | 202 | 202 | 199 | 202 | 199 | 194 | 187 | 180 | 175 | 179 | 177 | 176 | 174 | 175 | 176 | 173 |

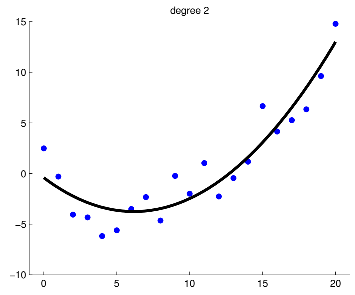
(b)

(b)

*Figure 1.7:* Left: an image of a flower. Right: what the computer sees. The array of numbers represents pixel intensities in the highlighted box. Source: [Mar82, p273] Used with kind permission of MIT Press.



(a)



(b)

*Figure 1.8:* (a) Linear regression on some 1d data. Figure generated by `linregDemo1`. (b) Polynomial regression on some 1d data. Figure generated by `linregPolyVsDegree`.

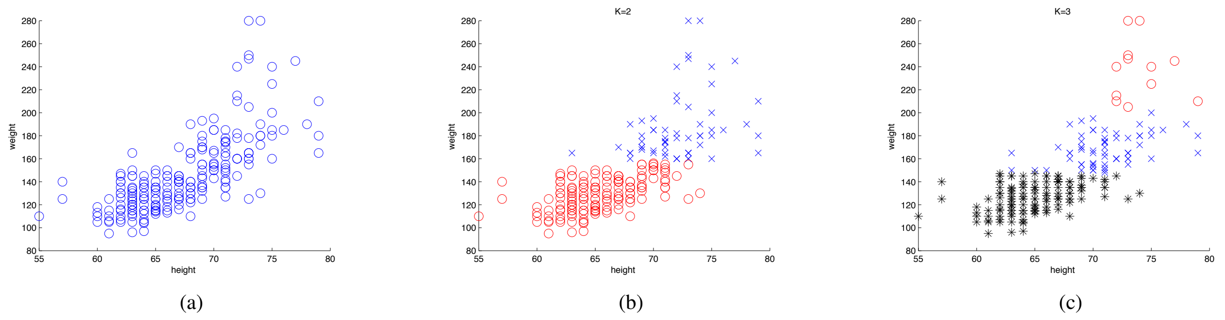


*Figure 1.9:* (a) The height and weight of some people. (b) A possible clustering using $K = 2$ clusters. (c) A possible clustering using $K = 3$ clusters. Figure generated by `kmeansHeightWeight`.

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.

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 (although see Chapter 17). This means that for most supervised learning problems, we can use univariate probability models (with input-dependent parameters), which significantly simplifies the problem.

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 manually label the data. Labeled data is not only expensive to acquire[4], but it also contains relatively little information, certainly not enough to reliably estimate the parameters of complex models. As Geoff Hinton has said,

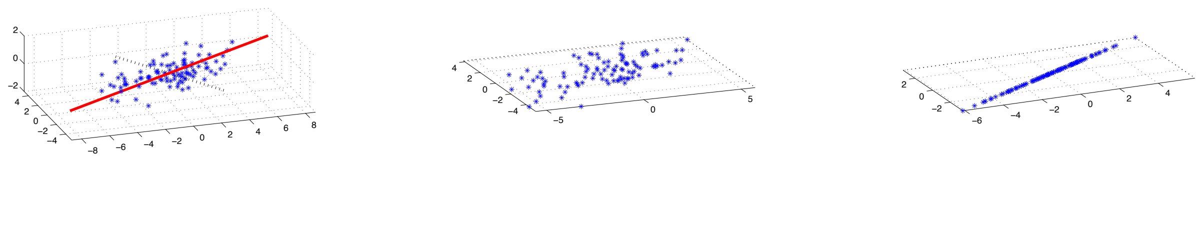
> When we're learning to see, nobody's telling us what the right answers are – we just look. Every so often, your mother says 'that's a dog,' but that's very little information. You'd be lucky if you got a few bits of information — even one bit per second — that way. The brain's visual system has $10^{14}$ 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 [Gor06]).

Below we describe some canonical examples of unsupervised learning.

### 1.3.1 Discovering clusters

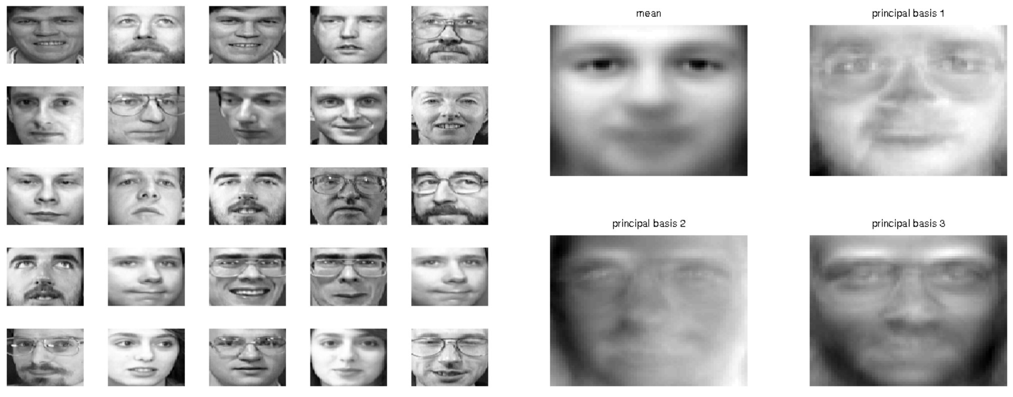
As a canonical example of unsupervised learning, consider the problem of **clustering** data into groups. For example, Figure 1.9(a) plots some 2d data, representing the height and weight of a group of 210 people. It seems that there might be various

[4] 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.



(a) (b) (c)

*Figure 1.10:* (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. (c) 1D representation of the data. Figure generated by `pcaDemo3d`.



(a) (b)

*Figure 1.11:* a) 25 randomly chosen $64 \times 64$ pixel images from the Olivetti face database. (b) The mean and the first three principal component basis vectors (eigenfaces). Figure generated by `pcaImageDemo`.

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 **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 1.9(b-c), where we use different colors to indicate the assignments.

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 [CKS+88] 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., [Ber06]).
- In biology, it is common to cluster flow-cytometry data into groups, to discover different sub-populations of cells (see e.g., [LHB+09]).

### 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

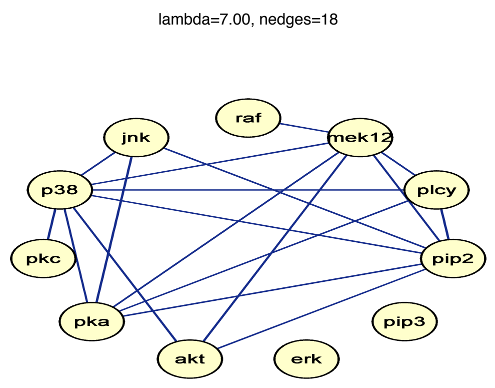


*Figure 1.12:* A sparse undirected Gaussian graphical model learned using graphical lasso (Section 25.7.2) applied to some flow cytometry data (from [SPP$^+$05]), which measures the phosphorylation status of 11 proteins. Figure generated by `ggmLassoDemo`.

shown in Figure 1.10, where we project some 3d data down to a 2d plane, and then down to a 1d line. The 2d approximation is quite good, since there is little variation off this plane, but the 1d approximation is a very poor approximation to the original data.

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 1.11.

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 10.1 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 24.2.2).
- 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 10.6).
- In computer graphics, it is common to project motion capture data to a low dimensional space, and use it to create animations. See Section 13.5 for one way to tackle such problems.

### 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 6, 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 an 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 [SPP$^+$05]. Figure 1.12 gives an example of a graph structure that was learned from this data using methods discussed in Section 25.7.2). As another example, [SaTSHJ06] 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.

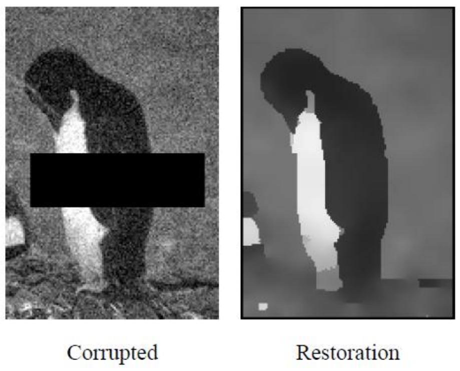


*Figure 1.13:* (a) A noisy image with an occluder. (b) An estimate of the underlying pixel intensities, based on a pairwise MRF model. Source: Figure 8 of [FH06]. Used with kind permission of Pedro Felzenszwalb.

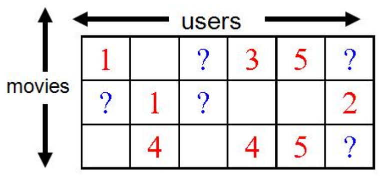


*Figure 1.14:* Netflix movie-rating data. The matrix consists of 17,770 movies and 480,189 users. Only 100,480,507 entries (on the scale 1:5) are known. A subset of these, denoted by ?, are known, but must be predicted during the competition; the rest are available as training data. Empty cells are unknown.

- One application of (Gaussian) graphical models is in financial portfolio management, where accurate models of the covariance between large numbers of different stocks is important. [CW07] 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.

## 1.3.4 Matrix completion

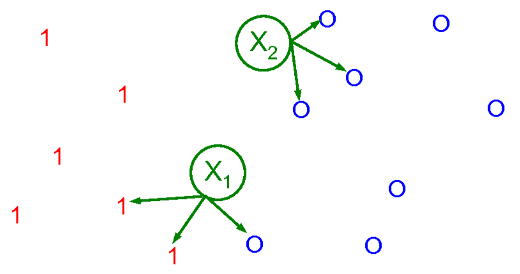
Sometimes we have missing data, that is, variables whose values are unknown. For example, we might have conducted a survey, and some people might not have answered certain questions. Or we might have various sensors, some of which fail. The corresponding design matrix will then have "holes" in it; these 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 1.13, 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 17.2.3 and 11.8.4 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 (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. See Figure 1.14 for an example.



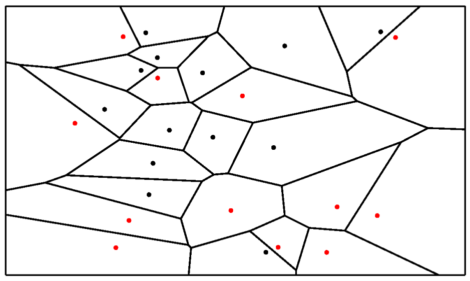


*Figure 1.15:* (a) Illustration of a $K$-nearest neighbors classifier in 2d for $K = 3$. The 3 nearest neighbors of test point $\mathbf{x}_1$ have labels 1, 1 and 0, so we predict $p(y = 1|\mathbf{x}_1, \mathcal{D}, K = 3) = 2/3$. The 3 nearest neighbors of test point $x_2$ have labels 0, 0, and 0, so 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 [DHS01]. Figure generated by `knnVoronoi`.

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/`). On 21 September 2009, the prize was awarded to a team of researchers known as "BellKor's Pragmatic Chaos". Section 24.6.2 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 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., [HvdMC$^+$10]).

It is common to solve such tasks using **frequent itemset mining**, which create association rules (see e.g., [HTF09, 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., [HvdMC$^+$10]. 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.

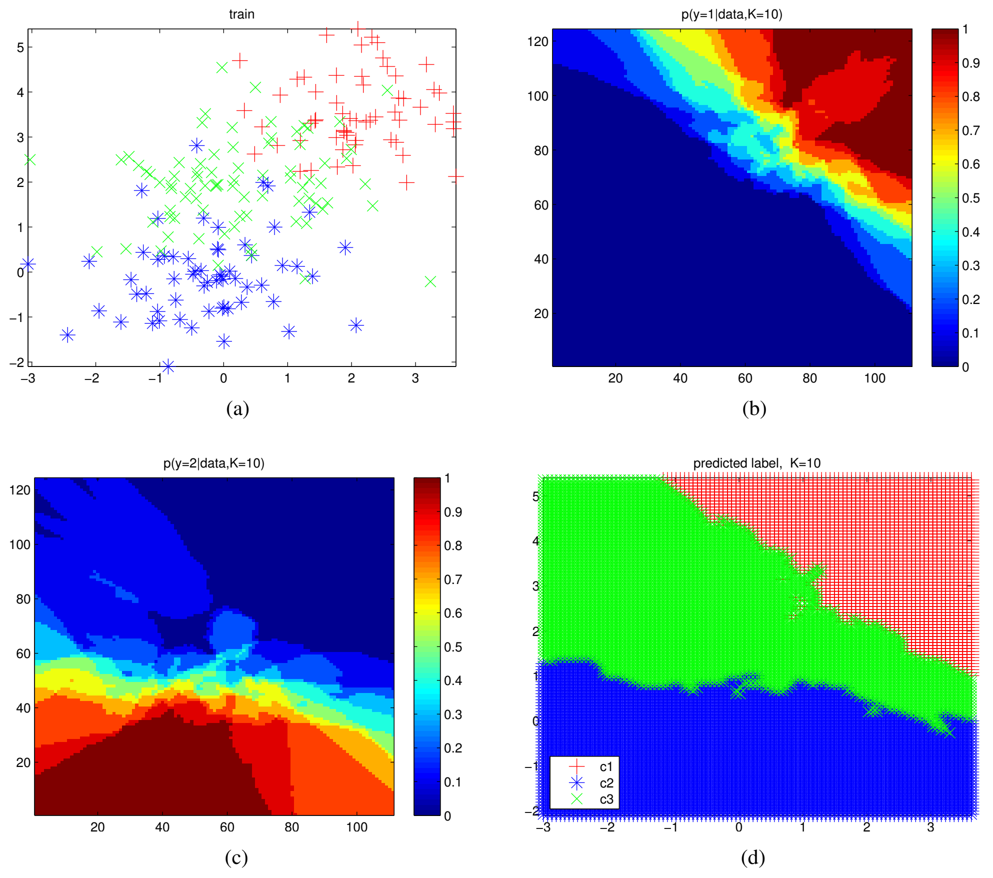


*Figure 1.16:* (a) Some synthetic 3-class training data in 2d. (b) Probability of class 1 for KNN with $K = 10$. (c) Probability of class 2. (d) MAP estimate of class label. Figure generated by `knnClassifyDemo`.

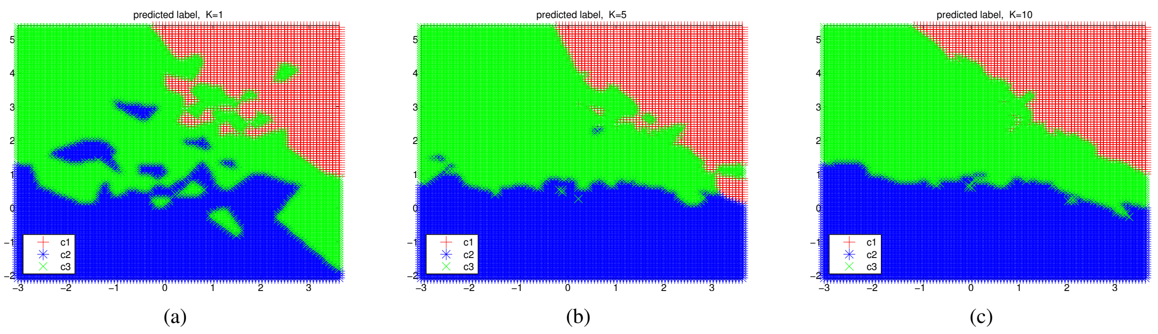


*Figure 1.17:* Prediction surface for KNN on the data in Figure 1.16(a). (a) K=1. (b) K=5. (c) K=10. Figure generated by `knnClassifyDemo`.

### 1.4.2 A simple non-parametric model for classification: the $K$-nearest neighbor classifier

Another example of a simple 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 $\mathbf{x}$, counts how many members of each class are in this set, and returns that empirical fraction as the estimate, as illustrated in Figure 1.15. 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 12.7.3. The most common distance metric to use is Euclidean distance (which limits the applicability of the technique to data which is real-valued).

Figure 1.16 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 $\hat{y}(\mathbf{x}) = \text{argmax}_c(y = c|\mathbf{x}, \mathcal{D})$.

### 1.4.3 Overfitting and model selection

If the distance function is fixed, then the only free parameter in the KNN model is the value $K$. This can have a large effect on the predictions, as we now discuss. In particular, a KNN classifier with $K = 1$ induces a **Voronoi tessellation** of the points (see Figure 1.15(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. This is illustrated in Figure 1.17(a).

When $K = 1$, the method clearly makes no errors on the training set, but the result prediction surface is very "wiggly". Therefore the method may not work well at predicting future data. A function that fits the training data well but that is overly complex, and that does not generalize well to novel test data, is said to have **overfit** the data. Avoiding overfitting is one of the most important issues in machine learning.

In Figure 1.17(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.

How, then, should we pick $K$? A natural approach is to compute the **misclassification rate** on the training set. This is defined as follows:

$$\text{err}(f, \mathcal{D}) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(f(\mathbf{x}_i) \neq y_i) \tag{1.4}$$

where $f(\mathbf{x}; K)$ is our classifier. In Figure 1.18(a), we plot this error rate vs $K$ (in blue). 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 8.3 for details). This can be approximated by computing the misclassification rate on a

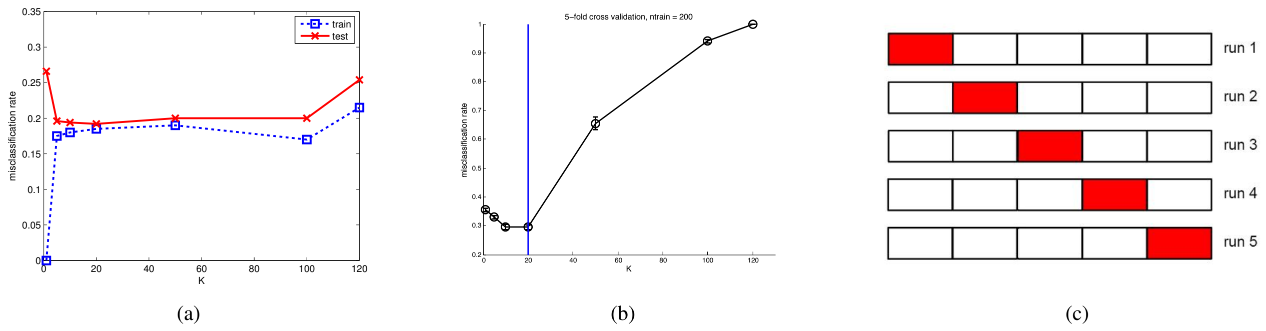


*Figure 1.18:* (a) Misclassification rate vs $K$ in a K-nearest neighbor classifier. Complexity of the model decreases as we move to the right. Dotted blue line: training set (size 200). Solid red line: test set (size 500). (b) 5-fold cross-validation estimate of the generalization error. Error bars indicate standard error of the mean. The vertical blue indicates the chosen model. (c) Schematic of 5-fold cross validation. Figure generated by `knnClassifyDemo`.

large independent *test set*, not used during model training. We plot the test error vs $K$ in Figure 1.18(a) in red. 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.[5] 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, and use the resulting model on future test data. (If we want to estimate the performance of this resulting best model on future data, we need a third, separate test set that was not used at all for model selection or training.)

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 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 1.18(c). 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 $i$.

In Figure 1.18(b) we show the results of 5-fold CV for our KNN example. We see that this is a reasonable proxy for the generalization error, in the sense that it has the same general U-shape, and the minimum (indicated by the vertical blue line) is in roughly the same location. The meaning of the error bars is explained in Section 8.3.8.2.

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 later we will discuss a more probabilistic approach to this problem.

## 1.4.4 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 possible performance if $N \to \infty$ [CH67].

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 [HTF09, 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% 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 1.19. Since the entire range of the data is only 1 along each dimension, we see that the method is no longer very

[5] 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.

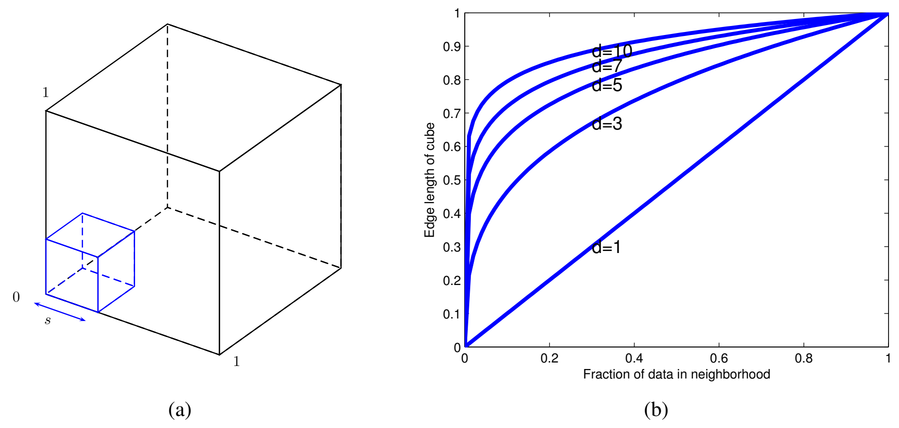


*Figure 1.19:* Illustration of the curse of dimensionality. (a) We embed a small cube of side $s$ inside a larger unit cube. (b) We plot the edge length of a cube needed to cover a given volume of the unit cube as a function of the number of dimensions. Based on Figure 2.6 from [HTF09]. Figure generated by `curseDimensionality`.

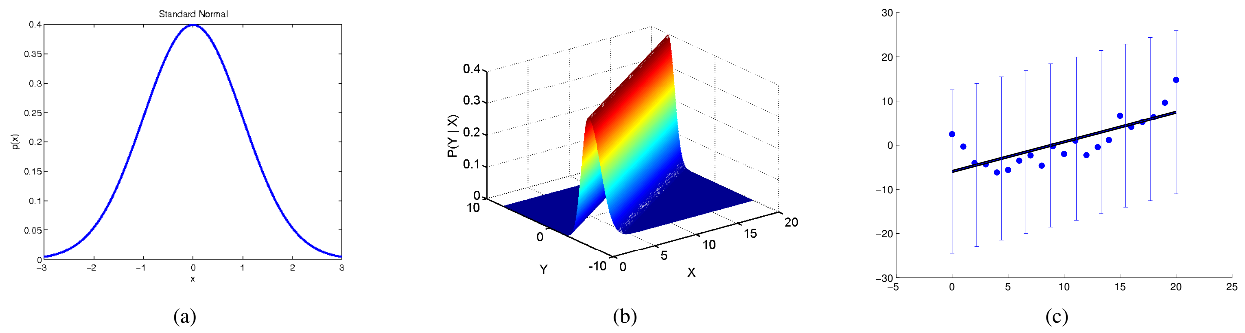


*Figure 1.20:* (a) A Gaussian pdf with mean 0 and variance 1. Figure generated by `gaussPlotDemo`. (b) Visualization of the conditional density model $p(y|x, \boldsymbol{\theta}) = \mathcal{N}(y|w_0 + w_1 x, \sigma^2)$. The density falls off exponentially fast as we move away from the regression line. Figure generated by `linregWedgeDemo2`. (c) We plot the mean and 95% credible interval of $p(y|x, \boldsymbol{\theta}) = \mathcal{N}(y|w_0 + w_1 x, \sigma^2)$ for a linear regression model. Figure generated by `linregDemo1`.

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.

## 1.4.5 Simple parametric models for classification and regression

The main way to combat the curse of dimensionality is to make assumptions about the nature of the input-output mapping. This is usually done by using a **parametric model** to represent the mapping. That is, we use a model of the form $p(y|\mathbf{x}) = f(y, \mathbf{x}, \boldsymbol{\theta})$, where $f$ is some kind of function, and $\boldsymbol{\theta}$ are the parameters of this function. We give some examples below.

### 1.4.5.1 Linear regression

The simplest parametric model for regression is known as **linear regression**. This is a model of the form

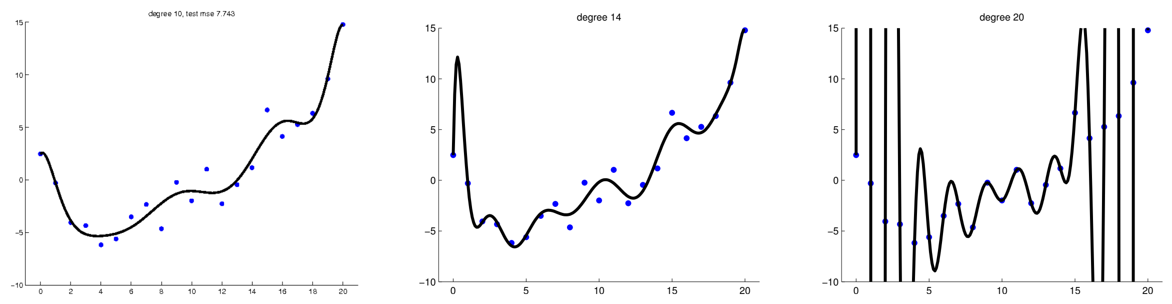
$$y = \mathbf{w}^T \mathbf{x} + \epsilon \tag{1.5}$$

where $\mathbf{w}^T \mathbf{x} = \sum_{j=1}^D w_j x_j$ is a linear combination of the inputs, and $\epsilon$ represents the deviation away from a straight line, known as the **residual error** or **noise**. We often add an offset or bias term $w_0$ to the mean, which changes the overall level of the response. To simplify notation we often add a dummy 1 term to $\mathbf{x}$, so that $\mathbf{w}^T \mathbf{x} = w_0 + \sum_{j=1}^D w_j x_j$.

We typically assume the noise term $\epsilon$ has a Gaussian[6] distribution with mean 0 and variance $\sigma^2$, which we write as $\epsilon \sim \mathcal{N}(0, \sigma^2)$. The corresponding **probability density function** or **pdf** is given by

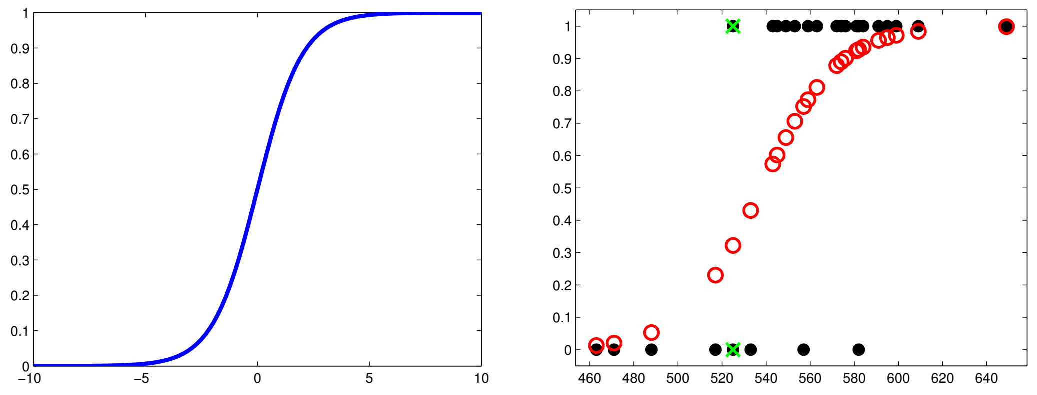
$$\mathcal{N}(y|\mu, \sigma^2) \quad := \quad \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(y-\mu)^2} \tag{1.6}$$

[6] Carl Friedrich Gauss (1777–1855) was a German mathematician and physicist.



(a) (b) (c)

*Figure 1.21:* Polynomial of degrees 10, 14, 20 fit by least squares to 21 data points. Figure generated by `linregPolyVsDegree`.



(a) (b)

*Figure 1.22:* (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`.

When we plot this distribution, we get the well-known **bell curve** shown in Figure 1.20(a).

Another way to write the linear regression model is as follows:

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

This makes it clear that the mean is a linear function of the inputs, but the variance is fixed. This is illustrated in 1d in Figure 1.20(b-c).

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

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|\mathbf{w}^T\boldsymbol{\phi}(\mathbf{x}), \sigma^2) \tag{1.8}$$

This is known as **basis function expansion**. A simple example are polynomial basis functions, where the model has the form

$$\boldsymbol{\phi}(x) = [1, x, x^2, \ldots, x^d] \tag{1.9}$$

Figure 1.21 illustrates the effect of changing $d$: increasing the degree $d$ allows us to create increasingly complex functions.

However, we need to be careful to avoid overfitting. For example, consider what happens if we only have $N = 21$ data points, and we fit a degree $d = 20$ polynomial: such a model can perfectly **interpolate** the data (just as a straight line can perfectly fit any two points), but the resulting function is very "wiggly", as shown in Figure 1.21. One way to pick a model of the "right" complexity is to use cross-validation. We will discuss other methods below.

### 1.4.5.2 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[7], which is more appropriate for the case when $y \in \{0, 1\}$. That is, we use

$$p(y|\mathbf{x}, \mathbf{w}) = \text{Ber}(y|\mu(\mathbf{x})) \tag{1.10}$$

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

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 $0 \le \mu(\mathbf{x}) \le 1$ by defining

$$\mu(\mathbf{x}) = \text{sigm}(\mathbf{w}^T\mathbf{x}) \tag{1.11}$$

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

$$\text{sigm}(\eta) := \frac{1}{1+\exp(-\eta)} = \frac{e^\eta}{e^\eta + 1} \tag{1.12}$$

The term "sigmoid" means S-shaped: see Figure 1.22(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

$$p(y|\mathbf{x}, \mathbf{w}) = \text{Ber}(y|\text{sigm}(\mathbf{w}^T\mathbf{x})) \tag{1.13}$$

This is called **logistic regression** due to its similarity to linear regression, although it is actually a form of classification. Logistic regression is very widely used. For example, it forms the basis of Google's SmartASS ad system mentioned earlier. It also forms the basis of several more sophisticated models, such as neural networks (Section 14.5), relevance vector machines (Section 12.3.2), etc.

A simple example of logistic regression is shown in Figure 1.22(b), where we plot

$$p(y_i = 1|x_i, \mathbf{w}) = \text{sigm}(w_0 + w_1 x_i) \tag{1.14}$$

where $x_i$ is the SAT[8] 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 7.4.4).

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

$$\hat{y}(x) = 1 \iff p(y=1|\mathbf{x}) > 0.5 \tag{1.15}$$

By looking at Figure 1.22(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.

In Figure 1.22, the input data was one-dimensional. However, logistic regression can easily be extended to higher-dimensional inputs. For example, Figure 1.23 shows plots of $p(y=1|\mathbf{x}, \mathbf{w}) = \text{sigm}(\mathbf{w}^T\mathbf{x})$ for 2d input and different weight vectors $\mathbf{w}$. If we threshold these probabilities at 0.5, we induce a linear decision boundary, whose normal (perpendicular) is given by $\mathbf{w}$.

### 1.4.6 Parameter estimation

Given a parametric model, such as linear or logistic regression, how should we estimate the parameters, $\boldsymbol{\theta}$? A natural approach is to compute a MAP estimate of the parameters, given by

$$\hat{\boldsymbol{\theta}} = \text{argmax}\, p(\boldsymbol{\theta}|\mathcal{D}) \tag{1.16}$$

This captures the intuitive notion that we should use the parameter values that are most probable given the data. The function that we are maximizing, $p(\boldsymbol{\theta}|\mathcal{D})$, is known as the **posterior distribution** of the parameters, and reflects what we know about them after having seen the data. We can compute this using Bayes rule (Section 2.2.2.4), which tells us that

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathcal{D})} \tag{1.17}$$

Here $p(\boldsymbol{\theta})$ is the **prior distribution** for $\boldsymbol{\theta}$, and encodes what we know before we saw the data. The term $p(\mathcal{D}|\boldsymbol{\theta})$ is the **likelihood of the data**, and measures how well $\boldsymbol{\theta}$ predicts the observed data. Finally, $p(\mathcal{D})$ is a normalization constant given by

$$p(\mathcal{D}) = \int p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta})d\boldsymbol{\theta} \tag{1.18}$$

[8] 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 [JA99, p87]).

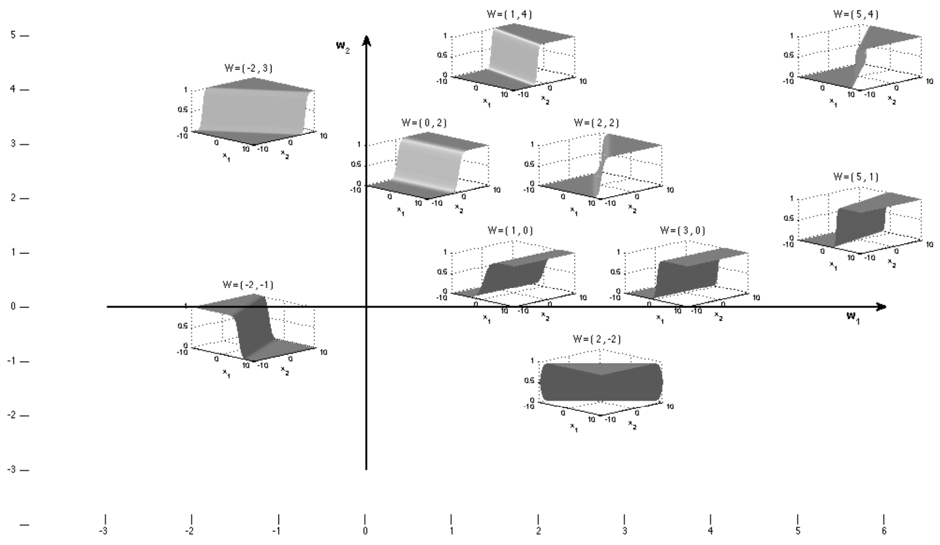


*Figure 1.23:* Plots of $\text{sigm}(w_1 x_1 + w_2 x_2)$. Here $\mathbf{w} = (w_1, w_2)$ defines the normal to the decision boundary. Points to the right of this have $\text{sigm}(\mathbf{w}^T\mathbf{x}) > 0.5$, and points to the left have $\text{sigm}(\mathbf{w}^T\mathbf{x}) < 0.5$. Based on Figure 39.3 of [Mac03]. Figure generated by `sigmoidplot2D`.

We can ignore this constant if we are only interested in estimating $\boldsymbol{\theta}$. In other words, we can rewrite the MAP estimate as follows:

$$\hat{\boldsymbol{\theta}} = \text{argmax}\, p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta}) \tag{1.19}$$

It is conventional to maximize the log of the posterior, rather than the posterior; this converts the product into a sum, but does not affect the location of the maximum, since log is a monotonic function. Thus we can write

$$\hat{\boldsymbol{\theta}} = \text{argmax} \log p(\mathcal{D}|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) \tag{1.20}$$

All that remains is to specify the form of $p(\mathcal{D}|\boldsymbol{\theta})$ and $p(\boldsymbol{\theta})$, and to devise a suitable optimization algorithm. We will discuss these issues at length later in the book. However, we give one some simple example below, in order to introduce to some key concepts.

#### 1.4.6.1 Maximum likelihood estimation and least squares

If we do not know anything about what parameter values are good ones to use, we can use a **uniform prior**, $p(\boldsymbol{\theta}) \propto 1$, which says that all values are equally likely. (This will be justified in Section 2.8.2.) In this case, the MAP estimate becomes equivalent to the **maximum likelihood estimate** or **MLE**, which is defined as

$$\hat{\boldsymbol{\theta}} := \arg\max_{\boldsymbol{\theta}} \log p(\mathcal{D}|\boldsymbol{\theta}) \tag{1.21}$$

It is common to assume the training examples are independent and identically distributed, commonly abbreviated to **iid**. This means we can write the log-likelihood as follows:

$$\ell(\boldsymbol{\theta}) := \log p(\mathcal{D}|\boldsymbol{\theta}) = \sum_{i=1}^{N} \log p(y_i|\mathbf{x}_i, \boldsymbol{\theta}) \tag{1.22}$$

Instead of maximizing the log-likelihood, we can equivalently minimize the **negative log likelihood** or **NLL**:

$$\text{NLL}(\boldsymbol{\theta}) := -\sum_{i=1}^{N} \log p(y_i|\mathbf{x}_i, \boldsymbol{\theta}) \tag{1.23}$$

The NLL formulation is sometimes more convenient, since many optimization software packages are designed to find the minima of functions, rather than maxima.

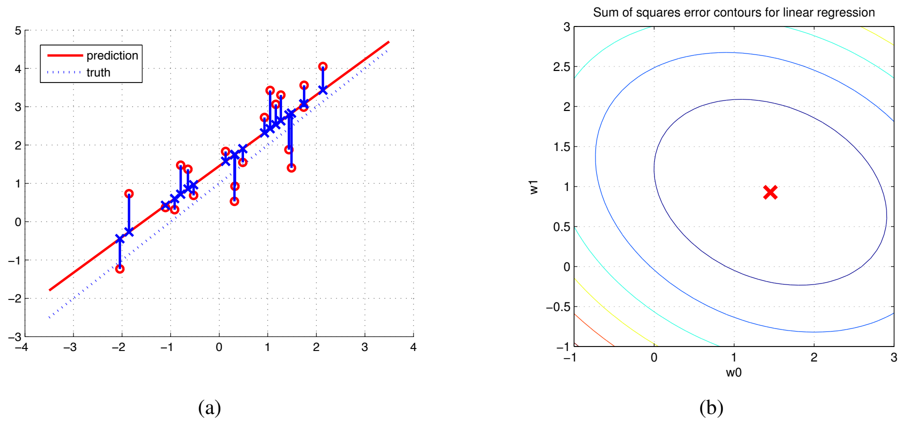


*Figure 1.24:* (a) In linear least squares, we try to minimize the sum of squared distances from each training point (denoted by a red circle) to its approximation (denoted by a blue cross), that is, we minimize the sum of the lengths of the little vertical blue lines. The red diagonal line represents $\hat{y}(x) = w_0 + w_1 x$, which is the least squares regression line. Note that these residual lines are not perpendicular to the least squares line, in contrast to Figure 10.5. Figure generated by `residualsDemo`. (b) Contours of the RSS error surface for the same example. The red cross represents the MLE, $\mathbf{w} = (1.45, 0.93)$. Figure generated by `contoursSSEdemo`.

Now let us apply the method of MLE to the linear regression setting. Inserting the definition of the Gaussian into the above, we find that the log likelihood is given by

$$\ell(\boldsymbol{\theta}) = \sum_{i=1}^{N} \log \left[ \left( \frac{1}{2\pi\sigma^2} \right)^{\frac{1}{2}} \exp \left( -\frac{1}{2\sigma^2} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 \right) \right] \tag{1.24}$$

$$= \frac{-1}{2\sigma^2} RSS(\mathbf{w}) - \frac{N}{2} \log(2\pi\sigma^2) \tag{1.25}$$

RSS stands for **residual sum of squares** and is defined by

$$\text{RSS}(\mathbf{w}) := \sum_{i=1}^{N} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 \tag{1.26}$$

The RSS is also called the **sum of squared errors**, or SSE, and SSE$/N$ is called the **mean squared error** or **MSE**. It can also be written as the square of the $\ell_2$ **norm** of the vector of residual errors:

$$\text{RSS}(\mathbf{w}) = ||\boldsymbol{\epsilon}||_2^2 = \sum_{i=1}^{N} \epsilon_i^2 \tag{1.27}$$

where $\epsilon_i = (y_i - \mathbf{w}^T \mathbf{x}_i)$.

We see that the MLE for $\mathbf{w}$ is the one that minimizes the RSS, so this method is known as **least squares**. This method is illustrated in Figure 1.24(a). The training data $(x_i, y_i)$ are shown as red circles, the estimated values $(x_i, \hat{y}_i)$ are shown as blue crosses, and the residuals $\epsilon_i = y_i - \hat{y}_i$ are shown as vertical blue lines. The goal is to find the setting of the parameters (the slope $w_1$ and intercept $w_0$) such that the resulting red line minimizes the sum of squared residuals (the lengths of the vertical blue lines).

In Figure 1.24(b), we plot the NLL surface for our linear regression example. We see that it is a quadratic "bowl" with a unique minimum. It turns out that there is a simple closed form solution for this MLE, which we derive in Section 7.2.1. Importantly, this is true even if we use basis function expansion, such as polynomials, because the NLL is still *linear in the parameters* $\mathbf{w}$, even if it is not linear in the inputs $\mathbf{x}$.

### 1.4.6.2 Regularization

One problem with ML estimation is that it can result in overfitting. The reason is that it is picking the parameter values that are the "best" for modeling the training data; buti if the data is noisy, such parameters often result in complex functions. As a simple example, suppose we fit a degree 14 polynomial to $N = 21$ data points. The resulting curve is very "wiggly", as shown in Figure 1.25(a). The corresponding least squares coefficients (excluding $w_0$) are as follows:

```
6.560, -36.934, -109.255, 543.452, 1022.561, -3046.224, -3768.013,
8524.540, 6607.897, -12640.058, -5530.188, 9479.730, 1774.639, -2821.526
```

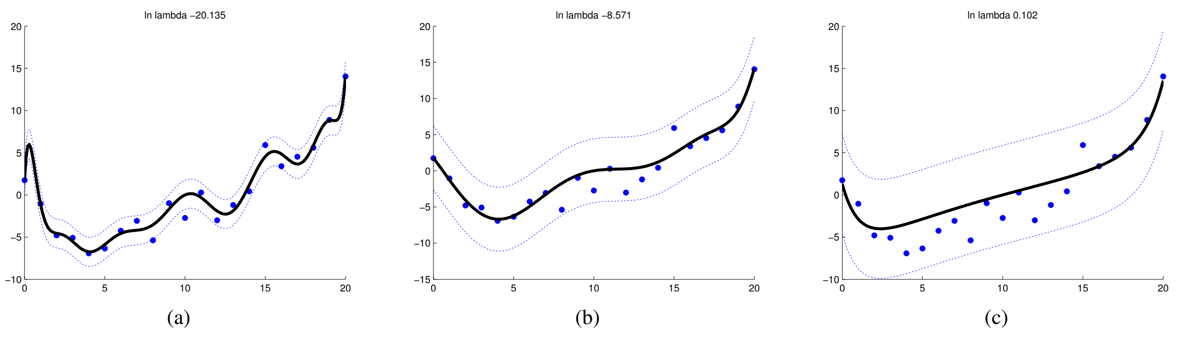


*Figure 1.25:* Degree 14 Polynomial fit to $N = 21$ data points with increasing amounts of $\ell_2$ regularization. Data was generated from noise with variance $\sigma^2 = 4$. The error bars, representing the noise variance $\sigma^2$, get wider since we are ascribing more of the data variation to the noise. Figure generated by `linregPolyVsRegDemo`.

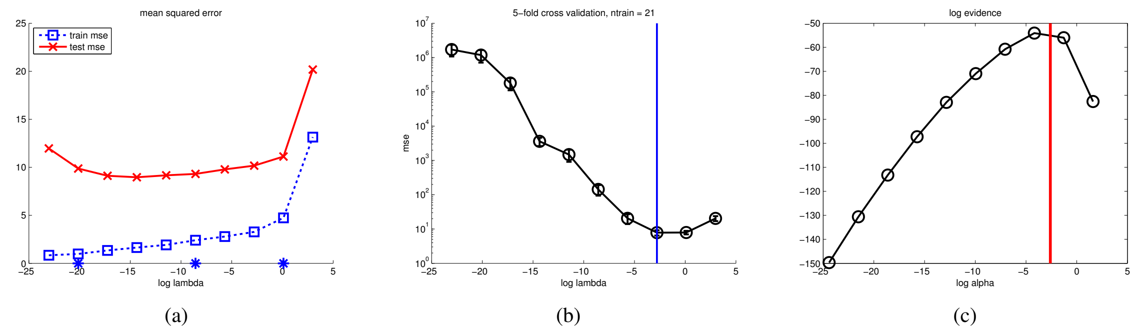


*Figure 1.26:* (a) Training and test set error for a degree 14 polynomial fit by ridge regression, plotted vs $\log(\lambda)$. Data was generated from noise with variance $\sigma^2 = 4$. Note: Models are ordered from complex (small regularizer) on the left to simple (large regularizer) on the right. The stars correspond to the values used to plot the functions in Figure 1.25. (b) Estimate of test MSE produced by 5-fold cross-validation (see Section 1.4.3). The lowest CV error is indicated by the vertical line. Note the vertical scale is in log units. (c) Log marginal likelihood vs $\log(\lambda)$. The largest value is indicated by the vertical line. Figure generated by `linregPolyVsRegDemo`.

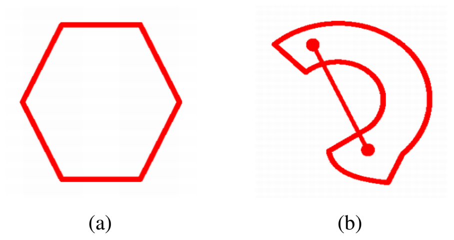


*Figure 1.27:* (a) Illustration of a convex set. (b) Illustration of a nonconvex set.

We see that there are many large positive and negative numbers. These balance out exactly to make the curve “wiggle” in just the right way so that it almost perfectly interpolates the data. But this situation is unstable: if we changed the data a little, the coefficients would change a lot.

We can encourage the parameters to be small, thus resulting in a smoother curve, by using a zero-mean Gaussian prior:

$$p(\mathbf{w}) = \prod_j \mathcal{N}(w_j|0, \tau^2) \tag{1.28}$$

where $1/\tau^2$ controls the strength of the prior. The corresponding MAP estimation problem becomes

$$\operatorname*{argmax}_{\mathbf{w}} \sum_{i=1}^N \log \mathcal{N}(y_i|w_0 + \mathbf{w}^T\mathbf{x}_i, \sigma^2) + \sum_{j=1}^D \log \mathcal{N}(w_j|0, \tau^2) \tag{1.29}$$

It is a simple exercise to show that this is equivalent to minimizing the following:

$$J(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^N (y_i - (w_0 + \mathbf{w}^T\mathbf{x}_i))^2 + \lambda||\mathbf{w}||_2^2 \tag{1.30}$$

where $\lambda := \sigma^2/\tau^2$. Here the first term is the MSE/ NLL as usual, and the second term, $\lambda \geq 0$, is a complexity penalty. This technique is known as **ridge regression**, or **penalized least squares**. In general, adding a Gaussian prior to the parameters of a model to encourage them to be small is called $\ell_2$ **regularization** or **weight decay**. Note that the offset term $w_0$ is not regularized, since this just affects the height of the function, not its complexity. By penalizing the sum of the magnitudes of the weights, we ensure the function is simple (since $\mathbf{w} = \mathbf{0}$ corresponds to a straight line, which is the simplest possible function, corresponding to a constant.)

We illustrate this idea in Figure 1.25, where we see that increasing $\lambda$ results in smoother functions. The resulting coefficients also become smaller. For example, using $\lambda = 10^{-3}$, we have

```
2.128, 0.807, 16.457, 3.704, -24.948, -10.472, -2.625, 4.360, 13.711,
10.063, 8.716, 3.966, -9.349, -9.232
```

In Figure 1.26(a), we plot the MSE on the training and test sets vs $\log(\lambda)$. We see that, as we increase $\lambda$ (so the model becomes more constrained), the error on the training set increases. For the test set, we see the characteristic U-shaped curve, where the model overfits and then underfits. It is common to use cross validation to pick $\lambda$, as shown in Figure 1.26(b). In Section 1.4.8, we will discuss a more probabilistic approach.

We will consider a variety of different priors in this book. Each of these corresponds to a different form of **regularization**. This technique is very widely used to prevent overfitting.

#### 1.4.6.3 Convexity

When discussing least squares, we noted that the NLL had a bowl shape with a unique minimum. The technical term for functions like this is **convex**. Convex functions play a very important role in machine learning.

Let us define this concept more precisely. We say a *set* $\mathcal{S}$ is **convex** if for any $\boldsymbol{\theta}, \boldsymbol{\theta}' \in \mathcal{S}$, we have

$$\lambda\boldsymbol{\theta} + (1-\lambda)\boldsymbol{\theta}' \in \mathcal{S}, \ \ \forall\, \lambda \in [0,1] \tag{1.31}$$

That is, if we draw a line from $\boldsymbol{\theta}$ to $\boldsymbol{\theta}'$, all points on the line lie inside the set. See Figure 1.27(a) for an illustration of a convex set, and Figure 1.27(b) for an illustration of a non-convex set.

A *function* $f(\boldsymbol{\theta})$ is called convex if its **epigraph** (the set of points above the function) defines a convex set. Equivalently, a function $f(\boldsymbol{\theta})$ is called convex if it is defined on a convex set and if, for any $\boldsymbol{\theta}, \boldsymbol{\theta}' \in \mathcal{S}$, and for any $0 \leq \lambda \leq 1$, we have

$$f(\lambda\boldsymbol{\theta} + (1-\lambda)\boldsymbol{\theta}') \leq \lambda f(\boldsymbol{\theta}) + (1-\lambda)f(\boldsymbol{\theta}') \tag{1.32}$$

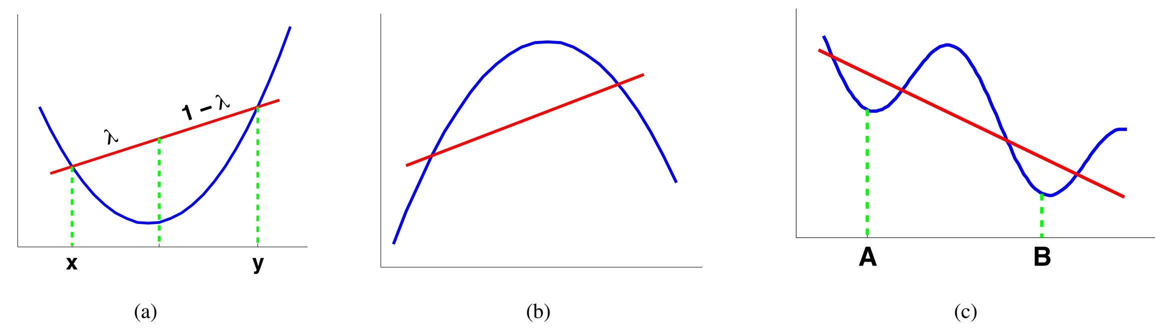


*Figure 1.28:* (a) Illustration of a convex function. We see that the chord joining $(x, f(x))$ to $(y, f(y))$ lies above the function. (b) Illustration of a concave function. (c) A function that is neither convex nor concave. A is a local minimum, B is a global minimum. Figure generated by `convexFnHand`.

See Figure 1.28 for a 1d example. A function is called **strictly convex** if the inequality is strict. A function $f(\boldsymbol{\theta})$ is **concave** if $-f(\boldsymbol{\theta})$ is convex. Examples of scalar convex functions include $\theta^2$, $e^\theta$, and $\theta \log \theta$ (for $\theta > 0$). Examples of scalar concave functions include $\log(\theta)$ and $\sqrt{\theta}$.

Intuitively, a (strictly) convex function has a "bowl shape", and hence has a unique global minimum $\theta^*$ corresponding to the bottom of the bowl. Hence its second derivative must be positive everywhere, $\frac{d}{d\theta} f(\theta) > 0$. A twice-continuously differentiable, multivariate function $f$ is convex iff its Hessian is positive definite for all $\boldsymbol{\theta}$.[9] In the machine learning context, the function $f$ corresponds to the NLL or penalized NLL.

Models where the penalized NLL is convex are desirable, since this means we can always find the globally optimal MAP estimate. We will see many examples of this later in the book. However, many models of interest will not have concave likelihoods or priors. In such cases, we will discuss ways to derive locally optimal parameter estimates.

#### 1.4.6.4 The benefits of more data

Regularization is the most common way to avoid overfitting. However, another effective approach — which is not always available — is to use lots of data. It should be intuitively obvious that the more training data we have, the better we will able to learn.[10] So we expect the test set error to decrease to some plateau as $N$ increases.

This is illustrated in Figure 1.29, where we plot the mean squared error incurred on the test set achieved by polynomial regression models of different degrees vs $N$ (a plot of error vs training set size is known as a **learning curve**). The level of the plateau for the test error consists of two terms: an irreducible component that all models incur, due to the intrinsic variability of the generating process (this is called the **noise floor**); and a component that depends on the discrepancy between the generating process (the "truth") and the model: this is called **structural error**.

In Figure 1.29, the truth is a degree 2 polynomial, and we try fitting polynomials of degrees 1, 2 and 25 to this data. Call the 3 models $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_{25}$. We see that the structural error for models $\mathcal{M}_2$ and $\mathcal{M}_{25}$ is zero, since both are able to capture the true generating process. However, the structural error for $\mathcal{M}_1$ is substantial, which is evident from the fact that the plateau occurs high above the noise floor.

For any model that is expressive enough to capture the truth (i.e., one with small structural error), the test error will go to the noise floor as $N \to \infty$. However, it will typically go to zero faster for simpler models, since there are fewer parameters to estimate. In particular, for finite training sets, there will be some discrepancy between the parameters that we estimate and the best parameters that we could estimate given the particular model class. This is called **approximation error**, and goes to zero as $N \to \infty$, but it goes to zero faster for simpler models. This is illustrated in Figure 1.29.

So far, we have been talking about test error, which is what we mostly care about. But it is also interesting to look at the training error vs $N$ for the different models. For models that can capture the truth, the training error will *increase* to some plateau as $N$ increases. The reason is this: initially the model is sufficiently powerful to simply memorize the training data, but as we are given more examples, it becomes harder to fit them perfectly given a fixed-complexity model. Eventually the error on the training set will match the error on the test set, as shown in Figure 1.29. (If the error on the training set increases with $N$, it is a sure sign that we are overfitting.)

In domains with lots of data, simple methods can work surprisingly well [HNP09]. However, there are still reasons to study more sophisticated learning methods, because there will always be problems for which we have little data. For example, even in such a data-rich domain as web search, as soon as we want to start personalizing the results, the amount of data available for any

[9] Recall that the Hessian is the matrix of second partial derivatives, defined by $H_{jk} = \frac{\partial f^2(\theta)}{\partial \theta_j \partial \theta_k}$. Also, recall that a matrix $\mathbf{H}$ is **positive definite** iff $\mathbf{v}^T \mathbf{H} \mathbf{v} > 0$ for any non-zero vector $\mathbf{v}$.

[10] This assumes the training data is randomly sampled, and we don't just get repetitions of the same examples. Having informatively sampled data can help even more; this is the motivation for an approach known as active learning, where you get to choose your training data.

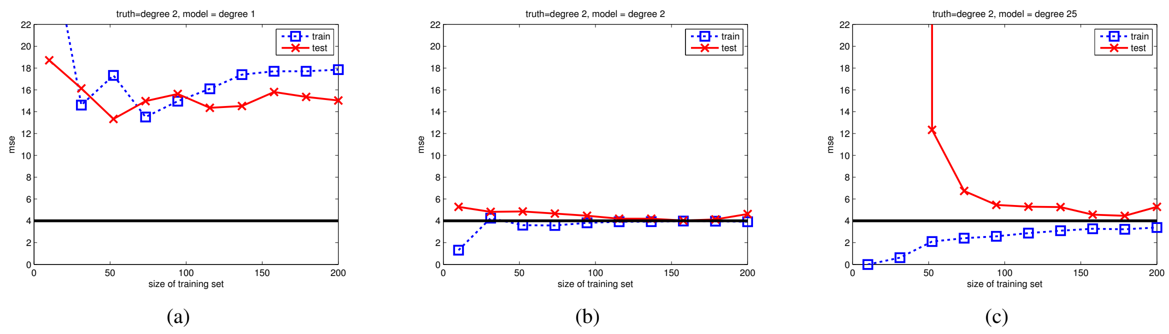


*Figure 1.29:* MSE on training and test sets vs size of training set, for data generated from a degree 2 polynomial with Gaussian noise of variance $\sigma^2 = 4$. We fit polynomial models of varying degree to this data. (a) Degree 1. (b) Degree 2. (c) Degree 25. Note that for small training set sizes, the test error of the degree 25 polynomial is higher than that of the degree 2 polynomial, due to overfitting, but this difference vanishes once we have enough data. Note also that the degree 1 polynomial is too simple and has high test error even given large amounts of training data. Figure generated by `linregPolyVsN`.

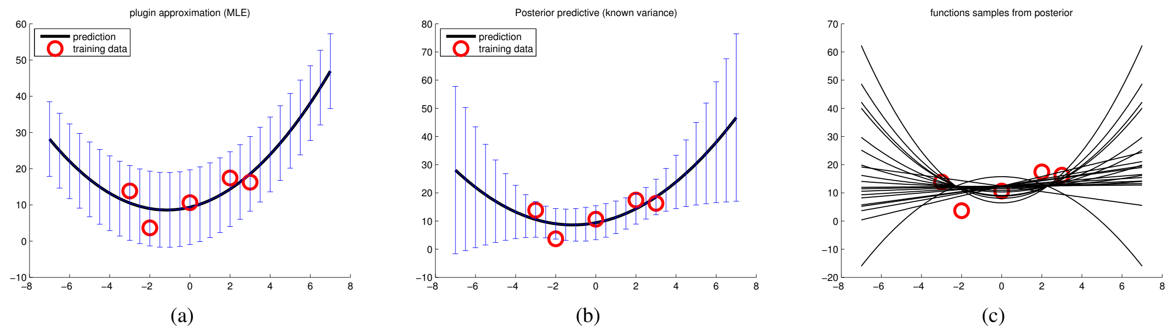


*Figure 1.30:* (a) Plug-in approximation to predictive density (we plug in the MLE of the parameters). (b) Posterior predictive density. Black curve is posterior mean, error bars are 2 standard deviations of the posterior predictive density. (c) 10 samples from the posterior predictive. Figure generated by `linregPostPredDemo`.

given user starts to look small again (relative to the complexity of the problem). In such cases, we may want to learn multiple related models at the same time, which is known as multi-task learning. This will allow us to "borrow statistical strength" from tasks with lots of data and to share it with tasks with little data. We will discuss ways to do later in the book.

### 1.4.7 Predicting the future

It's tough to make predictions, especially about the future. — Yogi Berra

To predict a future ouput $y$ given an input $\mathbf{x}$ and some training data $\mathcal{D}$, we should compute $p(y|\mathbf{x}, \mathcal{D})$. This is called the **posterior predictive distribution**, since it is the predictive distribution of the response variable $y$ given the input $\mathbf{x}$, after having seen the training data $\mathcal{D}$. By the law of total probability (Equation 2.4), this can be computed as follows

$$p(y|\mathbf{x}, \mathcal{D}) = \int p(y|\mathbf{x}, \boldsymbol{\theta}) p(\boldsymbol{\theta}|\mathcal{D}) d\boldsymbol{\theta} \tag{1.33}$$

where we have assumed that $p(y|\mathbf{x}, \mathcal{D}, \boldsymbol{\theta}) = p(y|\mathbf{x}, \boldsymbol{\theta})$, since knowing the parameters $\boldsymbol{\theta}$ renders the past training data irrelevant. Now suppose we approximate the posterior by a best guess, such as the MLE or MAP estimate:

$$p(\boldsymbol{\theta}|\mathcal{D}) \approx \delta_{\hat{\boldsymbol{\theta}}}(\boldsymbol{\theta}) \tag{1.34}$$

where $\delta_{\hat{\boldsymbol{\theta}}}(\boldsymbol{\theta})$ represents a delta-function, that puts all of its mass on one point (see Section 2.4.2). Inserting that into the above equation we get

$$p(y|\mathbf{x}, \mathcal{D}) \approx \int p(y|\mathbf{x}, \boldsymbol{\theta}) \delta_{\hat{\boldsymbol{\theta}}}(\boldsymbol{\theta}) d\boldsymbol{\theta} = p(y|\mathbf{x}, \hat{\boldsymbol{\theta}}) \tag{1.35}$$

This is called a **plug-in approximation** to the posterior predictive density. However, this approximation is suboptimal, since it ignores uncertainty in our parameter estimate. The Bayesian method often gives better results than the plug-in, since it averages over many possible predictions, and therefore avoids overfitting. In addition, it more accurately reflects the uncertainty in our

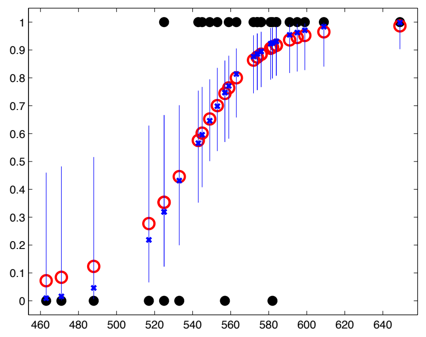


*Figure 1.31:* Posterior predictive density. The red circles is the median prediciton, and the lines denote the 5th and 95th percentiles of the predictive distribution. Figure generated by `logregSATdemoBayes`.

predictions. For example, Figure 1.30(b) shows that the Bayesian approach to linear regression has error bars that get wider as we move farther from the training data, whereas a plug-in approximation (even if we use MAP estimation) will result in fixed-sized error bars, since $\hat{\sigma}^2$ will be treated as a constant.

We can also sample from the posterior predictive, by sampling $\boldsymbol{\theta}^s \sim p(\boldsymbol{\theta}|\mathcal{D})$, and then use these samples as a plug-in estimate $p(y|\mathbf{x}, \boldsymbol{\theta}^s)$. The result is shown in Figure 1.30(c). This illustrates the set of functions that we think are probable given the data that we have seen so far.

As another example of this, consider logistic regression. In Figure 1.31, we plot the median and the 95% quantiles of the posterior predictive distribution, $p(y = 1|\mathbf{x}_i, \mathcal{D})$ for the SAT data; we approximate this by sampling from the posterior $p(\boldsymbol{\theta}|\mathcal{D})$ (see Section 7.5 for details). We see that the confidence in our predictions varies depending on the input that we observe. In particular, if we see a very low/ high SAT score, we are very confident the person will fail/pass the class, but we have more uncertainty about scores in the middle, which makes sense.

### 1.4.8 Bayesian model selection and Occam's razor

In Figure 1.21, we saw that using too high a degree polynomial results in overfitting, and using too low a degree results in underfitting. Similarly, in Figure 1.26(a), we saw that using too small a regularization parameter results in overfitting, and too large a value results in underfitting. In general, when faced with a set of models of different complexity (different *effective* number of parameters), how should we choose the best one? This is called the **model selection** problem, and is another important issue in machine learning.

One approach is to use cross-validation to estimate the generalization error of all the candiate models, and then to pick the model that seems the best. However, this requires fitting each model $K$ times, where $K$ is the number of CV folds. A more efficient approach is to compute the posterior over models,

$$p(m|\mathcal{D}) = \frac{p(\mathcal{D}|m)p(m)}{\sum_{m \in \mathcal{M}} p(m, \mathcal{D})} \tag{1.36}$$

From this, we can easily compute the MAP model, $\hat{m} = \operatorname{argmax} p(m|\mathcal{D})$. This is called **Bayesian model selection**.

If we use a uniform prior over models, $p(m) \propto 1$, this amounts to picking the model which maximizes

$$p(\mathcal{D}|m) = \int p(\mathcal{D}|\boldsymbol{\theta})p(\boldsymbol{\theta}|m)d\boldsymbol{\theta} \tag{1.37}$$

This quantity is called the **marginal likelihood integrated likelihood**, or the **evidence** for model $m$. The details on how to perform this integral will be discussed in Section 3.8.1. (Note that we were able to ignore this quantity for the purposes of parameter estimation, but it is necessary to compute it when performing model selection.)

One might think that using $p(\mathcal{D}|m)$ to select models would always favor the model with the most parameters. This is true if we use $p(\mathcal{D}|\hat{\boldsymbol{\theta}}_m)$ to select models, where $\hat{\boldsymbol{\theta}}_m$ is the MLE or MAP estimate of the parameters for model $m$, because models with more parameters will fit the data better, and hence achieve higher likelihood. However, if we integrate out the parameters, rather than maximizing them, we are automatically protected from overfitting: models with more parameters do not necessarily have higher *marginal* likelihood. This is called the **Bayesian Occam's razor** effect [Mac95b, MG05], named after the principle known as **Occam's razor**, which says one should pick the simplest model that adequately explains the data.[11]

[11] William of Occam (also spelt Ockham) was an English monk and philosopher, 1288–1348.

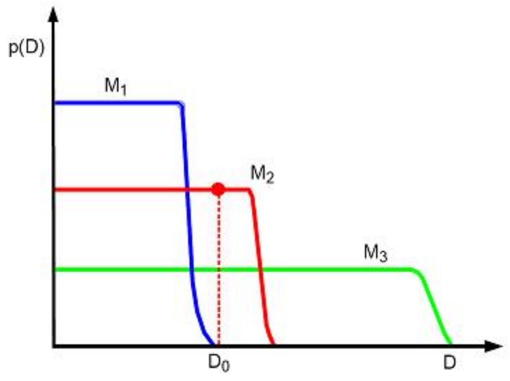


*Figure 1.32:* A schematic illustration of the Bayesian Occam's razor. The broad (green) curve corresponds to a complex model, the narrow (blue) curve to a simple model, and the middle (red) curve is just right. Based on Figure 3.13 of [Bis06a]. See also [MG05, Figure 2] for a similar plot produced on real data.

One way to understand the Bayesian Occam's razor is to notice that the marginal likelihood can be rewritten as follows, based on the chain rule of probability (Equation 2.5):

$$p(\mathcal{D}) = p(y_1)p(y_2|y_1)p(y_3|y_{1:2})\dots p(y_N|y_{1:N-1}) \tag{1.38}$$

where we have dropped the conditioning on $\mathbf{x}$ for brevity. This is similar to a leave-one-out cross-validation estimate (Section 1.4.3) of the likelihood, since we predict each future point given all the previous ones. (Of course, the order of the data does not matter in the above expression.) If a model is too complex, it will overfit the "early" examples and will then predict the remaining ones poorly.

Another way to understand the Bayesian Occam's razor effect is to note that probabilities must sum to one. Hence $\sum_{\mathcal{D}'} p(\mathcal{D}'|m) = 1$, where the sum is over all possible data sets. Complex models, which can predict many things, must spread their probability mass thinly, and hence will not obtain as large a probability for any given data set as simpler models. This is sometimes called the **conservation of probability mass** principle, and is illustrated in Figure 1.32. On the horizontal axis we plot all possible data sets in order of increasing complexity (measured in some abstract sense). On the vertical axis we plot the predictions of 3 possible models: a simple one, $M_1$; a medium one, $M_2$; and a complex one, $M_3$. We also indicate the actually observed data $\mathcal{D}_0$ by a vertical line. Model 1 is too simple and assigns low probability to $\mathcal{D}_0$. Model 3 also assigns $\mathcal{D}_0$ relatively low probability, because it can predict many data sets, and hence it spreads its probability quite widely and thinly. Model 2 is "just right": it predicts the observed data with a reasonable degree of confidence, but does not predict too many other things. Hence model 2 is the most probable model.

As a concrete example of the Bayesian Occam's razor, consider the data in Figure 1.33. We plot polynomials of degrees 1, 2 and 3 fit to $N = 5$ data points. It also shows the posterior over models, where we use a Gaussian prior (see Section 7.3 for details). There is not enough data to justify a complex model, so the MAP model is $d = 1$. Figure 1.34 shows what happens when $N = 30$. Now it is clear that $d = 2$ is the right model (the data was in fact generated from a quadratic).

As another example, Figure 1.26(c) plots $\log p(\mathcal{D}|\lambda)$ vs $\log(\lambda)$, for the polynomial ridge regression model, where $\lambda$ ranges over the same set of values used in the CV experiment. We see that the maximum evidence occurs at roughly the same point as the minimum of the test MSE, which also corresponds to the point chosen by CV.

When using the Bayesian approach, we are not restricted to evaluating the evidence at a finite grid of values. Instead, we can use numerical optimization to find $\lambda^* = \text{argmax}_\lambda \, p(\mathcal{D}|\lambda)$. This technique is called **empirical Bayes** or **type II maximum likelihood** (see Section 8.5 for details). An example is shown in Figure 1.26(c): the vertical line shows the EB estimate, which is close to the CV estimate, which in turn is close to the value that yields minimal test error.

### 1.4.9 No free lunch theorem

All models are wrong, but some models are useful. — George Box [BD87, p424].[12]

We have now seen several examples of classifiers: K-nearest neighbors, logistic regression, logistic regression with polynomial basis function expansion, etc. Similarly, we can define several kinds of regression models and unsupervised density models. Each model differs in terms of the assumptions it makes about the nature of the distribution, either $p(y|\mathbf{x})$ or $p(\mathbf{x})$. These assumptions are known as **inductive bias**.

Much of machine learning is concerned with devising different models (and different algorithms to fit them). Which model is best? In fact, there is no universally best model; this is sometimes called the **no free lunch theorem** [Wol96]. The basic idea

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

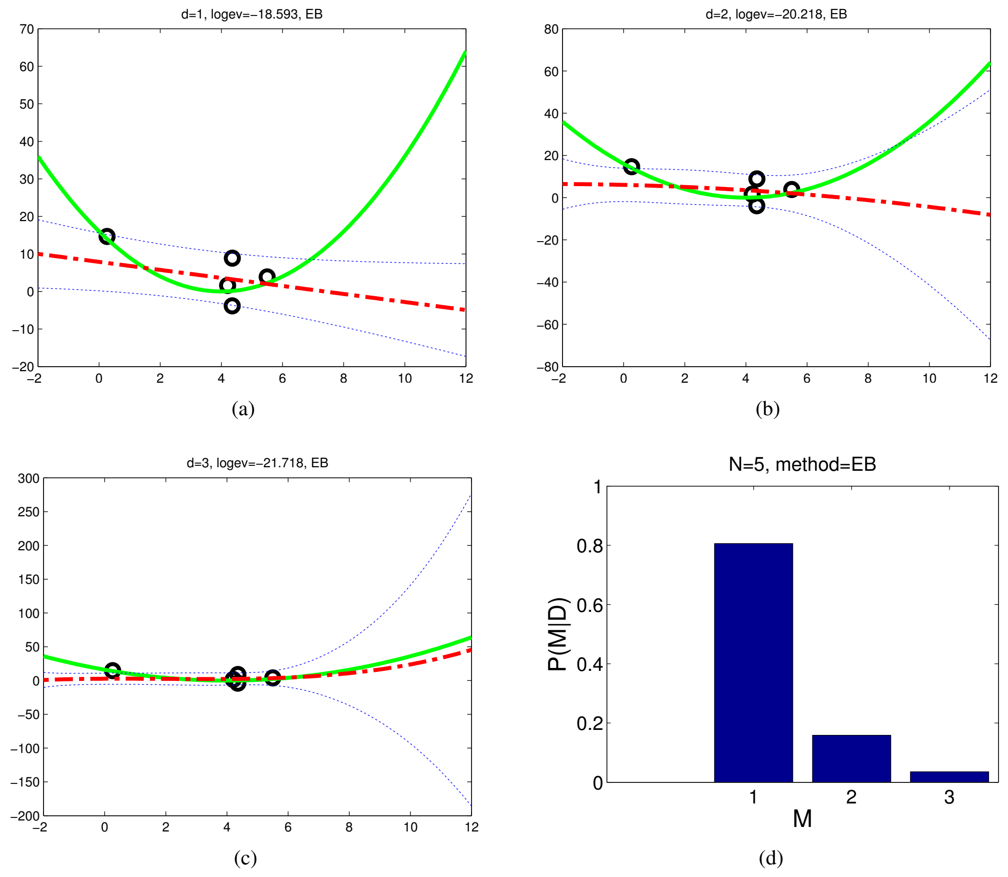


*Figure 1.33:* (a-c) We plot polynomials of degrees 1, 2 and 3 fit to $N = 5$ data points using empirical Bayes. The solid green curve is the true function, the dashed red curve is the prediction (dotted blue lines represent $\pm\sigma$ around the mean). (d) We plot the posterior over models, $p(d|\mathcal{D})$, assuming a uniform prior $p(d) \propto 1$. Based on A figure by Zoubin Ghahramani. Figure generated by `linregEbModelSelVsN`.

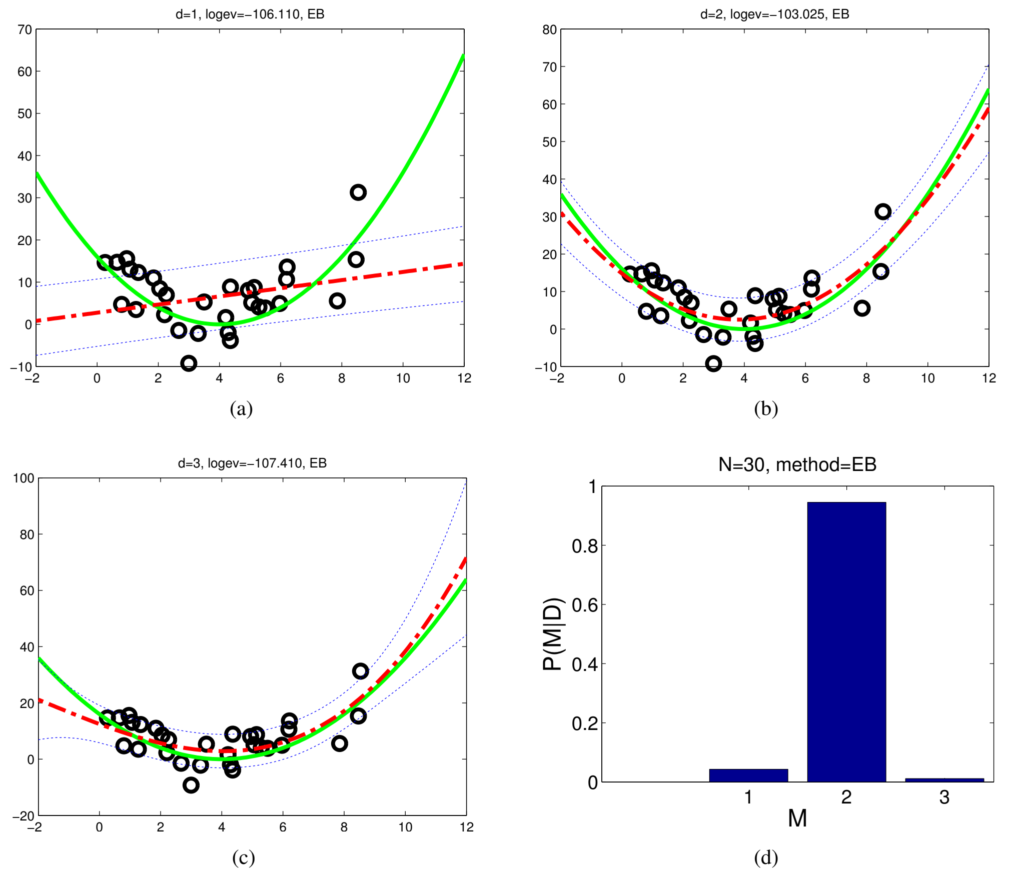


*Figure 1.34:* Same as Figure 1.33 except now $N = 30$. Figure generated by `linregEbModelSelVsN`.

is that a set of assumptions that works well in one domain may work poorly in another. For example, in the SAT example of Figure 1.22, we see that the assumptions behind logistic regression are quite reasonable; the use of something more complicated would not be advisable in such a simple setting, due to the risk of overfitting. However, more complex data sets may demand more complex models.

As a consequence of the no free lunch theorem, we need to develop many different types of model (likelihoods and priors), each of which corresponds to a different set of assumptions. This explains why you will encounter so many different models in this book. And for each model, there are often many different algorithms we can use to train the model, which make different speed-accuracy-complexity tradeoffs, as we will see.

## 1.5 Exercises

#### 1.5.0.1 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?

#### 1.5.0.2 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.

#### 1.5.0.3 Dummy encoding and linear models

(Source: [Hof09, p151])

Consider a linear regression model of the form

$$y_i = w_1 x_{i1} + w_2 x_{i2} + w_3 x_{i3} + w_4 x_{i4} + \epsilon_i \quad (1.39)$$

where $x_{i1} = 1$, $x_{i2} = g_i \in \{0, 1\}$ specifies if person $i$ is in a control group or not, $x_{i3}$age of person i, and $x_{i4} = x_{i2} \times x_{i3}$. So we have

$$\mathbb{E}[y|\mathbf{x}_i, g_i = 0] = w_1 + w_3 \text{age}_i \quad (1.40)$$

$$\mathbb{E}[y|\mathbf{x}_i, g_i = 1] = (w_1 + w_2) + (w_3 + w_4)\text{age}_i \quad (1.41)$$

Hence the difference in offsets between the two groups is $w_2$, and the difference in slopes is $w_4$. Sketch the regression line for the two groups assuming $w_1 = 1$, $w_3 = 1$ and with the following settings for the other parameters: (1) $w_2 = 0$, $w_4 = 0$, (2) $w_2 = 0$, $w_4 = 1$, (3) $w_2 = 1$, $w_4 = 0$, (4) $w_3 = 1$, $w_4 = 1$. You should have 4 figures, each with 2 lines. You can draw the figures by hand, or use Matlab. Assume the age ranges from 0 to 10.