Machine Learning Pattern Recognition

Lecture Notes

Dr. G. Englebienne

September 13, 2019

Contents

Chapter 1

Introduction

These lecture notes were written to provide you with a handy reference to the material that was presented in the Machine Learning: Pattern Recognition course. It is not meant to replace the book of the course [Bishop, 2007], rather, it is meant to illustrate topics that may seem a bit dry or emphasise subjects that are deemed important to this class. It will often repeat what was said in the class, in the hope that this will help you to understand the material better, but it is not meant to replace the lectures (read: I do hope that you will not try to skip the lectures because you have the lecture notes...)

1.1 Machine Learning

There is still no universal consent as to what artificial intelligence really is, exactly. But most of the definitions boil down to allowing machines to deal with (and excel in) their environment in an autonomous way. The variation in the definitions depends very much on what we define the environment to be. Early AI restricted the environment to, say, a chess game. Or to theorem proving. Those environments are inherently very complex, and we are far from having "solved" those problems. But we have made a lot of progress, and in some cases machines perform better than humans at those tasks. Deep blue beat Gary Kasparov in 1997. However, the physical world we live in is inherently far more complex than such "toy" worlds; the main inherent difficulty of the physical world is uncertainty.

Yet biological machines deal with the physical world quite well. So we know it can be done, and we have quite a challenge to meet. The fact that we can do things ourselves doesn't help us very much in getting machines to do it: there are many problems we can solve very well (*e.g.*, walking, understanding speech, reading natural text, producing grammatically correct speech, recognising an emotion on somebody's face, ...), but we don't really know how we do it. Each of these problems is very complex, and we don't know how to get a machine to do them. So if we ever want to have machines that can do these things, they'll have to learn to do them by themselves. And that's what this course is about.

1.2 Smoothness, distributions and such.

So how can we know what to do with a new observation, from other observations that we've seen before. As an example, how can we decide that two flowers are from the same species rather than from different species? First, we need to find out what the observation is going to consist of. As humans, we have multiple different senses that we rely on to learn what we're interested in: vision, hearing, touch, smell, taste. Some senses are better for some tasks than others, and one aspect of the learning process is to learn which sense to use for what task, and how to combine these measurements. Similarly, in machine learning, we have access to different sensors, measurements and preprocessing methods. So imagine that our sensors provide us with length measurements of the petals and sepals of flowers. We could plot these, as in Figure 1.1. In this plot, the different point styles indicate measurements from different varieties, and this provides us with the following valuable insight: the datapoints corresponding to the different varieties generally occupy different regions of the feature space. In general, the points will have a certain distribution in the space. In this case,

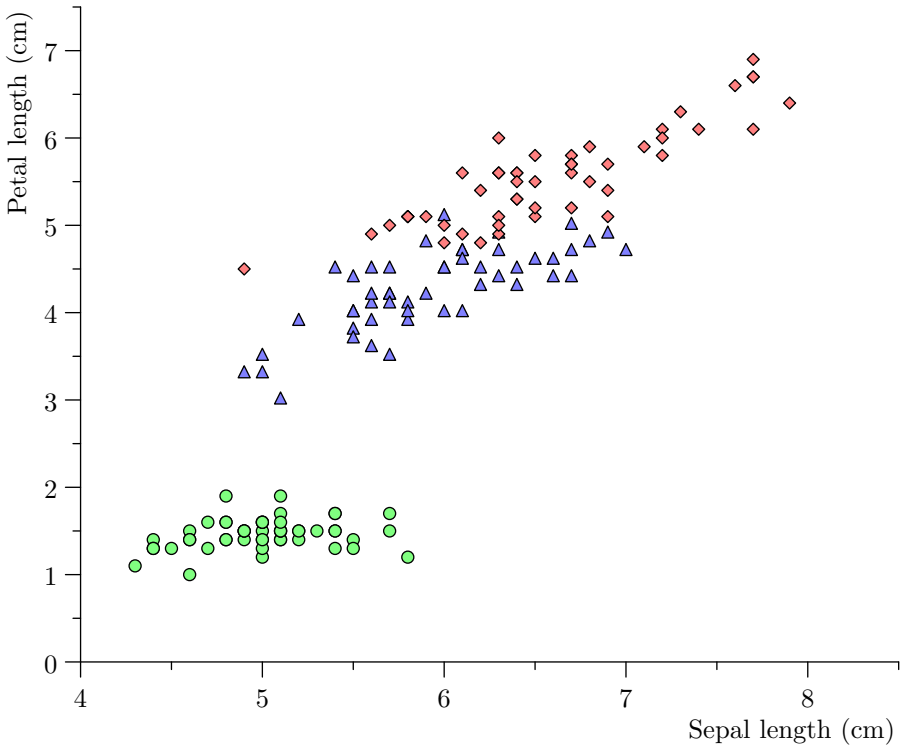


Figure 1.1: Distribution of measurements of petal and sepal lengths for three varieties of iris flowers: *Iris Setosa* (green circles), *Iris versicolor* (blue triangles) and *iris virginica* (red squares).

the datapoints from the three varieties we are looking at all have slightly different distributions, and we can use that to differentiate between them. One way for a learning machine to differentiate between these varieties, would therefore be to learn what the corresponding distributions are in the measurement space. One key assumption that we will often make, is that the datapoints are independent (that is, the position of a datapoint depends on what class it belongs to and, given that class, not on the positions of the other datapoints), and that they all have the same distribution in the space (they are identically distributed). We call such datapoints "independent and identically distributed" or "i.i.d."

1.3 Areas of Machine Learning

There are multiple ways in which machine learning can be organised. One frequently used taxonomy of machine learning techniques classifies techniques and algorithms based on whether the desired output is known. If this is the case, we call the problem a *supervised learning* problem. In this case, we possess a number of training examples, for which we know the desired output. If the desired output is not known, the problem is an *unsupervised learning* problem. In this case, we ask the machine to optimise some function, but we do not really know what the output should be.

1.3.1 Supervised learning

Supervised learning problems are typically problems that humans can solve or know the answer to, but don't really know how to solve. Examples may be speech recognition, or weather prediction (simply try to predict today's weather based on last week's measurements). These techniques are often classified according to their purpose. We can discriminate between two great families of supervised machine learning techniques based on this distinction:

Classification techniques Sometimes, the purpose of a machine is to discriminate between a finite number

of classes. For example, we may want an intelligent car to be able to discriminate between pedestrians and road signs, or a quality control robot to discriminate between correctly manufactured items and manufacturing failures, *etc.*. This is a broad class of problems, and many different machine learning techniques focus only on these.

Regression techniques In other settings, the purpose of a machine will be to output continuous values. For example, we may want an intelligent car to be able to control the steering in such a way as to stay on the road, or to control speed so as to bring us to our destination in a speedy, comfortable and safe way, or we may want a machine to be able to predict the amount of rainfall for tomorrow. Or the amount of electrical power that will be required at 5pm, tomorrow. Techniques that focus on outputting such a continuous value are called regression techniques.

The problem of supervised learning can be approached from different angles, which again leads to a taxonomy of methods:

Non-parametric methods As we mentioned, it is not possible to store all the measurement values that might ever occur, and to store the desired answer for that measurement value. However, since we do have some example values with corresponding label, we could store those (or a selection of those) and perform classification or regression on new measurements based on how similar those measurements are to the stored values. Such methods are called non-parametric methods.

Discriminant functions Another possible approach is to choose a function that will provide an output of the desired format for a given input of the measurement's format. For example, in a two-class classification problem, we may chose a function that returns $f(\mathbf{x}) = 0$ for a measurement $\mathbf{x}$ in class $\mathcal{C}_1$ and $f(\mathbf{x}) = 1$ for a measurement in class $\mathcal{C}_2$. Learning then consists of finding values for the parameters of this function, so that it performs correctly on the largest possible number of examples.

Model-based approaches Instead of finding a function that will hopefully provide us with the right output given some input, we could also look at what the inputs look like for each possible output. In other words, we build a model of the data, and use it to perform the task at hand. To do this, we need to assume that the data's distribution can be described by a probability density function of a certain family (often, the Gaussian distribution is used for this), and we then need to estimate the parameters of that distribution. When we know the distribution of the data for each possible class, we can compute the probability that a new datapoint should be seen if it belonged to any of the classes, and based on that we can compute the probability that a datapoint belongs to any given class. The big advantage of this method is that it is possible to not only provide the most likely classification, but also a confidence estimate of that classification.

1.3.2 Unsupervised learning

Unsupervised learning techniques are given the measurements, but no information as to what we expect to obtain from this data. Such techniques can be divided into two broad categories: clustering and dimensionality reduction.

Clustering Clustering deals with the problem of trying to group data by identifying some structure in the data, when no corresponding labels are known. This is something that humans excel at, but that is very hard to evaluate objectively. Finding the "correct" *number* of clusters is an especially challenging task. Clustering can be very useful as a form of data compression: it is often not necessary to know what the measurement was in order to perform a given task: knowing which cluster it belonged to is often enough. For example, consider the task of predicting a plane's speed from its size. It is probably enough to know that a plane's size falls in the "tourism", "fighter jet" or "commercial transport" category in order to be able to make the prediction. The precise size measurements are not relevant.

Dimensionality reduction It is often possible to make many measurements simultaneously, yielding very high-dimensional data. As an example, consider a digital camera. Such a camera may have a few million sensors, each registering how much light fell on them during a given time span. Each picture taken with

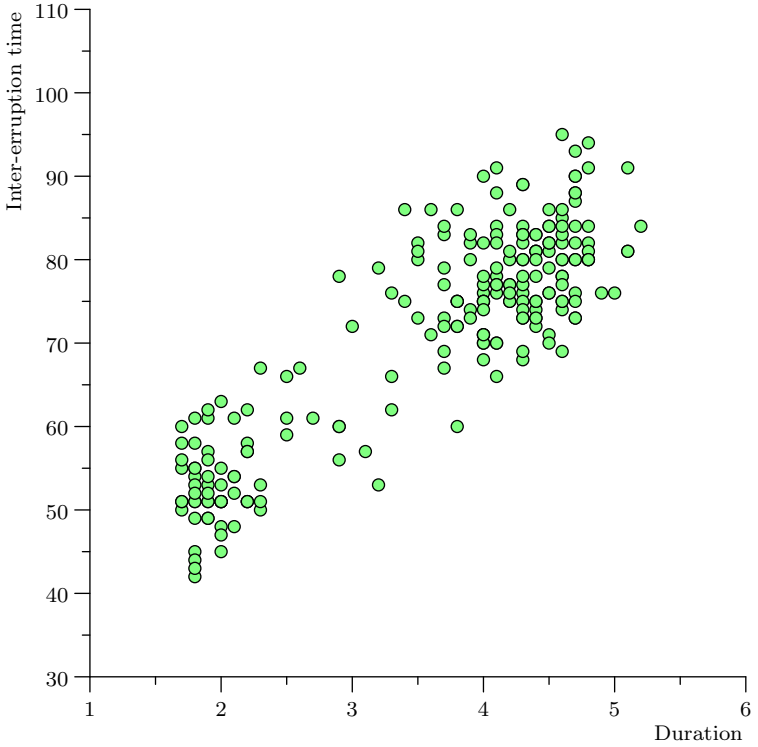


Figure 1.2: Example of some measurements that might be used for clustering. Humans will readily detect two clusters in this data. But how did we do that? Most importantly, why don't we see three clusters in there?

such a camera may therefore be seen as a very high-dimensional vector of measurements. We cannot list all the possible images that such a sensor could take, because there are far too many possibilities — even though this number is finite. We do know, however, that real images will have characteristics that limit the number of possible valid images. For example, most pairs of neighbouring pixels in a real image will have very similar colour and intensity. We can therefore reduce the dimensionality of the datapoint without discarding any relevant information. Automated techniques to do this are called dimensionality reduction techniques.

Chapter 2

Math preliminaries

There are a few mathematic basics that we need to be familiar with for this course. They may require some time getting acquainted with or, if you have already seen these things in detail before, they may benefit from being refreshed. In this chapter, we briefly review vector and matrix arithmetic, probability theory, and function differentiation. We do not go into much detail: we merely review those concepts that are needed for our purposes.

2.1 Vectors

The first concept we must have a look at, are vectors. Vectors define a magnitude and a direction in a space. In machine learning, vectors are typically feature vectors, consisting of the measurement of a set of features. The N-dimensional vector $\mathbf{v}$ is defined as:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ \vdots \\ v_N \end{bmatrix} \tag{2.1}$$

We will follow the convention that a vector is denoted by a boldface lowercase letter. We can define the sum of two N-dimensional vectors $\mathbf{a}$ and $\mathbf{b}$ as

$$\mathbf{a} + \mathbf{b} = \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix} + \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 + b_1 \\ \vdots \\ a_n + b_n \end{bmatrix} \tag{2.2}$$

Geometrically, the vector sum and the corresponding multiplication of a vector with a scalar are illustrated, in two dimensions, in Figure 2.1. In general, the definition of a vector depends on the space it is defined in, but we limit ourselves to Hilbert spaces, *i.e.*, a generalisation of the Euclidean space to more than three dimensions. We define the *dot product*, *inner product* or *scalar product* (lots of names for the same thing) of

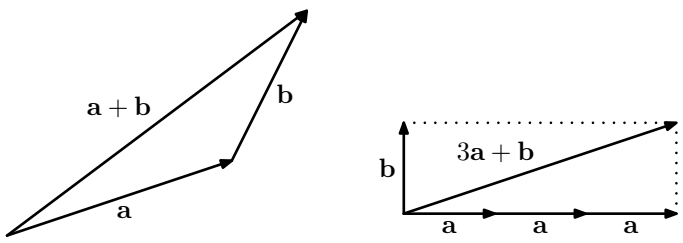


Figure 2.1: Geometric illustration of the vector sum, in two dimensions

two vectors **a** and **b**, as follows:

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^\top \mathbf{b} = \sum_{i=1}^{N} a_i b_i \tag{2.3}$$

$$= a_1 b_1 + \cdots + a_N b_N \tag{2.4}$$

We can now define important concepts:

The length or 2-norm of a vector

$$|\mathbf{a}| = \sqrt{\mathbf{a} \cdot \mathbf{a}} \tag{2.5}$$

The angle between two vectors is related to the inner product as:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos\theta \tag{2.6}$$

A unit vector is a vector whose length equals one: $|\mathbf{a}| = 1$

Vector projection If **a** is a unit vector, the inner product $\mathbf{a} \cdot \mathbf{b} = |b| \cos(\theta)$, is the length of the orthogonal projection of **b** onto **a**. Here θ is the angle between **a** and **b** (see Figure 2.2). If **a** is not a unit vector, this result is scaled by the length of **a**. We shall rely on vector projection pretty extensively when discussing dimensionality reduction, so it is important you understand this concept.

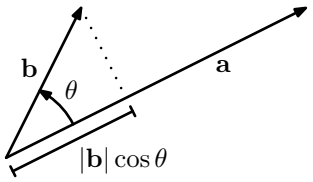


Figure 2.2: Vector projection in two dimensions, where **a** is a unit vector

Linear independence Two vectors are linearly independent if their inner product is zero ($\theta = 90^o$; they are orthogonal).

2.2 Matrices

Matrices are arrays of scalars, called the entries of the matrix, and are denoted with uppercase bold letters. A matrix of order $m \times n$ has m rows and n columns. A vector can therefore be seen as matrix of order $n \times 1$. Matrix addition is defined similarly to vector addition:

$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} b_{11} & \cdots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nm} \end{bmatrix} \tag{2.7}$$

$$\mathbf{A} + \mathbf{B} = \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1m} + b_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} + b_{n1} & \cdots & a_{nm} + b_{nm} \end{bmatrix} \tag{2.8}$$

The matrix product is similar to the vector inner product, where the inner product is applied to every combination of a row in **A** with a column in **B**. It is only defined for matrices of order $n \times m$ and $m \times p$

(where the number of columns of $\mathbf{A}$ equals the number of rows of $\mathbf{B}$), as follows:

$$\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} b_{11} & \cdots & b_{1p} \\ \vdots & \ddots & \vdots \\ b_{m1} & \cdots & b_{mp} \end{bmatrix} \tag{2.9}$$

$$\mathbf{AB} = \begin{bmatrix} a_{11}b_{11} + \cdots + a_{1m}b_{m1} & \cdots & a_{11}b_{1p} + \cdots + a_{1m}b_{mp} \\ \vdots & \ddots & \vdots \\ a_{n1}b_{11} + \cdots + a_{nm}b_{m1} & \cdots & a_{n1}b_{1p} + \cdots + a_{nm}b_{mp} \end{bmatrix} \tag{2.10}$$

Based on these definitions, a few special matrices and matrix operations can be defined:

- The unit matrix or identity matrix, for which all entries are zero except the entries on the diagonal:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \tag{2.11}$$

  As a consequence, $\mathbf{AI} = \mathbf{IA} = \mathbf{A}$

- The inverse matrix of a square matrix (*i.e.*, a matrix of order $n \times n$), is defined such that

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I} \tag{2.12}$$

  Computing the inverse of a matrix is, in general, an expensive operation. It is, however, very useful in lots of situations.

- The transpose of a matrix: $\mathbf{A}^\top$ is defined such that if $\mathbf{A} = \mathbf{B}^\top$, $a_{ij} = b_{ji}$.

- Eigenvectors and eigenvalues. An eigenvector and eigenvalue of a matrix $\mathbf{A}$ are a vector $\mathbf{v}$ and a scalar λ, such that

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v} \tag{2.13}$$

  In other words, a vector $\mathbf{v}$ such that premultiplying it with the matrix $\mathbf{A}$ changes its length, by λ, but not its orientation. Such eigenvalue–eigenvector systems have many applications, and they will come back occasionally during the course.

2.3 Probability theory

A lot of what we will see this year is based on the theory of probabilities. This theory is simple and elegant, and extremely powerful. In probability theory we consider random variables, which we denote with capital letters. These variables can take different values with some probability; when a random variable takes a value, we call this an instantiation. Instantiations are denoted with lowercase letters, and the probability that a random variable, say A takes on the value a is denoted $p(A = a)$, often shortened to $p(a)$ for brevity. So for example, let R be a random variable denoting whether it is raining in Amsterdam. This variable can take the following values: r, indicating that it is raining and $\neg r$, indicating that it is not. The probability that it is raining at any given point in time is then given by $p(r)$. The expression $p(R)$ could be used to indicate the probability distribution over all possible values of R and would, in this case, correspond to the vector $(p(r), 1 - p(r))^\top$.

This probability is in general of course very high, but it will be very different in summer and in winter: it will be higher in summer, and lower in winter (because then it is either foggy, or it snows, or it hails. Or I'm abroad, in which case the weather is probably lovely in Amsterdam.) This brings us to the concept of

conditional probabilities: the probability that it rains *given that it is winter (w)* is denoted $p(r|w)$. Notice that this does not tell us the probability that it is winter, nor the probability that it rains "in general" — called the *marginal probability* that it rains: in order to know that, we would need to know the probability that it is winter (which seems to be a lot more than 0.25 around here). The *joint* probability that it is raining and it is winter, is denoted by $p(r, w)$. The complete joint probability distribution can then be written as:

$$P(R, W) = \begin{bmatrix} p(r, w) & p(r, \neg w) \\ p(\neg r, w) & p(\neg r, \neg w) \end{bmatrix} \tag{2.14}$$

Probability theory is based on the following two axioms:

$$\text{If } \models A, \text{ then } p(A) = 1 \tag{2.15}$$

$$\text{If } \neg(A \wedge B), \text{ then } p(A \vee B) = p(A) + P(B) \tag{2.16}$$

These axioms mean that *if an event is always true, the probability of that event is one* and *if two events are mutually exclusive, the probability of one or the other being true is the sum of their respective probabilities.*

All other theorems of probability theory derive from these axioms. Below, we give the most important rules of probability, which we will use time and again during this course. We derive the first two for binary variables, just to convince you that the above axioms are all we need to develop probabilistic reasoning.

The probability $p(\neg a) = 1 - p(a)$ This follows directly from the axioms above: a and $\neg a$ are mutually exclusive, so that $p(a \vee \neg a) = p(a) + p(\neg a)$. Moreover, $\models a \vee \neg a$, and therefore, $p(a \vee \neg a) = p(a) + p(\neg a) = 1$.

Sum rule of probabilities The sum rule of probabilities states that $p(a, b) + p(a, \neg b) = p(a)$. We can prove this as follows:

$$a \models a \wedge (b \vee \neg b) \tag{2.17}$$

$$= (a \wedge b) \vee (a \wedge \neg b) \tag{2.18}$$

Therefore, since those are mutually exclusive, $p(a) = p(a, b) + p(a, \neg b)$.

We have shown this for binary random variables, but the proof can be extended to multinomial variables (variables that can take more than two values) and continuous variables (variables that can take any value between $\pm\infty$). When the joint probability of two variables is known, the process of computing the probability of one variable independently from the other is called marginalisation. This resulting probability is called the marginal probability

Product rule of probabilities The joint probability of two events a and b, written $p(a, b)$, can be decomposed as follows:

$$p(a, b) = p(a|b)p(b) \ . \tag{2.19}$$

This is read as "The probability of *a and b* equals the probability of *a given b* times the (marginal) probability of *b*"

2.3.1 Bayes' theorem

One of the most famous theorems of probability theory, and certainly an extremely valuable one in machine learning, is Bayes' theorem. This theorem states that

$$p(a|b) = \frac{p(b|a)p(a)}{p(b)} \tag{2.20}$$

The theorem is easy to prove from the product rule of probabilities:

$$p(a|b)p(b) = p(a, b) = p(b|a)p(a) \tag{2.21}$$

$$p(a|b) = \frac{p(b|a)p(a)}{p(b)} \tag{2.22}$$

2.3.2 Dealing with probabilities, the Graphic Novel

The basic rules of probabilities are quite straightforward and provide us with an extremely powerful tool, but their interpretation and the intuitive grasp is elusive and may sometimes be hard to obtain. Dealing with probabilities is something that requires practice. In this section, I want to walk us through a — purely hypothetical — example and illustrate it graphically.

Let us consider the the following situation.[1] It is 3 o'clock in the morning. Pouring rain slowly awakes a lonely student on a park bench. As he regains approximate consciousness, our soggy hero wonders where he is. For reasons of his own, he does not wonder how he got there. To answer this burning question, he has at his disposal some basic facts of his life:

1. He must be either in Enschede, Amsterdam or Rotterdam. Those are the only cities where his present predicament could conceivably befall him. But if he had no further information, he would be equally likely to be in any of these cities.

2. Based on his extensive past experience, our student knows more, however. He knows that when he goes out in Enschede, the next morning he does not remember what happened half of the time. In Amsterdam, things are even worse, he blacks out in 80% of the cases. In Rotterdam, however, he tends to be very reasonable and this kind of situation has only happened to him about 10% of the time.

Armed with these facts, and with his intimate familiarity with the basic tools of probability, our young friend starts to figure out where he is. The prior probability that the city he is in is Enschede, Amsterdam or Rotterdam is the same in all cases: $p(C = e) = p(C = a) = p(C = r) = \frac{1}{3}$. This is illustrated graphically in Figure 2.3 (a). The fact that he has trouble remembering the finer details of what happened last night depends on the city he is in, and can be formalised as $p(M = \emptyset|C = e) = 0.5, p(M = \emptyset|C = a) = 0.8$ and $p(\emptyset|C = r) = 0.1$. This is shown in Figure 2.3 (b), and can now be used to compute the joint probability of the city the student is in, and whether he would black out in that city. This is illustrated in Figure 2.3 (c), which shows the probabilities $p(M = \emptyset, C = e), p(M = \neg\emptyset, C = e), p(M = \emptyset, C = a), p(M = \neg\emptyset, C = a), \ldots$ In general, this is $p(C, M)$.

Now our student may not be at the zenith of his mental capacities, but he can still do Bayesian inference. He knows that the probability that he is in a particular city, say Enschede, $p(C = e|M = \emptyset)$ or $p(e|\emptyset)$ for brevity, can be computed using Bayes' theorem: $p(e|\emptyset) = \frac{p(\emptyset|e)\,p(e)}{p(\emptyset)}$. Of these, $p(\emptyset)$ is not readily available to our student's addled brain, although it is hovering near the surface of his consciousness: he knows $p(C, M)$ (Figure 2.3 (c)), what he needs is the subset of that probability that corresponds to $p(M = \emptyset)$, which he gets by going over all the cities he might be in and adding up the probabilities that he might be in those cities and blacking out in there: $p(\emptyset) = \sum_c p(C = c, M = \emptyset)$, which is illustrated in Figure 2.3 (d). Now he does realise that $M = \emptyset$ is a given, so using Bayes' rule, he rescales all the $p(\emptyset, C)$ so that, effectively, $p(\emptyset|\emptyset) = 1$ and the scaled probabilities of being in a particular city represent $p(C|\emptyset)$. This is illustrated in Figure 2.3 (e).

Integrating new evidence But the hero of our story is not done yet. He is, by now, getting thoroughly soaked, and he knows that the current weather is not related to his being there in the first place. At least, he tries very hard to convince himself that, even though this kind of stuff seems to happen to him all the time, no, there is no rain god and the bastard does not hate him. No!

So, having done this bit of mental morning gymnastics, he proceeds to infer where he is, assuming that $R \perp\!\!\!\perp M|C$: rain is conditionally independent of his memory loss given the city he's in (in other words, it's not because he's here and he's feeling under the weather that the weather is crap). His current belief is that he is in Enschede with a probability of approximately 36%, in Amsterdam with 57% and in Rotterdam with 7% (Figure 2.3 (e), repeated in Figure 2.4 (a) for convenience). He knows that the climate is different in these cities: Enschede has a deeply continental climate, while Amsterdam and Rotterdam have gentle maritime climates. The probability that you'll wake up soaked in rain at three in the morning at this time of year is 80% in both Amsterdam and Rotterdam, but only 20% in Enschede (Figure 2.4 (b). So our disciple of the reverend applies his mentor's theorem all over again to update his belief of his location: he computes the joint probability of rain and location (Figure 2.4 (c)), marginalises out the location to get the probability that

[1] Any resemblance to actual persons, living or dead, is purely coincidental

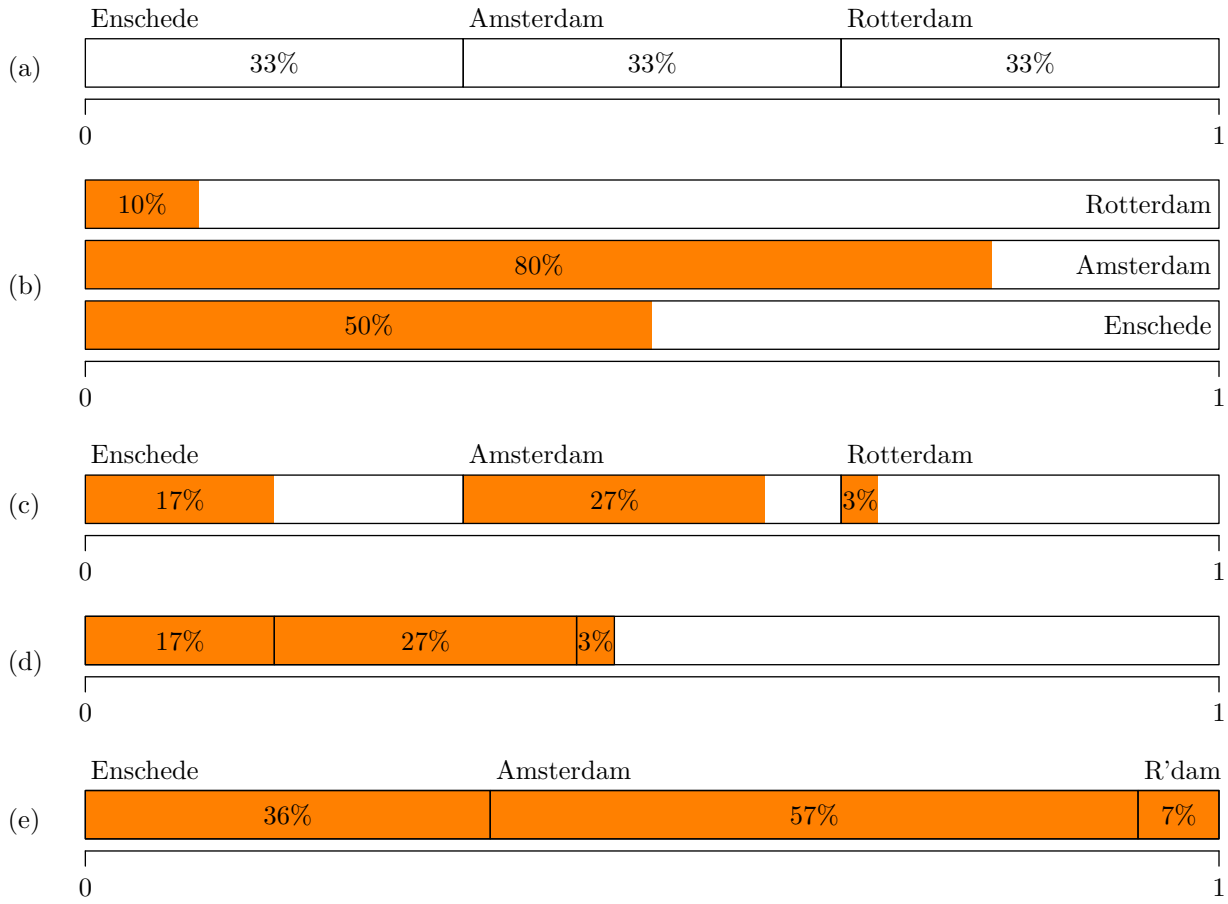


Figure 2.3: Graphical representation of the probability mass assigned to the various instantiations described in our hypothetical example (see text for details): (a) prior probability of being in a particular city, (b) likelihood of observing memory loss given the city, (c) joint probability — notice how the probability of being in a city remains identical to the prior, (d) re-arranged version of c, (e) posterior probability of the location given that we observed memory loss

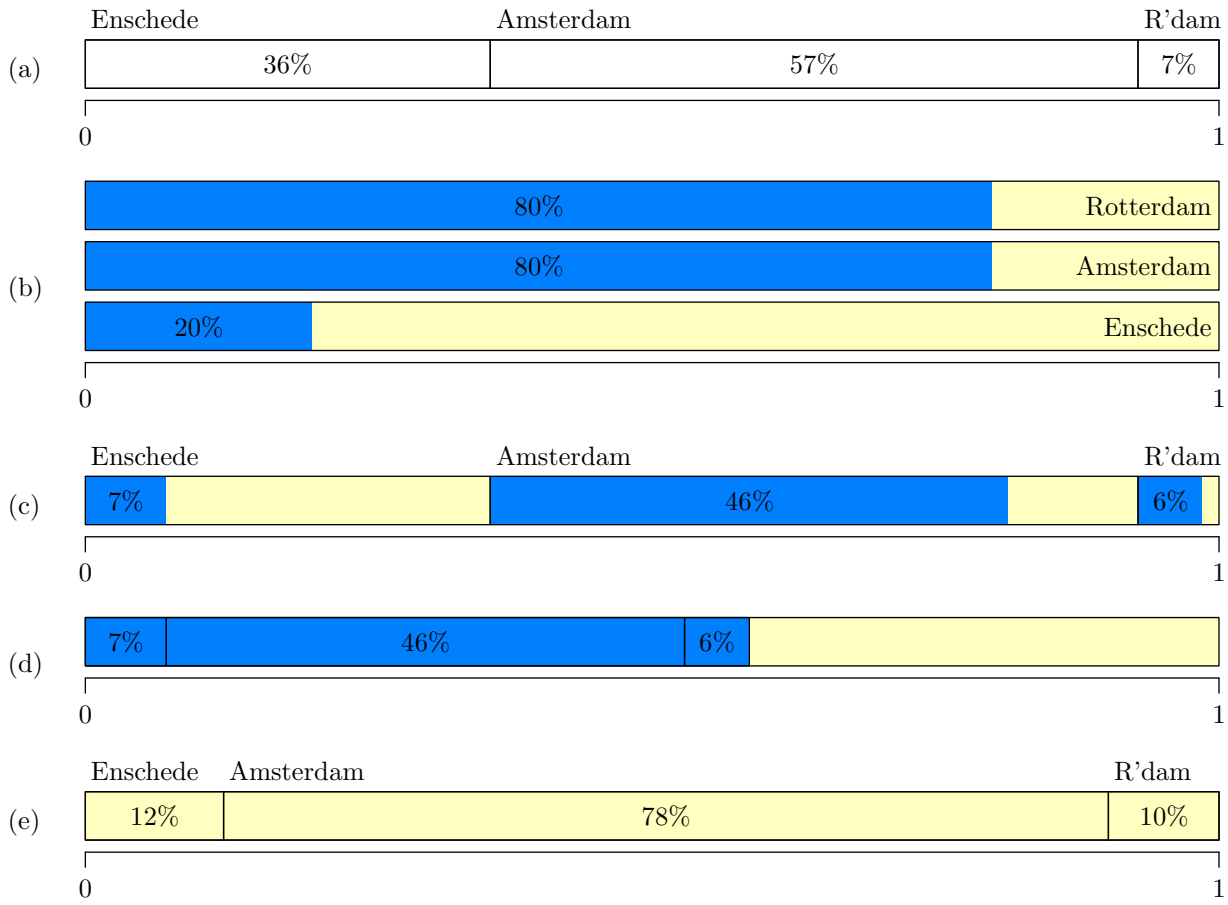


Figure 2.4: Integrating further evidence in our purely fictional example. See text for imagined details.

it rains (Figure 2.4 (d)) and computes the posterior probability that he is in any of these cities (Figure 2.4 (e)). He is now pretty sure that he must be in Amsterdam.

This capacity to apply Bayes' theorem repeatedly, taking the posterior of the previous iteration as prior for the next iteration, makes Bayesian inference extremely powerful. Keep in mind, however, that this recursive application is only valid if the observations are independent (given the state of the variable we are trying to estimate).

Logic Probability theory and the application of Bayes' rule can be seen as an extension to predicate logic. To illustrate this, at the moment our friend believes he is probably in Amsterdam and starts thinking about going home when, suddenly, the blaring cry of a foghorn echoes through the night. The gentle throbbing is instantly transformed into a splitting headache.

Now, the sound of a foghorn in the middle in a city is not a common occurrence. In fact, he has never experienced this before, and there are good legal reasons for that. He does wonder for a moment whether he may have imagined it. But the headache is still there, and is proof enough that he did not hallucinate . . . this time. He wonders whether his friends may have something to do with it, and considers that if they wake up in a cell tomorrow, at least they'll be dry.

But if somebody sounded a foghorn, that would mean there is a cargo ship nearby, and that would mean he must be in Rotterdam instead. Wait, he thinks, slow down — let's do this properly. The probability of hearing a foghorn in Enschede is zero; it's a landlocked city, there are no cargo ships there. Seagoing vessels can fare up Amsterdam, but essentially only cruise ships do so. OK, let's say the probability of hearing a foghorn in the city would be infinitesimally small, but not zero. Let's say, 10^{-8}, meaning that one could hear a foghorn blaring through the city once in a hundred thousand nights. In Rotterdam, this

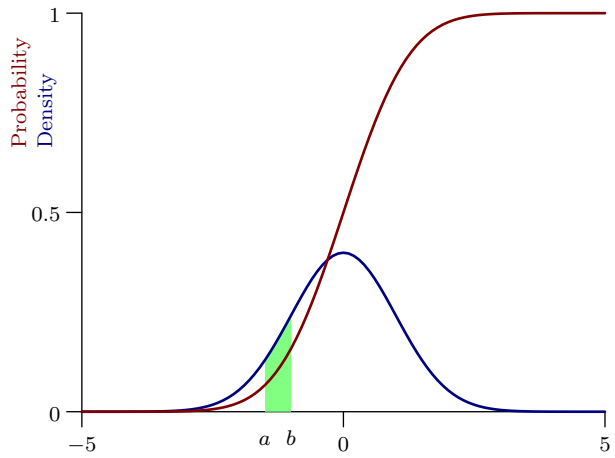


Figure 2.5: Illustration of a PDF, the Gaussian distribution with zero mean and unit variance (blue), and the corresponding CDF (red). Notice how the CDF is below the PDF for some values of x: the CDF indicates probabilities, the PDF indicates probability densities.

probability is similarly very small, but not nearly as small. Let's say, once in a thousand nights. Again applying Bayes' rule to his previous belief that there was a 12% chance he was in Enschede, a 78% chance he was in Amsterdam and a 10% chance that he was in Rotterdam, our evidence now becomes very small ($0 + .78 \times 10^{-8} + 10 \times 10^{-3} = 0.0100000078$). In other words, our Epicurean friend finds himself in an unexpected situation, namely: hearing a foghorn in the middle of the city in the middle of the night, but it is the situation he observes. And it is a very informative observation: applying Bayes' rule tells him there's a 99.99992% chance that he is actually in Rotterdam. If Amsterdam had no port, the likelihood of hearing a horn there would have been zero as well, and the probability of being in Rotterdam would have been one — our problem would have degenerated into a straightforward problem of predicate logic.

Having found his bearings, T. gets up and stumbles towards the railway station. If he hurries, he'll be in time for his beloved lecture in Machine Learning.

2.3.3 Probability Density Functions

When a random variable can take a finite number of values, the sum of the probability that the variable takes on each of these values must be one (since the values are, of course, mutually exclusive). The probability is distributed over the possible values, and is called a probability distribution. If the variable is continuous, however, it can take on infinitely many values. The probability of any one value must therefore be zero.

We can, however, define a probability density function (PDF) to indicate how the probability is distributed over the continuous space. Such a function allows us to compute the probability that the variable should lie in a certain interval $[a, b]$, by integrating the PDF over that interval. The integral of this function over the complete domain of the function must be one, since any valid parameter value must lie in the function's domain. Closely related to the PDF is the cumulative density function (CDF), which is defined as

$$\mathrm{CDF}(x) = \int_{-\infty}^{x} \mathrm{PDF}(v)\,\mathrm{d}v \tag{2.23}$$

The CDF is useful when we want to compute the probability that a value should lie in an interval, since from the definition we get that $p(x \in [a, b]) = \mathrm{CDF}(b) - \mathrm{CDF}(a)$. To illustrate this, Figure 2.5 shows a common PDF, the Gaussian distribution, and the corresponding CDF. In practice, the difference between probability density and probability distribution matter little, and in this course we will rarely distinguish between them.

2.4 Function differentiation

In machine learning, we are often able to state a learning problem as the optimisation of some function. Doing so will typically require the derivative of the function with respect its parameter or, if the function has multiple parameters, the function's gradient with respect to those.

Function optimisation is sometimes possible analytically, by finding the function's gradient and setting it equal to zero. If we can solve the resulting (set of) equation(s), we have found an optimum of the function. Often, however, the function will be too complex to be able to find its optimum in closed form. In that case, iterative methods such as gradient descent, or more advanced gradient-based methods will be required. Here, we review some basic principles of function differentiation so that you are familiar with them when we need them later on.

2.4.1 Basic differentiation

Here are some basic rules of differentiation:

$$\frac{\partial}{\partial x} x^a = a x^{a-1} \qquad \frac{\partial}{\partial x} \sin x = \cos x \tag{2.24}$$

$$\frac{\partial}{\partial x} \exp(x) = \exp(x) \qquad \frac{\partial}{\partial x} \cos x = -\sin x \tag{2.25}$$

$$\frac{\partial}{\partial x} \log(x) = \frac{1}{x} \qquad \frac{\partial}{\partial x} \tan x = \frac{1}{\cos^2 x} \tag{2.26}$$

$$\frac{\partial}{\partial x} a\, x = a \qquad \frac{\partial}{\partial x} a^x = \log(a) a^x \tag{2.27}$$

$$\frac{\partial}{\partial x} f(x)\, g(x) = f(x) \frac{\partial}{\partial x} g(x) + g(x) \frac{\partial}{\partial x} f(x) \qquad \frac{\partial}{\partial x} \frac{f(x)}{g(x)} = \frac{-f(x) \frac{\partial}{\partial x} g(x) + g(x) \frac{\partial}{\partial x} f(x)}{g^2(x)} \tag{2.28}$$

2.4.2 Chain rule of differentiation

Perhaps the most useful trick in the book when it comes to differentiation is the chain rule. It allows us to compute the derivative of very complex functions analytically, and we will see many examples where it comes in handy. As always, practice makes perfect, and this is one rule that is worth getting the hang of.

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \frac{\partial g(x)}{\partial x} \tag{2.29}$$

As an example, let's prove Eq. (2.27) from the other basic rules, as follows:

$$\begin{aligned}
\frac{\partial}{\partial x} a^x &= \frac{\partial}{\partial x} \exp(\log(a^x)) && (2.30)\\
&= \frac{\partial}{\partial x} \exp(x \log a) && (2.31)\\
&= \frac{\partial}{\partial x \log a} \exp(x \log a) \frac{\partial}{\partial x} x \log a && (2.32)\\
&= \exp(x \log a) \frac{\partial}{\partial x} x \log a && (2.33)\\
&= a^x \log a && (2.34)
\end{aligned}$$

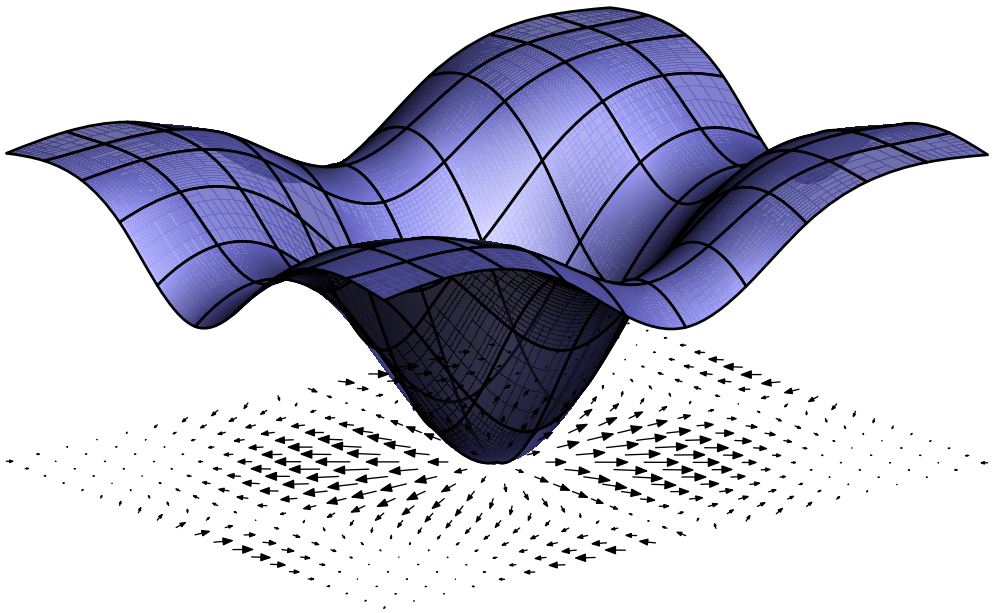


Figure 2.6: Illustration of the gradient of the function $f(x_1, x_2) = -(\cos^2 x_1 + \cos^2 x_2)^2$. The corresponding gradient is depicted as a vector field in the ground plane. Arrows point towards locations in the input space for which the function value is higher.

2.5 Gradients

Consider a scalar function f of multiple variables $x_1, \ldots, x_N$, denoted $f(\mathbf{x})$. The gradient of the function (with respect to $\mathbf{x}$) is then defined as follows:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_N} \end{bmatrix} \tag{2.35}$$

Each dimension of the gradient (indexed between 1 and N in our case) indicates the rate of increase of the function along the corresponding dimension of $\mathbf{x}$, and the resulting vector indicates the direction of steepest increase of the function. This is illustrated as in Figure 2.6. It bears emphasising that the gradient is a property of the input space. We are used to plotting functions as an extra dimension, but this dimension is no part of the (input) space: conceptually functions are also properties of the space. It is, therefore, more intuitive to show functions as a colour coding of the input space rather than as an extra dimension, and we will sometimes do so, but the disadvantage of that is that it gives a less precise idea of the function's shape. All of this may seem trivial (or confusing, perhaps) now, but do bear it in mind, as this is a common source of confusion when talking about Lagrange multipliers.

2.5.1 Function optimisation

As a consequence of this geometric interpretation, gradients are very useful for function optimisation. The function will be optimal (*i.e.*, either maximal or minimal) where the gradient equals the zero vector. If we can solve the set of equations $\nabla f(\mathbf{x}) = \mathbf{0}$, then we have then found the optimum. Often, however, solving this set will not be possible yet computing the gradient for any particular value of $\mathbf{x}$ will be possible. In that case, we can maximise the function by starting at any position in the input space, and moving a certain step size along the gradient, until its magnitude becomes too small. Clearly, to minimise the function, the same procedure can be used with the negative gradient. Such an iterative procedure allows us to find a (local) optimum for the function. Notice that the start point affects the result if the function has local minima.

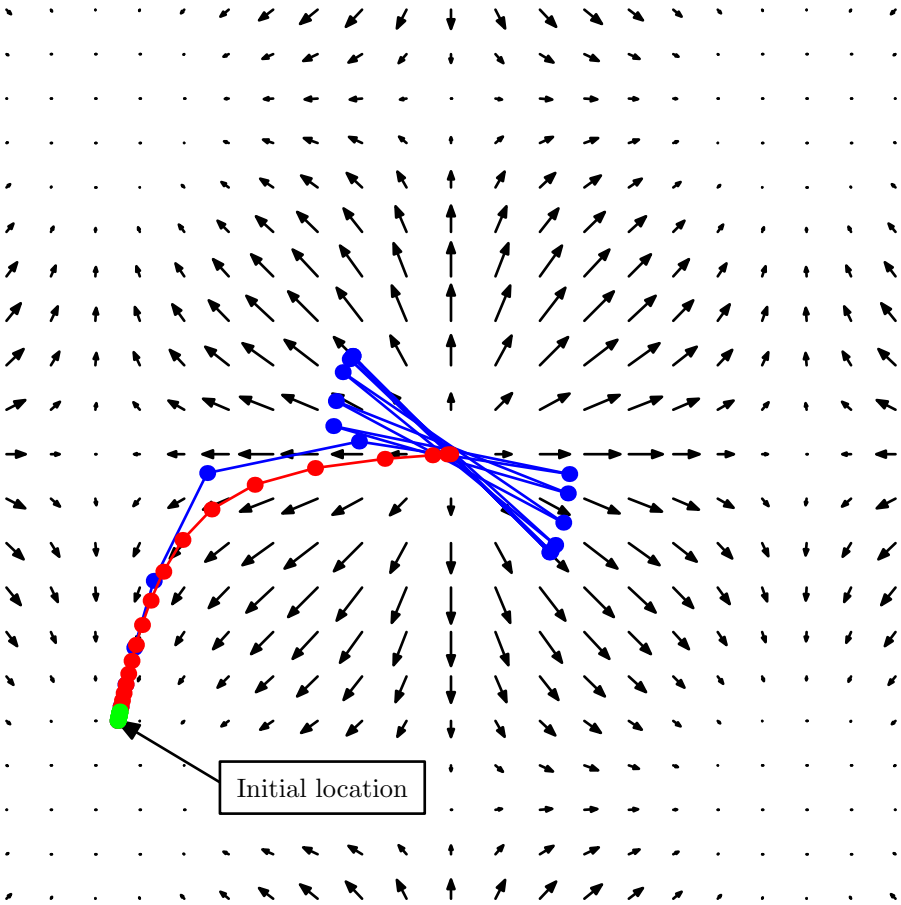


Figure 2.7: Illustration of gradient descent, for the same function as depicted in Figure 2.6. See the text for details. This figure shows 20 iterations of the algorithm, starting at the location indicated by the arrow and with three different step sizes. The green points show the progress with too small a step size, the blue points with too large a step size, while the red points show an example that converges correctly.

Running the algorithm multiple times with different start positions and keeping the best solution can help. Also notice that choosing a correct step size is important. Figure 2.7 illustrates three different choices of stepsize, starting from the same position. If the step size is too small (green), the algorithm takes forever (*i.e.*, too many iterations to be of useful value) to converge. If the stepsize is too large, the algorithm may reach the neighbourhood of the goal very fast, only to overshoot it and oscillate around the optimum (blue). The red line shows a good choice for the step size.

There are many improvements on the basic gradient descent algorithm, including adaptive step sizes (which address the problem shown above) and the incorporation of terms simulating momentum (which can help escaping sub-optimal local solutions). In the case of strongly non-convex functions, gradient descent may still require many iterations and end up in a sub-optimal solution. Methods such as Newton–Ralphson methods or approximations of such methods, such as (L)BFGS [Liu and Nocedal, 1989], can then be used. These use the Hessian matrix, *i.e.*, the matrix of second derivatives, in addition to the gradient: this provides information about the curvature of the function, and can therefore result in more efficient optimisation.

Chapter 3

Training and Testing

The key aspect of machine learning is, of course, that we learn from some examples in order to perform some task on new examples. This may seem obvious, but it means that we cannot assume that if a machine performs well on the training data, it will also perform well on new data. As an extreme example, one strategy which would not work is the following: store all the training examples and check, for each new datapoint, whether we have already seen it. If you know it, return the correct answer from the training data, otherwise give a random answer. The problem is, of course, that the probability that we will ever see the same datapoints again approaches zero for any realistic problem. Yet it is important to realise that the above-mentioned strategy is the only one to give you an unbiased machine. It is also completely useless.

Instead, we need to find some function that performs well on unseen data, and learn it from training data. Notice how many functions would perform perfectly on the training data — infinitely many. It is therefore necessary to limit our solution space before we even begin learning.

3.1 A first look at classification

Suppose we want to create a machine that can distinguish between different varieties of sharks, based on two measurements that are available to us, *viz.*, their length and their speed in the water. We are given a set of measurements, for which we know the desired class. The (normalised) training data might look as depicted in Figure 3.1. How could we use this for classification? One very simple, yet very powerful, method is to do the following: when given a new data point, find the closest data point in the training set and give the new datapoint the same label. The resulting classification is depicted in Figure 3.2. Notice how datapoints misclassifications will occur due to noise in the training dataset. One would probably want $\mathbf{x}_1$ to be classified as green, and $\mathbf{x}_2$ to be classified as red. This is a very general problem that occurs frequently in machine learning, and is called overfitting. We shall investigate this in depth with an example of regression, in section 3.2, but in effect the problem is that our machine does not distinguish between the variations in the data that are due to the process and the variations that are due to the noise. For now, let us mention that we could improve the performance of our classifier by looking at more than one neighbour: we can find the k nearest neighbours and use voting to assign the class label. This technique is therefore called k-nearest-neighbours or kNN.

This very simple solution to our problem is very effective, sometimes surprisingly so, and will serve to introduce some more general issues that we encounter more generally in machine learning. We will spend some time analysing it, and it will serve to introduce measures of performance for classification. In general, keep this technique in mind when doing machine learning: it often provides a powerful baseline to compare more complicated techniques to.

Nevertheless, kNN has disadvantages which sometimes prevent it from being used in practical implementations.

1. The first problem is the space requirement. We must store and keep the complete training dataset in order to perform classification.

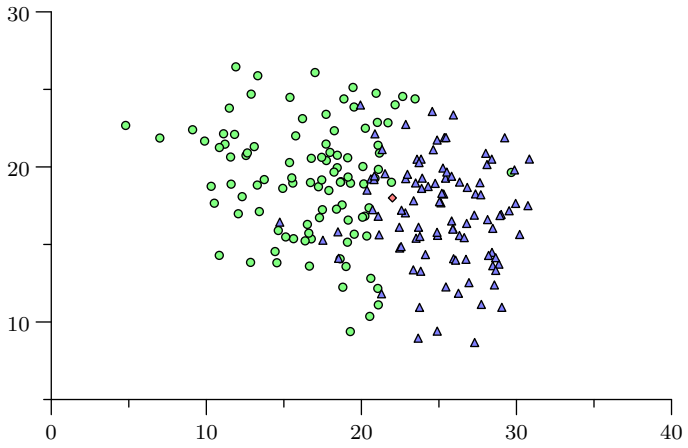


Figure 3.1: For a first attempt at classification, consider the dataset depicted here. We have two classes, the green and the blue class. How should we classify the new datapoint indicated by the red square?

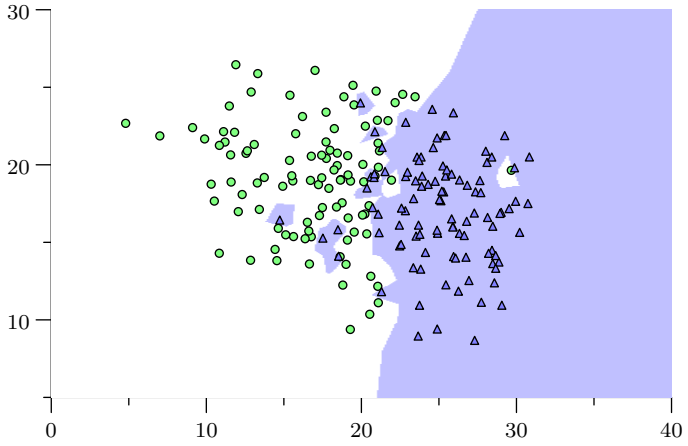


Figure 3.2: Depiction of how new datapoints would be classified with a 1-nearest-neighbour classifier: new datapoints falling in the areas shaded green would be classified as class 1, datapoints in the unshaded areas would be classified as class 2.

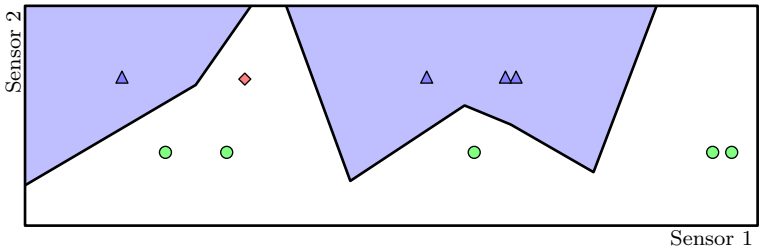


Figure 3.3: Illustration of the problems stemming from the use of Euclidean distance in kNN. In this example, the red square represents a new datapoint that we want to classify. The data is two-dimensional, but one dimension (sensor 1) is uninformative, while the other dimension (sensor 2) is a perfect predictor for the class label. The discriminant that we obtain when using 1-NN is depicted: every datapoint falling in the shaded area would be classified as belonging to the 'blue triangle' class. As we can see, our classifier misclassifies the datapoint, even though we have a fully informative sensor reading.

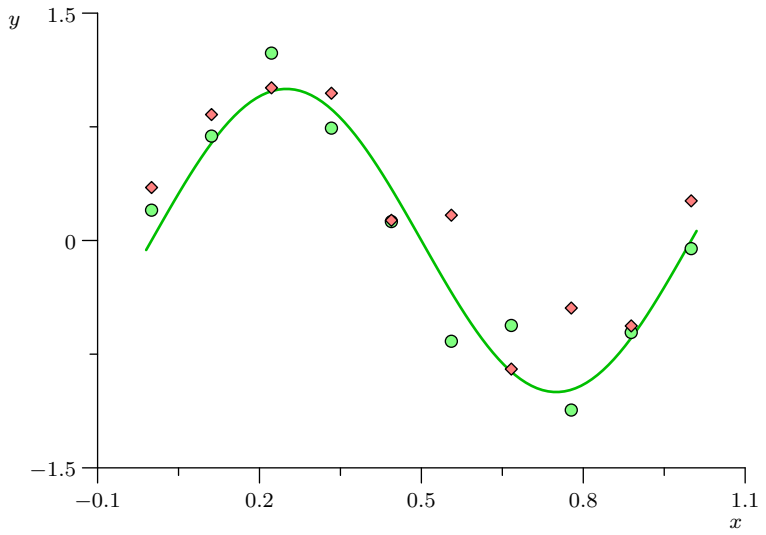


Figure 3.4: Curve-fitting example. The green line shows the process we are trying to learn, given by $\sin(2\pi x)$. We do not observe this process directly, but rather get observations which are corrrupted by noise with a Gaussian distribution. The green circles indicate data used for training, while the red squares indicate test data.

2. More importantly, a significant computational cost is involved: for each new datapoint, we need to search through the complete training set in order to find the nearest neighbours. This operation can be sped up somewhat through smart partitioning of the data space, but is nevertheless expensive compared to other techniques. In general, we do not really care about the computational cost of the training phase, but we do want the machine to be fast during operation. kNN does the oposite; it is fast during training and slow in operation.

3. A last problem with kNN is that it does not take the structure of the data space into account. To illustrate this point, examine Figure 3.3. In this figure, we depict data for which we have two measurements: sensor 1 and sensor 2. Of these, only sensor 2 is informative; sensor 1 is irrelevant. However, if we use kNN to classify a new datapoint, denoted by a black dot, we get a misclassification. Please notice that this problem is related to, but distinct from the problem depicted in Figure 3.2: the misclassification is due to the use of an inappropriate distance measure, not to noise in the data. Using multiple neighbours will modify the discriminant slightly, but will in general not solve the problem. We can still think of this as a form of overfitting (we're overfitting on sensor 1, since we're extracting information where there is none), but that's beside the point and actually even counterproductive. The real point here is that we can learn from the training data that sensor 1 is noisy and sensor 2 is not. A *more* flexible model can therefore learn to disregard sensor 1, but kNN cannot.

3.2 Polynomial curve fitting

Let us consider a curve-fitting example to illustrate the choices and difficulties involved in learning from data. Consider the data shown in Figure 3.4, consisting of N (input,target) pairs, denoted (x, t). This data was generating by computing $\sin(2\pi x)$ in the range [0..1], and measuring it at uniform intervals. The measurement, however, is corrupted by Gaussian noise. In practice, of course, we will not know what process generated the data, nor will we know the amount of noise in the measurements. It is illustrative, however, to consider this artificial example in order to gain some understanding of the issues we are dealing with.

Our first restriction of our search space comes in right here. We want to learn the process that generated this data, but we do not know what class of functions it belongs to. In this example, we will start by looking at polynomial functions. These are functions of the form $y(x, w_0, \ldots, w_m) = w_0 + w_1 x + w_2 x^2 + \cdots + w_m x^m$, and we need to learn the weights $w_0, \ldots, w_m$ from the training data. We have $m + 1$ unknown weights, so

we will need at least $N = m + 1$ datapoints. We can the denote predictive function as follows using matrix notation:

$$y(x, \mathbf{w}) = \boldsymbol{\phi}^\top \mathbf{w}, \tag{3.1}$$

where $\boldsymbol{\phi}$ is a vector containing the relevant powers of x and $\mathbf{w}$ is a vector of the corresponding weights:

$$\boldsymbol{\phi} = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^m \end{bmatrix} \qquad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}. \tag{3.2}$$

We need to learn the value of the weights $\mathbf{w}$ in order to have a representation of what the generating process is like: the best values for the weights are those that approximate the generating process most closely. Notice that $\boldsymbol{\phi}$ is a fixed set of transformations of the datapoint; the resulting approximation $y(x, \mathbf{w})$ is a weighted linear combination of these (non-linear) functions. This makes finding the weights simple — at least, it's much simpler than if the function had been a non-linear function of the weights.

For this first example, let us say that the best weight values, denoted $\mathbf{w}^*$, are the values that result in the smallest sum–squared error (E_{SSE}):

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} E_{SSE}(\mathbf{y}(\mathbf{x}, \mathbf{w}), \mathbf{t}) \tag{3.3}$$

$$= \arg\min_{\mathbf{w}} \sum_{i=0}^{N} (y(x_i, \mathbf{w}) - t_i)^2 \tag{3.4}$$

Again, we can write this in matrix notation by introducing the so-called design matrix $\boldsymbol{\Phi}$, where each row consists of the features extracted from the corresponding datapoints, and the vector $\mathbf{t}$, which consists of the corresponding targets:

$$\boldsymbol{\Phi} = \begin{bmatrix} 1 & x_1 & (x_1)^2 & \cdots & (x_1)^m \\ 1 & x_2 & (x_2)^2 & \cdots & (x_2)^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & (x_N)^2 & \cdots & (x_N)^m \end{bmatrix} \qquad \mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_N \end{bmatrix} \tag{3.5}$$

The vector of the predicted values for the given datapoints, $\mathbf{y}$ can then be written as

$$\mathbf{y} = \begin{bmatrix} y(x_1, \mathbf{w}) \\ y(x_2, \mathbf{w}) \\ \vdots \\ y(x_N, \mathbf{w}) \end{bmatrix} = \boldsymbol{\Phi}\mathbf{w}, \tag{3.6}$$

and the SSE can be written as

$$E_{SSE}(\mathbf{w}) = (\mathbf{t} - \mathbf{y})^\top (\mathbf{t} - \mathbf{y}) \tag{3.7}$$

$$= (\mathbf{t} - \boldsymbol{\Phi}\mathbf{w})^\top (\mathbf{t} - \boldsymbol{\Phi}\mathbf{w}) \tag{3.8}$$

So, how can we find the value of $\mathbf{w}$ that minimises $E_{SSE}(\mathbf{w})$? To find the optimum of $E_{SSE}(\mathbf{w})$, we take the first derivative with respect to the free parameters, and set it equal to zero:

$$\frac{\partial}{\partial \mathbf{w}} (\mathbf{t} - \boldsymbol{\Phi}\mathbf{w})^\top (\mathbf{t} - \boldsymbol{\Phi}\mathbf{w}) = 0 \tag{3.9}$$

In *The matrix cookbook* [Petersen and Pedersen, 2006], we find that the derivative is given by $-2\mathbf{\Phi}^\top(\mathbf{t}-\mathbf{\Phi}\mathbf{w})$, and so we get:

$$-2\mathbf{\Phi}^\top(\mathbf{t}-\mathbf{\Phi}\mathbf{w}) = 0 \tag{3.10}$$

$$\mathbf{\Phi}^\top\mathbf{t} - \mathbf{\Phi}^\top\mathbf{\Phi}\mathbf{w} = 0 \tag{3.11}$$

$$\mathbf{\Phi}^\top\mathbf{\Phi}\mathbf{w} = \mathbf{\Phi}^\top\mathbf{t} \tag{3.12}$$

$$\mathbf{w} = (\mathbf{\Phi}^\top\mathbf{\Phi})^{-1}\mathbf{\Phi}^\top\mathbf{t} \tag{3.13}$$

This may look a bit daunting at first, but it's actually simple: we are given the design matrix $\mathbf{\Phi}$ and the vector of targets $\mathbf{t}$, and we know algorithms to compute matrix products and matrix inverses. Computing the vector $\mathbf{w}$ is therefore just a straightforward application of these algorithms. Alternatively, we can stop at Eq. (3.12), which represents a system of N equations with m unknowns, where N is the number of datapoints and m is the order of the polynomial: we have stable algorithms to solve these.

3.3 Overfitting and Generalisation

Now that we are equipped to fit a polynomial function to our training data, we have one more choice to make: what should be the order of that polynomial? If we choose a low order polynomial, the function will not fit the data very well. As we can see in Figure 3.4, we are given ten datapoints. The best we can do, therefore, is to choose a ninth-order polynomial: then we must find the value of ten parameters, $w_0 \dots w_{10}$, and we have a system of ten equations with ten unknowns, which we can solve exactly.

However, we are not really interested in how well the system performs on the training data. Instead, we're interested in how well it will perform on future data. Figure 3.5 shows what happens as the order of the polynomial is increased,[1] and shows how this affects both these errors: the error on the training dataset — the training error — and the error on some other data generated from the same distribution — the test error.

The best thing a zeroth-order polynomial can do, is to predict the mean of the data. It is independent of the input x, and so the solution that minimises the error is the mean of the output. This does not fit the generating process very well, and hence both the training error and the test error are pretty high. If $m = 1$, we get a linear function, which is better: both the training and test error are reduced considerably. Increasing this to $m = 2$ does not do much, but $m = 3$ is much better. This actually looks more like our generating process, and both the training and testing errors are quite low. Increasing m further does not buy us much, until we reach $m = 9$, where the polynomial function is capable of fitting all training datapoints perfectly and our training error drops to zero. In order to do so, however, the fitted function needs to be very 'squiggly': it fits the training data perfectly, but looks nothing like the generating process. The error on the test dataset shoots through the roof.

The lower-order polynomials illustrate so-called underfitting (also called oversmoothing): our machine is not flexible enough to approximate the generating process, and is not capable of capturing all the relevant information in the training data. Around $m = 3$ we obtain what is called good generalisation: the machine captures the generating process well and performs well, both on the training data and on previously unseen data. As we get to higher order polynomials, especially at $m = 9$, we get a severe case of overfitting. The machine can fit the training data perfectly, including the noise that's present in it. The resulting fit has zero training error, but performs very badly on test data.

3.3.1 What affects overfitting

These are very general concepts, which are ubiquitous to machine learning. They occur in regression as well as in classification, and a major aspect of creating effective learning machines is to be able to assess and maximise generalisation. From the above example, we can see that multiple things affect how much a machine overfits:

[1] This figure is actually an animation. The weird icons at the bottom of the figure are clickable controls that you can use to step through the animation, play it, or vary the speed at which it is played. You need acrobat reader version 8 or above in order to see the animation.

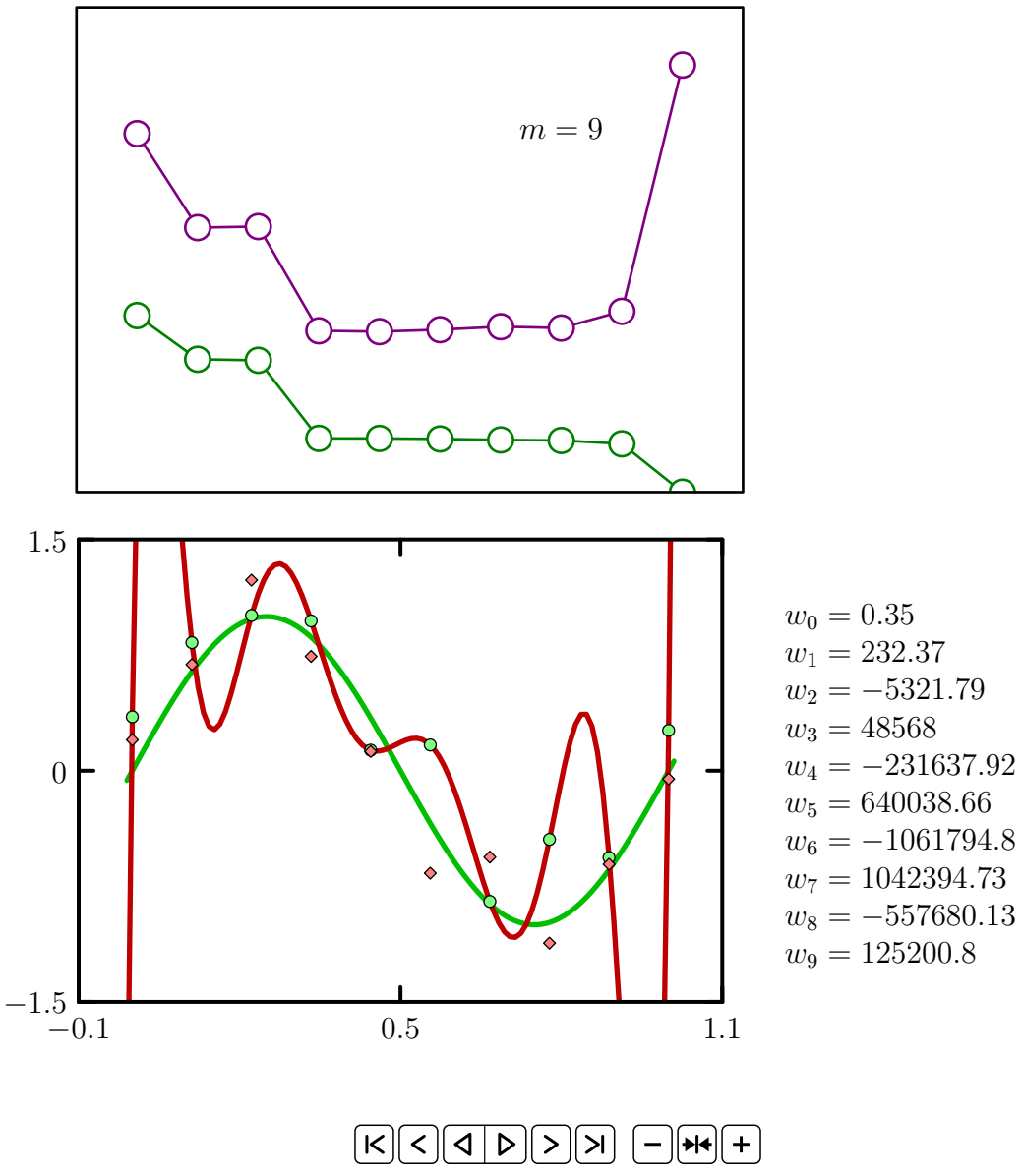


Figure 3.5: Animation illustrating overfitting. As the order of the polynomial increases, the machine can model ever more complex functions. When the number of parameters reaches the number of datapoints in the training set, we can find parameters that fit the training data exactly, but at the cost of increased error on the test set.

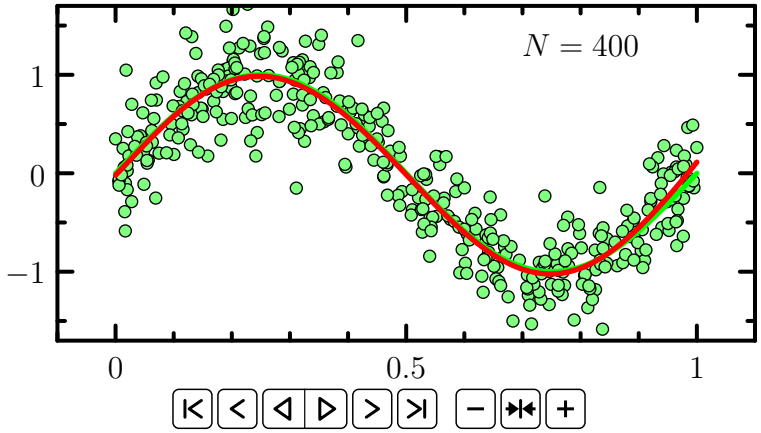


Figure 3.6: Illustration of how overfitting is reduced as the amount of data is increased. In this example, a ninth-order polynomial is fitted, without parameter penalisation, to training sets of increasing size. As the size of the training set is increased, the resulting function becomes more similar to the original process.

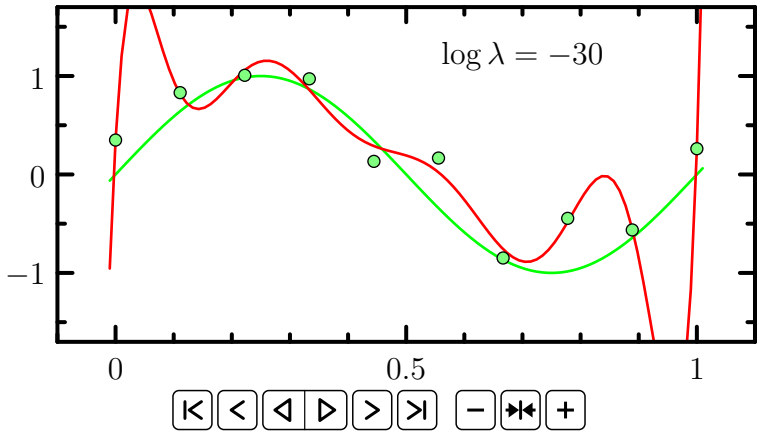


Figure 3.7: Illustration of how the smoothness of the function varies as a function of the penalisation factor λ. The machine overfits heavily when lambda is very small, and underfits when lambda becomes too large. A validation set can be used to select a correct value for λ.

Machine complexity The more flexible and complex a machine is, the more likely it is to overfit. Too simple a machine will not learn the underlying process, and too complex a machine will fit the noise. One way to avoid overfitting is therefore to select a machine of the right complexity for the given problem and the given training data.

Amount of training data In the case of the above example, one must realise that a sine wave could only be fitted exactly using a polynomial function of infinite order. However, in order to find the correct parameters for such a polynomial, we would require infinite amounts of training data (not to mention infinite computing resources, or infinite amounts of time.) In practice, we can use more flexible machines to approximate the underlying process as we get more training data. This is illustrated in Figure 3.6, where a ninth-order polynomial is trained with different training set sizes. As more data is available for smoothing, the overfitting becomes less severe.

There is, unfortunately, no direct relation between the optimal complexity of the model and the amount of training data, as it depends on how noisy the observations are (which we do not know.) However, there are criteria that allow us to judge whether the model complexity is suitable for the available training set size, such as the Akaike Information Criterion and the Bayesian Information Criterion.

Parameter values Even though a ninth-order polynomial may potentially result in very complex functions, if all the parameter values were zero, we'd end up with a straight line at $y = 0$. In general, smaller parameter values result in smoother functions, and a high-order polynomial may approximate the generating process quite well, even with little training data, if we penalise large parameter values.

This is illustrated in Figure 3.7. In this case, we use a slightly more complex error function: instead of minimising the SSE, we minimise a *penalised* sum-squared error. This cost function is as follows:

$$C(\mathbf{w}) = E_{SSE}(\mathbf{w}) + \lambda \mathbf{w}^\top \mathbf{w}, \tag{3.14}$$

i.e., we minimise a weighted sum of the SSE and the sum of the squared weight values. Here, λ is an externally set parameter to trade off the effect of the parameter penalisation and the fit to the data. If $\lambda = 0$, we obtain the original polynomial function fitting seen before. As $\lambda \to \infty$, the effect of the data becomes negligible. We can minimise this new error function in the same way as before: take the gradient with respect to $\mathbf{w}$ and set it equal to zero:

$$\frac{\partial}{\partial \mathbf{w}} C(\mathbf{w}) = 0 \tag{3.15}$$

$$-2\boldsymbol{\Phi}^\top(\mathbf{t} - \boldsymbol{\Phi}\mathbf{w}) + 2\lambda\mathbf{w} = 0 \tag{3.16}$$

$$-\boldsymbol{\Phi}^\top\mathbf{t} + (\boldsymbol{\Phi}^\top\boldsymbol{\Phi} + \lambda\mathbf{I})\mathbf{w} = 0 \tag{3.17}$$

$$(\boldsymbol{\Phi}^\top\boldsymbol{\Phi} + \lambda\mathbf{I})\mathbf{w} = \boldsymbol{\Phi}^\top\mathbf{t} \tag{3.18}$$

3.3.2 Avoiding overfitting

The most straightforward way to avoid overfitting, is by limiting the model's complexity. Since we do not have access to unlimited amounts of training data, the best machine that we will come up with, based on the training data, will never be optimal on future data. By limiting the complexity of our machine, we can, therefore, obtain results that perform well on future data. The question is then: how complex should our machine be? We cannot use the training data to select the machine's complexity, because the most complex machine will always tend to fit the training data best.

One thing we could do, is to split the data we have access to into two part: one part for training, and one part for evaluating whether we are overfitting. However, there are a few things we have to be careful about:

1. If we select a machine based on its performance on the data that wasn't used for training, then that's in itself a form of training. If we had chosen a different partitioning of the data, another machine might have turned out to look better. The machine's performance on this data is, therefore, not an unbiased estimate of its true performance. One possible solution to this problem is to split the available data into three datasets:

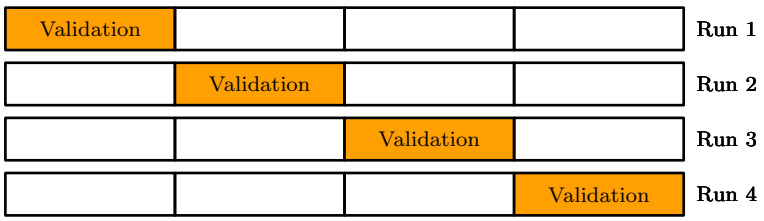


Figure 3.8: Illustration of k-fold cross-validation, for $k = 4$.

Training set This is the set of datapoints that we rely on to learn the model parameters

Validation set This set of datapoints allows us to select the model complexity or choose "hyperparameters" such as the parameter penalisation weight λ

Test set The set of datapoints that is used to evaluate the machine's performance on unseen data. These datapoints must not have been used in any way during the training phase, lest we obtain a biased (and overly optimistic) estimate of the performance.

2. Splitting the available data into different sets, and only using part of it for learning is wasteful. It is often the case that we have access to very limited amounts of training data, and discarding parts of it can have a severe effect on the performance. One solution is to make the validation set very small, so as to keep as much data as possible for training. The problem is then, of course, that the evaluation on the validation set is noisy. This problem can be alleviated by using cross-validation: train the machine k times on different splits (or "folds") of the dataset, and test on the rest. This is illustrated in Figure 3.8, where we split the data into 4 sets, 3 of which are used for training and one of which is used for validation. When we have access to extremely limited amounts of data, it makes sense to use as many folds as we have datapoints: this particular form of cross-validation is called leave-one-out cross-validation.

   Notice that cross-validation increases the amount of information that we can obtain from our dataset, but that it carries an important computational cost. In practice, the amount of available data will dictate how many folds we will use.

3. There is something inherently unsatisfying about adjusting the complexity of our machine to the amount of available data. In reality, we would like our machine to be complex enough to be able to represent the structure of the problem, independently of how much training data we have. We will come back to this point when looking at a Bayesian approach to probabilistic modelling of the problem.

Why simplicity?

We have seen that we can avoid overfitting by reducing a machine's complexity. This idea, which is often referred to as *Occam's razor*, is appealing and may make intuitive sense, but one would be warranted to wonder why it is, really, that simplicity (rather than anything else) should be the objective to pursue.

There are different schools of thought on this, but they all basically converge to the same idea. I will try to describe them briefly below, and then attempt to illustrate how they converge to the same thing.

No free lunch (NFL) The "no free lunch" idea basically says that we need some bias if we want to learn anything about things that we haven't observed directly. If any function is possible, any prediction is possible anywhere except at the given training datapoints. A bias is necessary for generalisation to be at all possible (this is called an "*inductive bias*"), but this idea does not indicate why any bias should be better than another.

However, Figure 3.9 illustrates the difference of fitting a first order polynomial ($f(x) = ax + b$) and a second order polynomial ($f(x) = ax^2 + bx + c$) based on the two given training points. The shaded area indicates all possible predictions, for all functions that fit the training data. As we can see, allowing more complex, flexible functions very soon becomes equivalent to having no bias at all. In this extreme example, we assume that the observations have no noise at all, and assume that all functions that fit our training data are equally probable.

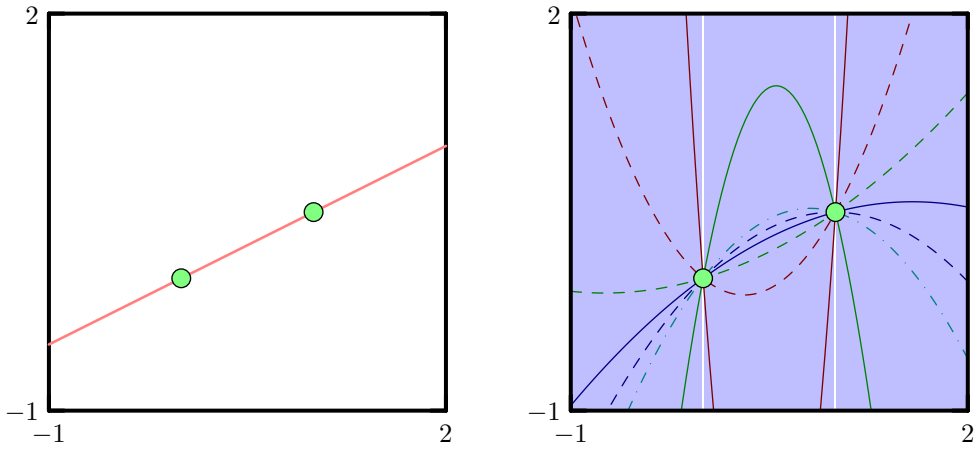


Figure 3.9: Illustration of inductive bias. In this example, we observe two datapoints, which we know to be noiseless, and attempt to recover the function that produced this data. Figure (a) illustrates the recovered first-order polynomial; Figure (b) indicates the possible results if a second order polynomial is used. In this figure, a few functions are sampled that fit the data. There are, of course, uncountably infinitely many quadratic functions that fit the data, and if they were all taken into account, the area of possible prediction would be the shaded area: for any input value any output is possible, except for the precise input values present in the training data, where only the corresponding output is possible.

Minimum description length (MDL) The idea of MDL is to use the model to compress the data. Since the model can be used for prediction, we do not need to store the target values: it is sufficient to store the model and the input values; the target values can be computed by the model. Of course, unless the model is perfect, the computed predictions will contain errors: so for lossless compression we need to store the difference between the actual target and the predicted target. However if the model performs better than random, the prediction errors will be smaller than the targets, and will therefore require fewer bits to encode.

The intuition is then that a model that provides the most compact representation is best. As a consequence, we have an immediate trade-off between the model complexity (we need to store the model itself, so we want to limit the number of bits needed to store that) and the prediction errors (the better the predictions, the fewer bits are needed to store the errors).

Bayesian approach If we simply follow the axioms of probabilities to their logical conclusion, then we should not limit complexity at all: all models that fit the data are possible, it's just that some models are more likely than others: our prediction should therefore take all possible models into account, each model weighted by its posterior probability.

If we analyse this probabilistic approach more closely, it becomes clear why, if we *don't* do the "right" thing but approximate it by choosing just one model, then choosing a simple model rather than a complex model is actually not a bad thing to do: the set of complex models *also* contains simple models, and many complex models will have a "core" of a single simple model in common. If we then sum over all models and weigh the models with their posterior probability, then each of the complex models gets a low probability and so do the predictions of those models, but the common predictions that correspond to the simple model are repeated more often and get a higher probability. To illustrate this, notice how i n Figure 3.9, the linear function is present in the class of quadratic function as a special case.

3.3.3 Sidetrack: Overfitting in Pigeons

In a famous experiment from 1948 (Skinner, 1948), hungry pigeons were briefly and automatically presented with food, at regular intervals. There was no relationship between the animal's behaviour and the fact that food was given, yet the following pattern would typically emerge: the pigeon would be doing something random, such as flapping a wing or turning its head when the food would be given. The pigeon would then repeat that action in the hope of calling the food back, and if the delay between the presentations of food is not too long, the odds are that it is still doing whatever it believes triggers the coming of food the next time that the food comes, so that the association is reinforced. In some cases, the birds would develop very complex dances in the hope of getting more food.
This is a simple case of massive overfitting. There is no correlation between the two variables ("behaviour" and "food"), but the pigeon massively overfits on the first datapoint. After that, because of the original overfitting, the training set contains many instances where the two variables co-occur and in this observed set there is a strong correlation between the two variables. The collected dataset is then biased, because the observed datapoints are neither independent nor identically distributed, which in turn leads to more overfitting.
The delay between presentations of food affects how independent the observations are, and with long delays this reinforcement pattern does indeed disappear.

3.4 Nearest Neighbours, revisited

To avoid overfitting with our nearest neighbours classifiers, we can do a number of things. The most commonly used solution is to look at multiple neighbours before deciding on the class prediction. For example, one might look at the $k = 3$ nearest neighbours of a new datapoint, and use voting to decide on the class label.[2] Such a classifier is illustrated in Figure 3.10. Similar to our regression example of section 3.2, large values of k lead to oversmoothing: the classifier loses its ability to find important structure in the data. When using kNN, people will, therefore, generally use a validation set to determine k, the number of neighbours to look at.

Another solution is to limit the number of prototypes or training data points. Looking back at Figure 3.2, it is clear that the vast majority of the prototypes do not affect the classification outcome of a 1-nearest-neighbours classifier. Moreover, we can identify some of the prototypes as being clearly responsible for overfitting, so that we can limit overfitting by reducing the number of prototypes. Finally, reducing the number of prototypes also reduces the computational complexity of a new classification. This can be an important consideration in some cases. Different techniques have therefore been proposed to select a limited number of prototypes when constructing the classifier.

3.5 Measuring Performance

The most straightforward measure of a classifier's performance is its accuracy. This is defined as the number of correctly classified datapoints over the total number of datapoints.

$$\text{Accuracy} = \frac{\#_{\text{correct classifications}}}{\#_{\text{datapoints}}} \tag{3.19}$$

Equivalently, the error rate of the classifier can be used, which is defined as $1 - \text{Accuracy}$. In the case of two-class classification, we often consider that the datapoints belong to a positive and a negative class. It is then customary to distinguish between true positives (TP, the number of datapoints from the positive

[2] Notice that, to avoid undecided votes, one should use odd values for k

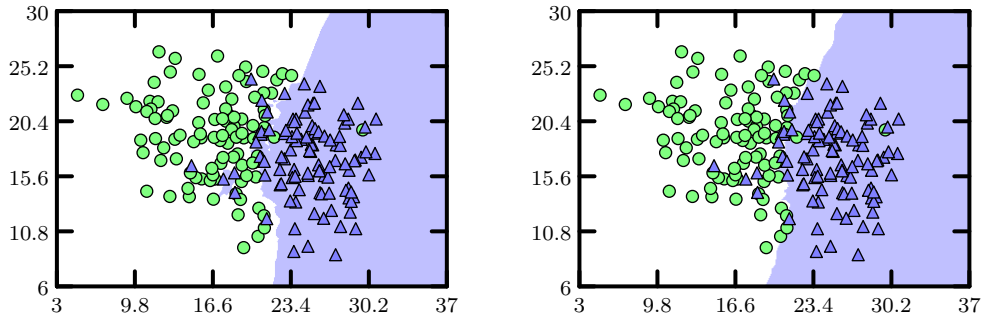


Figure 3.10: Example of a three-nearest-neighbours classifier (a) and a 19-nearest-neighbour classifier (b) applied to the same data as in Figure 3.2. Notice how the 3-NN classifier is less prone to overfitting than the 1-NN classifier depicted in that figure, while the 19-NN classifier discards important structure in the data (this can be verified with a validation set, but is also visible in the lower region of the plot)

class that are correctly classified), false positives (FP, the number of datapoints from the negative class that are wrongly classified as positive), true negatives (TN) and false negatives (FN). The accuracy can then be written as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.20}$$

However, these measures do not convey the complete picture: the classifier may make many more mistakes confusing one class for the other than the other way around. Such information can be captured in a confusion matrix. For the two-class problem, the confusion matrix looks as follows:

$$C = \begin{bmatrix} TP & FN \\ FP & TN \end{bmatrix}. \tag{3.21}$$

The rows correspond to the actual class, and the columns correspond to the output of the classifier. From this, the following measures are often extracted, which are used more or less frequently depending on the particular discipline people are working in:

$$\text{precision} = \frac{TP}{TP + FP} \tag{3.22}$$

$$\text{recall} = \text{sensitivity} = \frac{TP}{TP + FN} \tag{3.23}$$

$$\text{specificity} = \frac{TN}{FP + TN} \tag{3.24}$$

For multiple classes, this is extended in the straightforward way: for m classes, we have

$$C = \begin{bmatrix} n_{11} & \cdots & n_{1m} \\ \vdots & \ddots & \vdots \\ n_{m1} & \cdots & n_{mm} \end{bmatrix} \tag{3.25}$$

where, again, n_{ij} represents the number of datapoints from class $\mathcal{C}_i$ that have been classified as belonging to class $\mathcal{C}_j$. The accuracy is, therefore, found by summing the elements on the diagonal and dividing by the total number of elements.

3.6 Reject option

Sometimes we know that we are not very confident in our classification. In such cases, and if misclassification carries a large cost, it may be better to reject the datapoint and let some other instance (such as a human expert) deal with it.

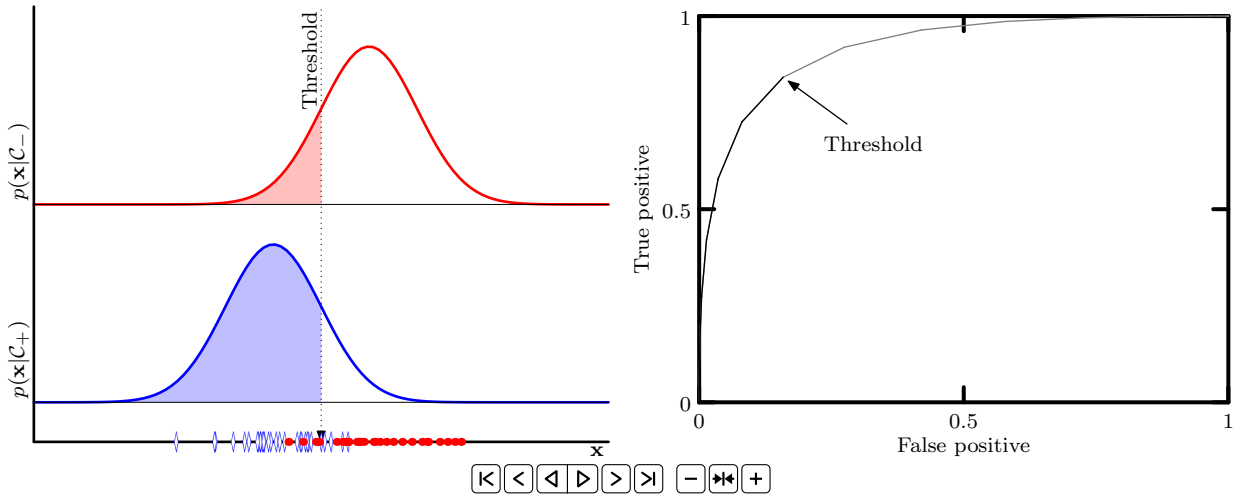


Figure 3.11: Animated illustration of the computation of an ROC curve. As we shift our discriminant to increase the number of true positive classifications, we will also increase the number of false positives.

3.7 Receiver Operating Characteristic (ROC)

The accuracy of a classifier does not always give a very meaningful picture of how the classifier performs on a real data. It is often the case that some misclassifications are worse than others. For example, consider a machine predicting whether you have cancer from some blood analysis results: it would be dramatic if it missed that you have cancer, and it would be much less damaging if it predicted cancer for a healthy person. In order to have a more informative idea of a classifier's behaviour as the decision threshold is changed, we can plot a receiver operating characteristic (ROC) curve.

Figure 3.11 illustrates how such a figure is made. In this case, we have two classes: a positive class $\mathcal{C}^+$ and a negative class $\mathcal{C}^-$. Our measurements are one-dimensional, and are depicted in the x-axis of the left plot. The distribution of the data is depicted, per class, above the data. If we vary our decision threshold from $-\infty$ to ∞, the number of correctly classified positive examples (indicated by the green shaded area) increases. As we progress towards ∞, we also start wrongly classifying negative examples as belonging to the positive class, as indicated by the red shaded area. The ROC curve, shown in the right graph of Figure 3.11, shows the relationship between the number of true positive and false positive classifications.

3.7.1 Area Under the Curve (AUC)

It is clear from the example above that we would like the curve to come as close as possible to $(0, 1)$, where all positive examples are correctly classified as such, and no negative examples are wrongly classified as positive. A good measure of the performance of a classifier is therefore the area between the x-axis and the ROC curve: the larger this area, the better the classifier performs under different threshold values. This is shown in Figure 3.12. Here, we compare the area under the ROC curve for different class distributions, modelled by Gaussian distributions in our classifier. As the classes overlap more, the area under the curve decreases until it reaches one half when the classes are indistinguishable.

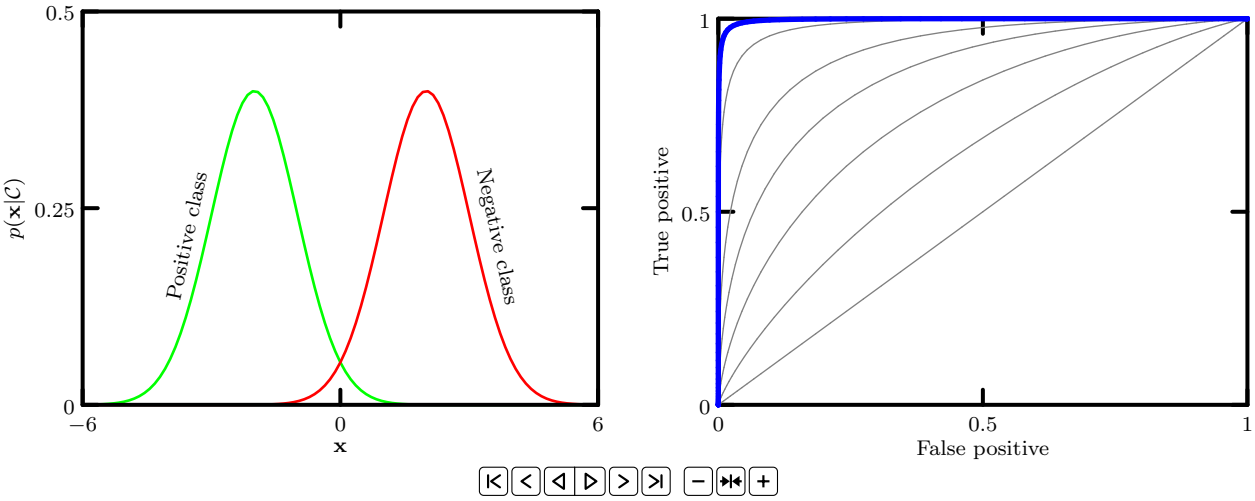


Figure 3.12: Illustration of how the area under the curve changes as the classes become less separable. Note that the AUC reflects how well the classifier can separate the classes, rather than how separable the classes are in general

Chapter 4

Linear discriminants

When doing classification, we are given some inputs that we want to assign a class label to. In effect, we want to compute some function f of our inputs $\mathbf{x}$ which returns the class label $\mathcal{C}$. Let us first look at a two-class problem, and see how we can generalise the result to more than two classes later.

Figure 4.1 depicts some training data from a two-dimensional, two-class classification problem. We want to find a binary function $y(\mathbf{x}) \in \{\mathcal{C}_1, \mathcal{C}_2\}$ that returns the correct class label for new data. We can construct such a function by first defining a continuous scalar function $y(\mathbf{x})$, and returning

$$f(\mathbf{x}) = \begin{cases} \mathcal{C}_1 & \text{if } y(\mathbf{x}) > 0 \\ \mathcal{C}_2 & \text{if } y(\mathbf{x}) < 0 \end{cases} \tag{4.1}$$

The discriminant of such a function is then defined as the set of points where the output of the function switches from one class to the other. In our case, this set of points is given by

$$y(\mathbf{x}) = 0 \ . \tag{4.2}$$

If this discriminant is given by a hyperplane in the input space (*i.e.*, a point in 1D, a straight line in 2D, a plane in 3D, *etc.*), it is called a linear discriminant. Classification functions with a linear discriminant are the simplest classifiers we can devise, and those are the ones we shall focus on in this chapter.

We can now express our learning process in terms of finding a suitable function $y(\mathbf{x})$ which is positive for all elements of class $\mathcal{C}_1$ in our training set, and negative for all training examples of class $\mathcal{C}_2$. Somehow, we would also like this function to be, in addition, positive for any novel examples of $\mathcal{C}_1$ and negative for novel examples of $\mathcal{C}_2$. Such a linear function is depicted in Figure 4.2.

How can we find such a function?

4.1 Linear perceptrons

Linear perceptrons are an early attempt at solving the problem of linear classification. At the time, they were inspired from the understanding of biological neurons. The idea was that each neuron has a few inputs (dendrites) and an output (axon).[1] The output would fire if the inputs were sufficiently stimulated. In linear perceptrons, the inputs are continuous values, and the output is zero if the perceptron does not fire, and one if it does. How much stimulation is required for the perceptron to fire is determined by the weights and the bias of the perceptron, as follows:

$$f(\mathbf{x}) = h(w_0 + \sum_{i=1}^{N} w_i x_i) \tag{4.3}$$

$$= h(\mathbf{x}^\top \mathbf{w}) \tag{4.4}$$

[1] While broadly correct, many different types of neural cells exist, and this is overly simplified compared to real neurons.

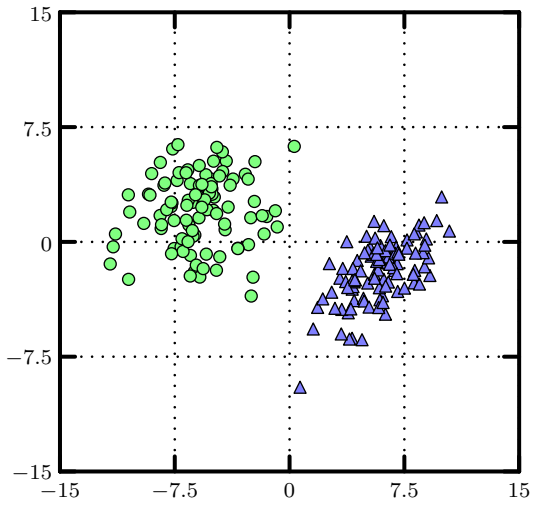


Figure 4.1: Example data for a two-class classification problem.

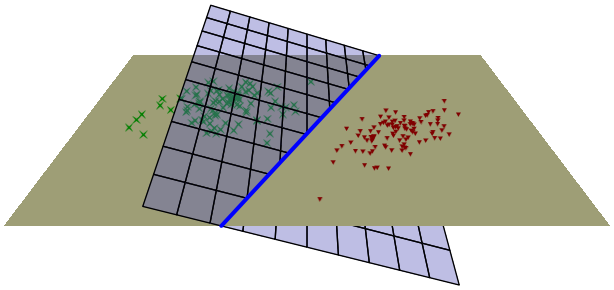


Figure 4.2: Illustration of a linear function that would result in correct classification of the complete training dataset. The corresponding discriminant is plotted as a blue line in the input space.

where $\mathbf{x} = (1, x_1, \ldots, x_n)$, and $h(\cdot)$ is the following step function:

$$h(x) = \begin{cases} +1 & \text{if } x \geqslant 0 \\ -1 & \text{otherwise.} \end{cases} \tag{4.5}$$

Linear perceptrons are trained by minimising the number of misclassified examples. However finding parameters $\mathbf{w}$ that minimise the number of misclassified datapoints is not trivial: indeed, since the error function is discontinuous (it increases in steps of plus or minus one wherever a change in $\mathbf{w}$ causes a datapoint to be classified differently), we cannot compute the gradient of the error function with respect to the parameters. Hence, we cannot perform regular gradient descent, and we certainly cannot find the optimal value of $\mathbf{w}$ analytically. We can nevertheless train perceptrons based on the perceptron criterion.

4.1.1 Perceptron learning algorithm

We would like to find a function $y(\mathbf{x}) = \mathbf{x}^\top \mathbf{w}$ which is larger than zero for all examples in class $\mathcal{C}^+$ and smaller than zero for the other examples. If we encode the target class label t_i (corresponding to the ith datapoint $\mathbf{x}_i$) as $\{+1, -1\}$ for $\mathcal{C}^+$ and $\mathcal{C}^-$ respectively, we would like all datapoints to satisfy

$$\mathbf{w}^\top \mathbf{x}_i \, t_i > 0 \; . \tag{4.6}$$

We can optimise the value of $\mathbf{w}$ using the perceptron criterion, which associates zero error with datapoints that are correctly classified, and a positive error with all misclassified datapoints. The interesting idea here is to make this error encode how far these datapoints are from the discriminant, rather than simply the fact that they are incorrecly clasified. To do this, we try to minimise

$$E_p(\mathbf{w}) = -\sum_{n \in \mathcal{M}} \mathbf{w}^\top \mathbf{x}_n \, t_n, \tag{4.7}$$

where $\mathcal{M}$ is the set of misclassified patterns. We can now apply stochastic gradient descent to minimise Eq. (4.9). The gradient of $E_p(\mathbf{w})$ with respect to $\mathbf{w}$ at a misclassified datapoint $\mathbf{x}_n$ is given by $-\mathbf{x}_n \, t_n$. We can, therefore, walk through the set of misclassified datapoints and modify the current $\mathbf{w}$ by subtracting a fixed step size times the gradient (subtracting, because we are minimising). We can choose the step size to be one, because the scale of $\mathbf{w}$ does not affect the location of the discriminant.[2] At each iteration of the algorithm, we therefore choose a misclassified point $\mathbf{x}_n$, and update $\mathbf{w}$ as

$$\mathbf{w}^{\text{next}} = \mathbf{w} + \mathbf{x}_n t_n \tag{4.8}$$

This result is pretty amazing, because it was felt at the time that such a simple update rule was biologically plausible. One could imagine a neuron, having received an input and produced the wrong output simply increasing its internal 'weights' with that same input. Since then, a lot has been learned about the workings of neurons and this particular model of neurological learning is not considered plausible anymore, but it was still a great step forward.

A few things to note about the perceptron, are:

- The perceptron learning algorithm is guaranteed to diminish the error contribution of the datapoint that is added to $\mathbf{w}$, but the change in $\mathbf{w}$ may, of course, lead to new points being misclassified. The global error is, therefore, not guaranteed to decrease at each iteration. Nevertheless, the perceptron convergence theorem proves that if a solution exists (*i.e.*, if the data is linearly separable), the perceptron learning algorithm is guaranteed to find it in a finite number of steps, although this number may still be arbitrarily large. In practice, therefore, if the algorithm does not seem to converge, we cannot tell whether this is due to the problem being non-linearly separable, or simply due to slow convergence.

- Even though the algorithm is guaranteed to find a solution if one exists, that particular solution is not guaranteed to be particularly good. Figure 4.3 shows different solutions where no datapoints are misclassified, and the perceptron learning algorithm would therefore quit. Some of these solutions are much more likely to perform well on future data than others, but the perceptron criterion cannot distinguish between them.

[2] The scale of $\mathbf{w}$ affects the 'angle' between the function and the input space, not the location of the hyperplane. A different step size would lead to a different $\mathbf{w}$, but the convergence properties of the algorithm would not change.

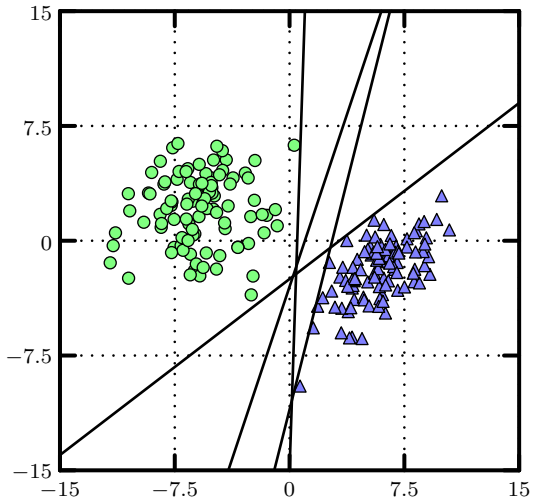


Figure 4.3: Examples of linear discriminants who would classify the training data correctly. Some would perform much better than others on future data, but the perceptron criterion cannot distinguish between them.

- A single perceptron can only handle linearly separable data. Thanks to the non-linear step function, however, multiple layers of perceptrons would be capable of handling non-linearly separable problems. This insight was only acquired decades after the discovery of the perceptron. However, although layers of perceptrons could handle complex problems, training would still be an enormous problem. Training a single perceptron can already take ages, and may not find a very good solution. Combinations of perceptrons would be exponentially harder.

  In chapter 8, we shall see an extension of the original perceptron which address the above shortcomings: artificial neural networks or multi-layer perceptrons (MLP).

4.2 More about Gradient Descent

When optimizing a classifier or regressor on a training set, we often end up performing gradient descent on a loss function (or, occasionaly, gradient ascent on a utility function[3]). This function takes into account the output of our machine, as computed on all the datapoints in our training set, as well as all the corresponding labels. For example, this function could be the average squared error between the predicted output and the desired value of a regression problem; the total probability of the targets given the inputs under a probabilistic model or, in the case of our linear perceptron, the sum over all misclassified datapoints, of the (sign-corrected) value of our linear function (Eq. (4.9)).

$$E_p(\mathbf{w}) = -\sum_{n\in\mathcal{M}} \mathbf{w}^\top \mathbf{x}_n \, t_n, \tag{4.9}$$

If we take the gradient of this function with respect to $\mathbf{w}$, we get

$$\nabla_{\mathbf{w}} E_p(\mathbf{w}) = -\sum_{n\in\mathcal{M}} \nabla_{\mathbf{w}} \left(\mathbf{w}^\top \mathbf{x}_n \, t_n\right) \tag{4.10}$$

$$= -\sum_{n\in\mathcal{M}} \mathbf{x}_n \, t_n \tag{4.11}$$

[3] A loss or error function increases as the performance becomes worse, a utility function increases as the performance gets better.

We could be optimising this function as follows: take the current value of $\mathbf{w}$, create the set of misclassified datapoints $\mathcal{M}$, cycle through this set and add the sign-corrected datapoints $\mathbf{x}_n\, t_n$ to $\mathbf{w}$. Then re-evaluate $\mathcal{M}$ based on the new value of $\mathbf{w}$. In this scenario, we are performing gradient descent on the error function with a step size of 1. This is also called "*batch*" gradient descent, because we are taking the whole batch of training data to compute both the error and the gradient of that error.

In the case of the Perceptron Learning algorithm, we re-evaluate $\mathcal{M}$ each time we've updated $\mathbf{w}$, for every misclassified datapoint we've processed. This is called "*stochastic*" gradient descent, because we're not actually computing the gradient of the error function as computed on the complete training data: rather, we are computing the gradient on a random subset of the dataset, *viz.*. the next misclassified datapoint we happen to encounter.[4] The reason why this works is that, even though we are not computing the gradient of the actual error, we are doing something that comes close when all datapoints have been considered.

So, why should we use an approximation to the gradient when we could just as easily use the actual gradient? Well, it turns out there are advantages to stochastic gradient descent:

1. Updates are fast: we don't need to process the whole dataset for each update, and this can make a real difference for very large datasets where the algorithm can converge before the entire dataset was even processed once using stochastic gradient descent.

2. Redundancy in the dataset is handled better. Consider the case where you have a dataset for which you perform gradient descent. As a thought experiment, imagine that you duplicate each datapoint in the dataset. If one were to perform batch gradient descent on this new dataset, the gradients would be identical to those of the original dataset, but would take twice as long to compute (and the memory requirements would double), while in the case of stochastic gradient descent, there is no performance penalty at all. This is an extreme case, but in many real-world datasets, some redundancy will be present, whether in the form of duplicated datapoints, or in the form of datapoints that are similar to each other. Stochastic gradient descent is then more effective.

3. Local optima can be avoided more easily. When the algorithm has reached a local optimum, and the gradient of the error function (computed over the complete training data) is, therefore, close to zero, it is difficult for an algorithm relying on the batch gradients to escape that optimum and reach a better optimum. When performing stochastic gradient descent, the gradient computed on each individual datapoint is typically not zero, however, and the algorithm keeps moving around to find better optima.

The stochastic nature of stochastic gradient descent does have the drawback that it tends to keep modifying the parameters stochasticly and that, unless the optimum of the function consists of a vast region of the parameter space (as is the case for the linear perceptron, where any discriminant that correctly separates the two classes has zero error and is optimal), the algorithm can stumble around forever. It is therefore important to keep track of the performance over time, to store the best model found so far, and to end the optimisation when no improvement has been made for a large number of iterations.

As a final note, stochastic gradient descent is an extreme case of the approximation to the gradient, where but a single datapoint is considered to approximate the gradient of a function that considers all datapoints. A less extreme version of such an approximation is when we consider a few datapoints instead, and we can then vary that number. Doing so is called "*mini-batch*" gradient descent, and it combines advantages from both methods.

4.3 Probabilistic modelling

Classification using some function of the inputs can be powerful and fast, both in training and in testing. Nevertheless, it exhibits one major problem: it does not provide us with information about the certainty of the classification.

Rather than looking for a function that separates different classes, we can instead look at how the data of the different classes is distributed. Based on our knowledge of this distribution, we can then compute the probability that a given datapoint should belong to one or the other class. The discriminant between

[4] The dataset can be processed by considering each datapoint in sequence or, better, by repeatedly sampling datapoints at random.

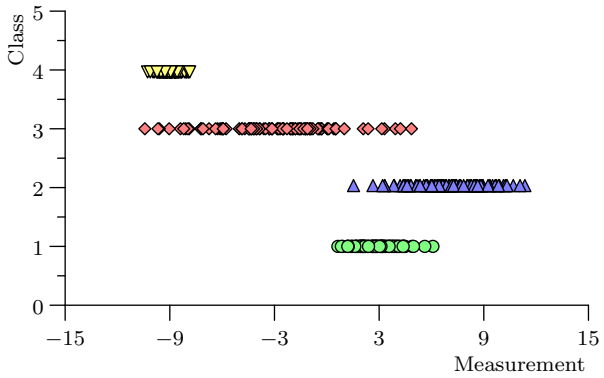


Figure 4.4: Illustration of the joint probability of class labels and measurements. In this case, we have multiple datapoints consisting of a single measurement and the corresponding class label. The class label is indicated in the y axis, and is discrete ($\mathbf{z} \in \{\mathcal{C}_1, \ldots, \mathcal{C}_4\}$). The measurement is continuous.

two classes then corresponds to the set of positions in dataspace where it is equally likely for a datapoint to belong to those two classes. As we will see shortly, for some distributions, this discriminant is linear.

4.3.1 Generative models

When building a probabilistic model of a classification problem, we are interested in the joint probability of the data examples $\mathbf{X} = \mathbf{x}_{1:N}$ and the corresponding class labels $\mathbf{Z} = \mathbf{z}_{1:N}$, $p(\mathbf{X}, \mathbf{Z})$. A simple artificial 1D example of such a joint distribution is shown in Figure 4.4. From this joint distribution, we can then derive the quantity that we're interested in: the probability of a certain class label given a data point, $p(\mathbf{z}|\mathbf{x})$. There are two ways in which we can obtain this quantity, and these result in the two great families of probabilistic models. We could factorise $p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x}|\mathbf{z})p(\mathbf{z})$ and use Bayes' rule to obtain

$$p(\mathbf{z}|\mathbf{x}) = \frac{p(\mathbf{x}|\mathbf{z})\, p(\mathbf{z})}{p(\mathbf{x})}, \text{ where } p(\mathbf{x}) = \sum_{\mathbf{z}} p(\mathbf{x}|\mathbf{z})\, p(\mathbf{z}) \ . \tag{4.12}$$

Such models are called *generative models*. The other possibility is to learn a functional form for $p(\mathbf{z}|\mathbf{x})$ directly. Such models are called *discriminative models*. Both types have their pros and cons.

Generative probabilistic models typically have the advantage that they are easier to train than discriminative models. In general (and contrary to popular belief) they also require less data to obtain a good representation of the joint likelihood distribution, and if the model is correct (*i.e.*, if the functions chosen to represent the distribution are indeed capable of representing the real distribution of the data,) using Bayes' rule to select the most probable class results in the lowest possible error in the limit of infinitely many classifications.

Let us consider the simple, 2-dimensional example of Figure 4.1 again. The input space has two dimensions (the class label could be plotted as a third dimension, or be represented by the colour or shape of the point, as in this case.) To construct a generative model, we start by learning what the probability is that a data point should belong to a certain class, if we don't know the actual value of the parameter. If class $\mathcal{C}_1$ occured much more often than class $\mathcal{C}_2$, we would know that the probability that a datapoint belongs to $\mathcal{C}_1$ would be higher, *a priori*, compared to $\mathcal{C}_2$. This probability is, therefore, called the *prior probability*. In this case, there are exactly 100 examples of each class, and if this were the only information that we had, we would assume that $p(\mathbf{z} = \mathcal{C}_1) = p(\mathbf{z} = \mathcal{C}_2) = \frac{1}{2}$. In some cases, we would know that the dataset is not reflecting the real prior distributions accurately, and we would encode other prior beliefs in the chosen values for the prior probabilities.

The next quantity to consider is the likelihood, $p(\mathbf{x}|\mathbf{z})$. We need to find some way to represent the distribution of the data in each class: in this case, a 2D Gaussian distribution seems to be a good choice. The Gaussian distribution is important, because it has some nice mathematical properties, and it appears to fit the distribution of many variables in the real world very well. For details on this distribution, refer to Appendix B.

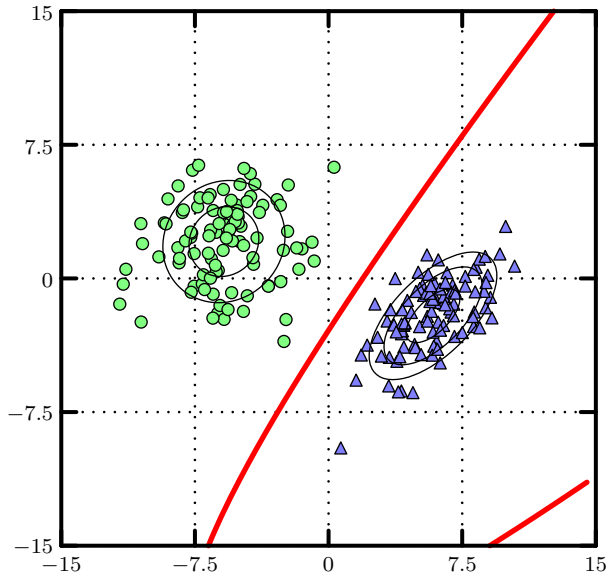


Figure 4.5: Fitting of Gaussian distributions to the example data.

The multivariate Gaussian distribution has two parameters, the mean vector and the covariance matrix. To learn the parameters of the distribution, we find the parameter values which maximise the likelihood. Details on how to do this are found in section 7.1. The resulting distributions are shown as contours in Figure 4.5, and the resulting discriminant is shown as a red line in the same figure.

Linear discriminant The discriminant in a probabilistic classification model indicates where $p(\mathcal{C}_1|\mathbf{x}) = p(\mathcal{C}_2|x)$. We could see in Figure 4.5 that when Gaussian distributions are used to describe the class-conditional distribution of the data, the resulting discriminant is non-linear. In fact, it is a quadratic function of the input, a fact that is hard to see graphically but easy to derive mathematically, as follows.

The Gaussian distribution is given by:

$$p(\mathbf{x}|\mathbf{z} = \mathcal{C}_k) = \frac{1}{(2\pi)^{d/2}|\mathbf{\Sigma}_k|^{1/2}} \exp -\tfrac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^\top \mathbf{\Sigma}_k^{-1}(\mathbf{x} - \boldsymbol{\mu}_k) \ , \tag{4.13}$$

which is scary the first time you encounter it, but is actually pretty easy to work with. We want to find the function where $p(\mathcal{C}_j|\mathbf{x}) = p(\mathcal{C}_k|\mathbf{x})$. To make our life simpler, we will actually look for the function where $\log p(\mathcal{C}_k|\mathbf{x}) = \log p(\mathcal{C}_l|\mathbf{x})$ (which is equivalent, because the logarithm is a monotonically increasing function). The logarithm of the Gaussian distribution is given by:

$$\log p(\mathbf{x}|\mathcal{C}_k) = -\frac{d}{2}\log(2\pi) - \frac{1}{2}\log|\mathbf{\Sigma}_k| - \tfrac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_k)^\top \mathbf{\Sigma}_k^{-1}(\mathbf{x} - \boldsymbol{\mu}_k) \ , \tag{4.14}$$

and if we set the log-probabilities of two classes equal to each other, $\log p(\mathcal{C}_j|\mathbf{x}) = \log p(\mathcal{C}_k|\mathbf{x})$ we obtain, after simplification

$$\log p(\mathcal{C}_j) - \log|\mathbf{\Sigma}_j| - (\mathbf{x} - \boldsymbol{\mu}_j)^\top \mathbf{\Sigma}_k^{-1}(\mathbf{x} - \boldsymbol{\mu}_j) = \log p(\mathcal{C}_k) - \log|\mathbf{\Sigma}_k| - (\mathbf{x} - \boldsymbol{\mu}_k)^\top \mathbf{\Sigma}_k^{-1}(\mathbf{x} - \boldsymbol{\mu}_k) \tag{4.15}$$

or

$$\begin{aligned}\log p(\mathcal{C}_j) - \log|\mathbf{\Sigma}_j| - \mathbf{x}^\top \mathbf{\Sigma}_j^{-1}\mathbf{x} + 2\mathbf{x}^\top \mathbf{\Sigma}_j^{-1}\boldsymbol{\mu}_j - \boldsymbol{\mu}^\top \mathbf{\Sigma}_j^{-1}\boldsymbol{\mu}_j \\ = \log p(\mathcal{C}_k) - \log|\mathbf{\Sigma}_k| - \mathbf{x}^\top \mathbf{\Sigma}_k^{-1}\mathbf{x} + 2\mathbf{x}^\top \mathbf{\Sigma}_k^{-1}\boldsymbol{\mu}_j - \boldsymbol{\mu}^\top \mathbf{\Sigma}_k^{-1}\boldsymbol{\mu}_j\end{aligned} \tag{4.16}$$

The only terms in Eq. (4.16) which are a function of $\mathbf{x}$ are $\mathbf{x}^\top\mathbf{\Sigma}^{-1}\mathbf{x}$, which is quadratic in $\mathbf{x}$, and $\mathbf{x}\mathbf{\Sigma}^{-1}\boldsymbol{\mu}$, which is linear in $\mathbf{x}$. The other terms are constant. As a consequence, the discriminant function is generally

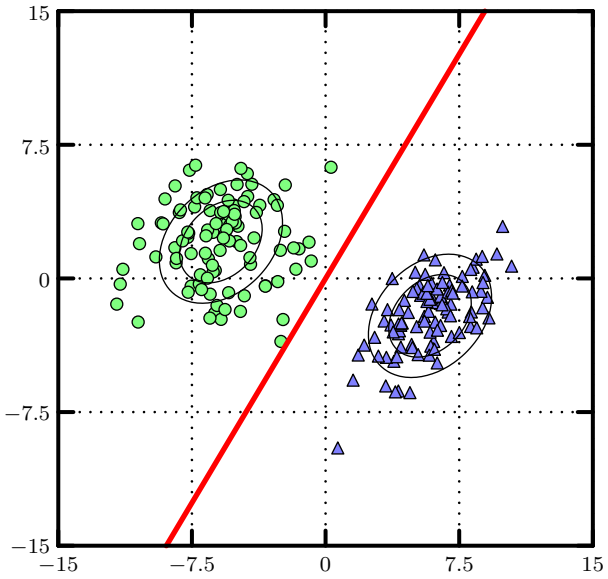


Figure 4.6: Fitting of Gaussian distributions to the example data, while constraining the covariances to be identical.

quadratic in $\mathbf{x}$, and will only be linear when the quadratic terms cancel out, *i.e.*, when $\mathbf{x}^\top \boldsymbol{\Sigma}_j^{-1}\mathbf{x} = \mathbf{x}^\top \boldsymbol{\Sigma}_k^{-1}\mathbf{x}$. This is the case when $\boldsymbol{\Sigma}_j^{-1} = \boldsymbol{\Sigma}_k^{-1}$. Notice that the linear terms do not cancel out when the covariances are identical, as long as the means are different. If we constrain the covariances to be identical when learning the distributions, we obtain the classifier depicted in Figure 4.6. This latter model is important for two reasons: historically, because it is formally equivalent with Fisher's discriminant, which is widely used, and practically, because constraining the model to be less flexible makes it less likely to overfit. When learning models by maximum likelihood, overfitting is a permanent concern, and constraining the covariance matrix of Gaussian distributions is often beneficial in practice. It reduces the number of free parameters which, otherwise, would be quadratic in the number of dimensions of the data: when few datapoints are available, or when the data is high-dimensional, different constraints can be applied to the covariance matrix.

4.3.2 Discriminative models

Where the generative models described the complete joint distribution of the data and the labels, $(\mathbf{x}, \mathbf{z})$, the quantity that we are really interested in, at the end of the day, is $p(\mathbf{z}|\mathbf{x})$. Instead of learning the whole joint distribution, we could focus instead on learning $p(\mathbf{z}|\mathbf{x})$ directly. If our model is correct, the two are equivalent, and the discriminative model cannot outperform the generative model. If the model is incorrect, however, the discriminative model can outperform the generative model. In order to do so, however, it needs

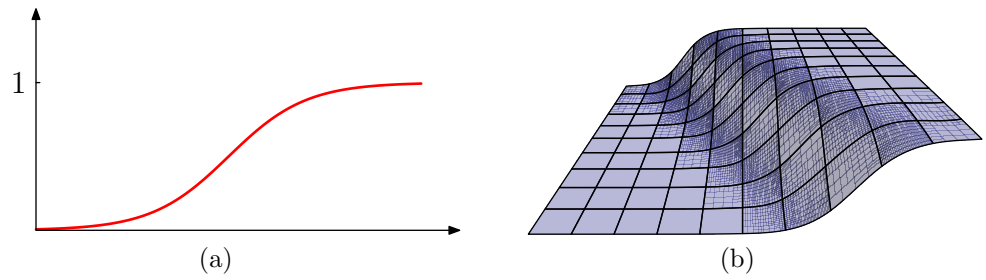


Figure 4.7: Figure (a): plot of $\sigma(x)$, where x is a scalar. Figure (b): plot of $\sigma(x_1 + x_2/2)$, where $\mathbf{x}$ is a two-dimensional vector.

sufficient data [Ng and Jordan, 2002].

An example of such a model is logistic regression. It assumes the following functional form for the posterior distribution:

$$p(\mathbf{z}|\mathbf{x}) = \sigma(\mathbf{w}\mathbf{x}) \tag{4.17}$$

$$= \frac{1}{1 + e^{-\mathbf{w}\mathbf{x}}} , \tag{4.18}$$

where, as usual, we extend $\mathbf{x}$ with $x_0 = 1$ so that we consider the bias term w_0 to be part of the vector $\mathbf{w}$ for notational convenience. Figure 4.7 shows what this function might look like in one and two dimensions.

Since the posterior probability is given by a linear function of the input space, the discriminant (where $p(\mathbf{z}|\mathbf{x}) = 0.5$) will be a linear function of $\mathbf{x}$. The parameter vector $\mathbf{w}$ defines *where* that discriminant lies and *how steeply* the posterior probability transitions from one class to the other, but the functional form of the discriminant is fixed by the functional form of the posterior probability function.

And this is where we can gain some insight as to why discriminative models can perform better than generative models when the modelling assumptions have been violated: we saw in the previous section that a generative model where the likelihood is given by Gaussian PDF will result in a linear discriminant if the covariance matrices are constrained to be the same for all classes. This may not be a valid constraint in practice, and enforcing it can result in a sub-optimal approximation of the posterior probability. Yet Gaussian PDFs with identical covariances are not the only PDFs that result in a linear discriminant. If our data had such a non-Gaussian distribution with a linear discriminant between the classes, then fitting Gaussian distributions to it, and constraining the covariances of these Gaussians to be identical (in order to ensure that the discriminant be linear) will indeed result in a linear discriminant, but a wrong linear discriminant, even though the correct discriminant is nevertheless also linear. Logistic regression, if trained on sufficient data, will find that correct discriminant, and, in general, may approximate the correct posterior probability more closely than a generative model would — even if its modelling assumptions also violate the true data distribution.

At the same time, estimating the parameters for the Gaussian distribution of each class requires few parameters, especially if the covariance matrix is shared. Finding the parameters of the logistic function requires more data, which illustrates how the advantage of discriminative models only becomes apparent when sufficient data is available.

Also notice that the parameters of the Generative models can be learned in closed form. By contrast, the parameter vector of logistic regression can only be optimised using numerical techniques such as gradient descent or improved variations thereof. In general, discriminative models are harder to train than generative models.

Finally, by definition, generative models model the distribution of the data. This has the advantage that they can deal with missing data, something which discriminative models cannot do. Being able to deal with missing data can be a huge advantage in practice, where, for example, it is often the case that the class labels are not known for all elements in the dataset. In such cases (called semi-supervised problems), generative models can use all the data, while discriminative models can only use the labelled examples.

4.4 Fisher's Linear Discriminant

One last linear discriminant that's worth mentioning, if mostly for historical reasons, is called Fisher's linear discriminant. It is based on the insight that building a linear hyperplane that separates two classes is equivalent to first projecting the whole dataset down (linearly) onto a single dimension, and then choosing the point in that one dimension that separates the two classes best. If we do such a projection, then the intuition behind Fisher's linear discriminant is that for good class separation, we want the means of the projected data to be far apart. However having the means far apart is not sufficient for good classification, if the variances of the projected classes are large enough to create overlap between the classes. The idea is then to find the linear projection that makes the distance between the projected means as large as possible, while at the same time keeping the variance of each projected class as small as possible.

Figure 4.8 illustrates the projection using Fisher's linear discriminant.

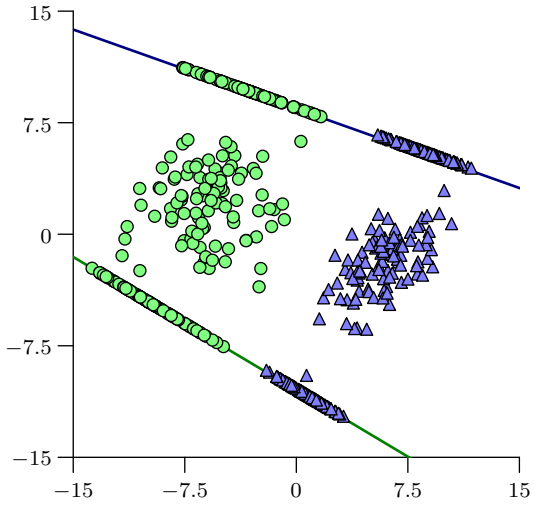


Figure 4.8: Illustration of Fisher's linear discriminant. The projection that maximises the distance between the means is given in blue; the projection that simultaneously minimises the within-class variance is given in green.

4.5 Non-linear Basis Functions

Linear discriminants are nice to work with because they are simple to implement, easy to train and simple enough not to be prone to overfitting. At the same time, of course, they are too simple for many problems: in many real-world classification problems the classes are not linearly separable. We will see more advanced classification techniques later, but before we get there, we should mention a simple yet very powerful trick that can be used to make non-linearly separable classes linearly separable.

The idea is that the data may not be separable in the observation space, but that it may be possible to make a fixed mapping from this space to another space where the datapoints would be linearly separable. The functions that operate this change of basis, called basis functions, must necessarily be non-linear for a non-linearly separable problem to become linearly separable. Figure 4.9 illustrates such a problem. The original data is not linearly separable, but the transformed data is. Of course, the difficulty is now that we need to find out how to transform the data in order to obtain linearly separable data.

Learning the non-linear basis functions from data is similar to training a non-linear classifier, and we will see in the future that some non-linear classifiers, most notably the Multi-Layer Perceptron, can be decomposed into a linear classifier and learnt non-linear basis functions. Such classifiers are sometimes called adaptive basis functions for that very reason.

4.5.1 Basis functions for regression

Although basis functions can be an important part of classification, they are even more important in the case of regression. Consider, for example, the problem of the polynomial curve fitting example used in section 3.2. Here, we found the weights for a weighted sum, so that this weighted sum approximates the target measurements as closely as possible for each input measurement. This sum consisted of fixed, nonlinear transformations of the input measurements: in other words, each polynomial function is a nonlinear basis function of the input variables, and we sought for a linear combination of these basis functions to approximate the target variable. We shall return to the use of basis functions at greater length when discussing kernel methods.

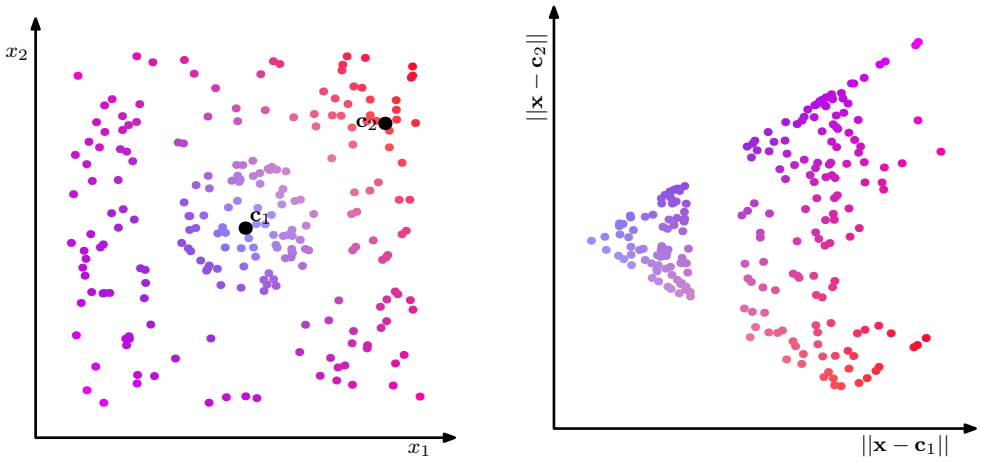


Figure 4.9: Transforming data with fixed, non-linear basis functions resulting in a space where the classes are linearly separable. In this example, x_1 has been replaced by the Euclidean distance from $\mathbf{c}_1$, while x_2 has been replaced with the Euclidean distance from $\mathbf{c}_2$. The points in both plots are shaded according to their distance from $\mathbf{c}_1$ and $\mathbf{c}_2$: points are shaded red according to their proximity to $\mathbf{c}_1$ and blue according to their proximity to $\mathbf{c}_2$.

Chapter 5

Bayesian Decision Theory

5.1 Classification

As we have seen in the previous chapter, using the joint probability distribution of all the variables in the system allows us to answer any questions about the system. Describing this joint probability is often difficult, and some thought must be given to how this is done. In this chapter, we will see how the joint probability distribution can be represented, how it can be learnt from examples, how it can be used to answer questions about variables in the system, and how much information one variable can give about another variable.

5.2 Optimal decisions

In section 4.3, we have seen a first example of how the joint probability of the observed variables $\mathbf{x}$ and the corresponding desired target variables t, $p(\mathbf{x}, t)$, provides us with all the information we need about what targets to expect for a given input. It also allows us to quantify the uncertainty associated with a prediction $\hat{t}$. The next few sections will investigate in more detail how we can learn this joint distribution, but at the end of the day we will need to make decisions based on this quantity. For example, imagine that you should know the probability of whether it shall rain today or not, based on a number of measurements (such as temperature, whether it rained during the night, the presence of clouds, *etc.*) contained in a vector $\mathbf{x}$. This knowledge is useful in order to decide whether to take an umbrella when leaving your den: taking your brolly[1] (or not) is a deterministic decision, and for either choice you can compute the probability that it should be the wrong choice (*i.e.*, taking the umbrella and not using it, or leaving the umbrella and be soaked).

We would like our decision to be optimal in some way. For example, we could choose to minimise the number of wrong decisions. In that case, it would seem intuitive to predict the outcome with the highest posterior probability; in our example, that would mean predicting rain if the posterior probability of rain is higher than the posterior probability of no rain. To show that this intuition is indeed correct, we compute the probability of making a mistake, and minimise that. The probability of a mistake, $p(\text{mistake})$ is given by $p(\mathcal{C}_1, \hat{\mathcal{C}}_2) + p(\mathcal{C}_2, \hat{\mathcal{C}}_1)$, the probability of predicting class two when the datapoint belongs to class one, plus the probability of predicting class one when the datapoint belongs to class two. In our example, we could have $\mathcal{C}_1$ indicating that it will rain today, $\hat{\mathcal{C}}_1$ indicating our decision to take the umbrella (*i.e.*, us deciding that it will indeed rain today), $\mathcal{C}_2$ that it will not rain today and $\hat{\mathcal{C}}_2$ the decision not to take our umbrella. We simply apply the familiar rules of probability to compute these quantities:

$$p(\mathcal{C}_1, \hat{\mathcal{C}}_2) = \int_{-\infty}^{\infty} p(\mathcal{C}_1, \mathbf{x}, \hat{\mathcal{C}}_2) \mathrm{d}x \qquad p(\mathcal{C}_2, \hat{\mathcal{C}}_1) = \int_{-\infty}^{\infty} p(\mathcal{C}_2, \mathbf{x}, \hat{\mathcal{C}}_1) \mathrm{d}x \tag{5.1}$$

and

$$p(\mathcal{C}_1, \mathbf{x}, \hat{\mathcal{C}}_2) = p(\hat{\mathcal{C}}_2 | \mathbf{x}, \mathcal{C}_1)\, p(\mathbf{x}, \mathcal{C}_1) \qquad p(\mathcal{C}_2, \mathbf{x}, \hat{\mathcal{C}}_1) = p(\hat{\mathcal{C}}_1 | \mathbf{x}, \mathcal{C}_2)\, p(\mathbf{x}, \mathcal{C}_2) \tag{5.2}$$

[1] "*Noo!*", yells the proofreader, "You can't do that; only people from the UK would know that brolly means umbrella". Well, not anymore.

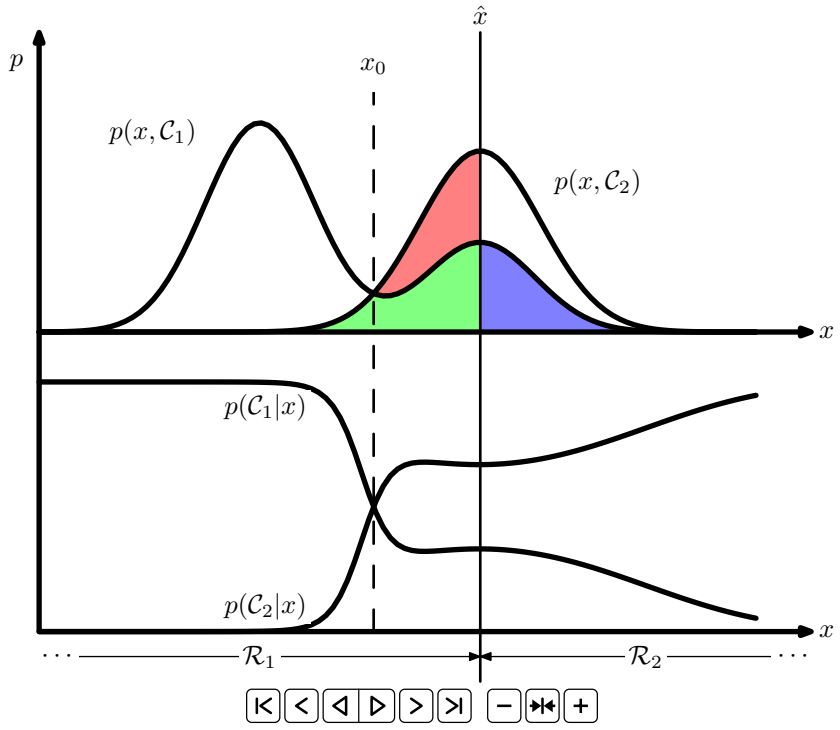


Figure 5.1: Plot of the joint probabilities of the classes $\mathcal{C}_1$ and $\mathcal{C}_2$, and the observations. Note that $p(\mathcal{C}, \mathbf{x}) = p(\mathbf{x}|\mathcal{C})\, p(\mathcal{C})$, and that the area under the sum of the two curves equals one. The decision threshold $\hat{x}$ indicates how points are classified: points belonging to $\mathcal{R}_1$ (*i.e.*, $x < \hat{x}$) are classified as $\mathcal{C}_1$, while points belonging to $\mathcal{R}_2$ ($x > \hat{x}$) are classified as $\mathcal{C}_2$. The green area indicates the amount of misclassified points from class $\mathcal{C}_2$ for which the same amount of correctly classified points from class $\mathcal{C}_1$ exist. The blue area indicates the converse: the amount of misclassified points from class $\mathcal{C}_1$ for which the same amount of correctly classified points from class $\mathcal{C}_2$ exist. Both areas are fixed contributions to the error, because the correct classifications and erroneous classifications cancel each other out in those areas. The red area indicates the additional misclassifications, from either class. This area is zero when the joint probabilities are equal and, therefore, when the conditional probabilities $p(\mathcal{C}_k|\mathbf{x})$ are equal for all k (indicated by x_0 in the plot). Note that although this plot was made for univariate observations (hence the use of scalar x rather than vector $\mathbf{x}$), the reasoning holds for both univariate and multivariate observations.

The decision of predicting one class or the other depends only on the input $\mathbf{x}$ and is deterministic, so that the probability of predicting $\hat{\mathcal{C}}_1$ is either zero or one, depending on the value of $\mathbf{x}$. This is illustrated in Figure 5.1, where we predict $\hat{\mathcal{C}}_1$ if $\mathbf{x}$ falls in region $\mathcal{R}_1$, and $\hat{\mathcal{C}}_2$ if $\mathbf{x}$ falls in $\mathcal{R}_2$: the probability of

$$p(\hat{\mathcal{C}}_2|\mathbf{x}, \mathcal{C}_1) = p(\hat{\mathcal{C}}_2|\mathbf{x}) = \begin{cases} 0 & \text{if } \mathbf{x} \in \mathcal{R}_1 \\ 1 & \text{if } \mathbf{x} \in \mathcal{R}_2 \end{cases} \tag{5.3}$$

and similarly for the decision of $\hat{\mathcal{C}}_1$. From these, we get that

$$p(\text{mistake}) = \int_{\mathcal{R}_1} p(\mathbf{x}, \mathcal{C}_2)\mathrm{d}x + \int_{\mathcal{R}_2} p(\mathbf{x}, \mathcal{C}_1)\mathrm{d}x \tag{5.4}$$

Because the union of the decision regions must span our whole input space, we can only minimise $p(\text{mistake})$ by choosing the decision boundary such that $p(\mathbf{x}, \mathcal{C}_2) < p(\mathbf{x}, \mathcal{C}_1)$ in $\mathcal{R}_1$ and vice versa in $\mathcal{R}_2$. From the product rule of probabilities, we have that $p(\mathcal{C}_k, \mathbf{x}) = p(\mathcal{C}_k|\mathbf{x})\, p(\mathbf{x})$. Since $p(\mathbf{x})$ is common to all classes, it does not affect the decision boundary and the decision rule that minimises the probability of making a mistake is found by assigning each datapoint to the class with the largest posterior probability $p(\mathcal{C}_k|\mathbf{x})$.

5.3 Loss minimisation

Sometimes we do not simply want to minimise the number of mistakes, but rather to minimise the impact of these mistakes. Since some mistakes may be worse than others, it can be beneficial to increase the number of benign mistakes if that results in a reduced number of 'bad' mistakes. A typical example of such a situation is in the medical domain, where the misclassification of a diseased patient as healthy can have far worse consequences than the misclassification of a healthy patient as diseased.

In such cases, we can capture this discrepancy in a *cost function* or *loss function* where different types of mistake get different weights: instead of minimising the error, we minimise the expectation of the loss. This expectation of the loss (or expected loss) is the average value of the loss that we would converge to if we kept classifying datapoints forever. By minimising the expected loss, we ensure that we minimise the loss over a (sufficiently large) number of examples. This is the best we can do.

In the above example, the cost function might be given by the following matrix:

$$L = \begin{bmatrix} 0 & 1000 \\ 1 & 0 \end{bmatrix}, \tag{5.5}$$

where a correct classification carries no penalty, the misclassification of a healthy individual as cancerous carries a penalty of one, and the misclassification of a cancerous patient as healthy carries a penalty of one thousand. Following the same reasoning as in section 5.2, we can compute the expectation as

$$\mathbb{E}[L] = \sum_k \sum_j \int_{-\infty}^{\infty} L_{kj}\, p(\hat{\mathcal{C}}_j, \mathcal{C}_k, \mathbf{x})\mathrm{d}\mathbf{x} \tag{5.6}$$

$$= \sum_k \sum_j \int_{\mathcal{R}_j} L_{kj}\, p(\mathcal{C}_k, \mathbf{x})\mathrm{d}\mathbf{x} \tag{5.7}$$

$$= \sum_j \int_{\mathcal{R}_j} \sum_k L_{kj}\, p(\mathcal{C}_k, \mathbf{x})\mathrm{d}\mathbf{x} \tag{5.8}$$

which implies that we should choose our regions $\mathcal{R}_j$ so as to minimise $\sum_k L_{kj} p(\mathcal{C}_k, \mathbf{x})$. Since we can simplify out $p(\mathbf{x})$ as in section 5.2, we obtain that the optimal decisions (the decisions which result in minimal loss over many experiments) is obtained by choosing the class $\hat{\mathcal{C}}_j$ for each datapoint $\mathbf{x}$ as

$$\hat{\mathcal{C}}_j = \arg\min_j \sum_k L_{kj}\, p(\mathcal{C}_k|\mathbf{x}) \tag{5.9}$$

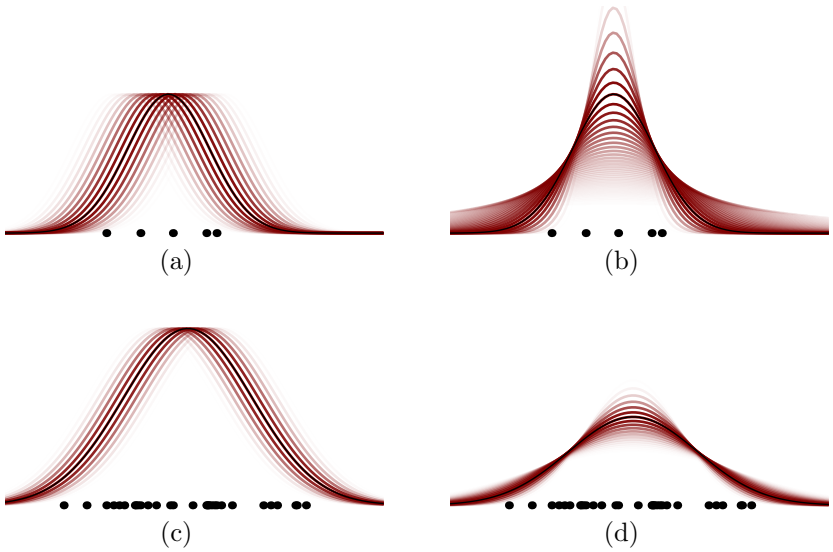


Figure 5.2: Illustration of the probability of PDF parameters given the data. the probability of the distribution is reflected in the shade of the plot: lighter for less probable distributions, and more intense for more probable distributions. The left plots (a and c) show the PDF for varying values of the mean μ, the right plots (b and d) show varying values of σ^2. The data consists of 5 normally distributed values for the top plots and 30 values for the bottom plots.

5.4 Estimating the distribution

The above computations assume that we know the distribution of the data. In reality, of course, this distribution is unknown and we need to estimate it from data. We can estimate the PDF that describes the distribution in three important ways:

1. by maximum-likelihood. In this case we simply choose the parameter values that maximise the probability of the data given those parameters, $p(\mathbf{X}|\boldsymbol{\theta})$.

2. by maximum a posteriori. Here, we explicitly state what our prior beliefs are about the parameter values and maximise the a posteriori probability of the parameters, $p(\boldsymbol{\theta}|\mathbf{X})$.

3. by Bayesian inference. Instead of finding the most likely distribution for our data, we acknowledge that the precise distribution is unknown. The distribution of the data is itself a random variable, which we can assign a probability to.

In the following, we shall mostly focus on maximum-likelihood (ML) techniques and *Maximum a Posteriori*(MAP) learning because, in general, these can be solved in closed form and are easy to work with. Moreover, if the amount of available training data is sufficiently large, the difference (in performance, but typically not in computational cost) between ML and fully Bayesian approaches becomes negligible. In practice, they are therefore quite frequently used. Bayesian techniques show their tremendous strength when the amount of available training data is small. Except for very specific models such as, for example, Gaussian Processes (Chapter being written), exact computations become intractable in full Bayesian models and approximations are then necessary: these will be treated in chapter 11.

But before we have a more detailed look at ML and MAP models, it is important to realise how they fit in the big picture. So this section tries to convey an intuitive understanding of Bayesian modelling.

5.4.1 Estimating a Gaussian PDF: Bayesian approach

Let us consider a simple example. We sampled some data, $\mathbf{X}$, from a Gaussian distribution (with zero mean and unit variance), as depicted in Figure 5.2. Based on this data, we would like to estimate the probability

density at a new datapoint $\mathbf{x}^*$. If we knew the distribution of the data, this would be easy: we could simply evaluate the PDF at $\mathbf{x}^*$. Unfortunately, we do not know the value of the parameters of that PDF.[2] If we consistently apply the rules of probability, we can estimate our desired quantity, $p(\mathbf{x}^*|\mathbf{X})$, as follows:

$$p(\mathbf{x}^*|\mathbf{X}) = \iint p(\mathbf{x}^*|\mu, \sigma^2)\, p(\mu, \sigma^2|\mathbf{X}) \mathrm{d}\mu \mathrm{d}\sigma^2 \tag{5.10}$$

Since we assume that the data has a Gaussian distribution, we can compute the probability of the data for any set of parameter values (μ, σ^2). From this, we can compute the probability of the parameter values given the data using Bayes' rule. We need to explicitly assume some prior over the parameter values, but let us assume that we don't know anything and choose a uniform distribution for those.[3]

The result is illustrated in Figure 5.3. Here, we compute the posterior probability of different value pairs for the parameters of the Gaussian PDF, and plot this as a colour map, for different amounts of training data. As the amount of training data is increased, the posterior distribution becomes more peaked around the ML estimator (red dot), and in the limit of infinitely many training examples, the ML estimator coincides with the ground truth while the complete probability mass is assigned to that ML estimator: the probability density at all other parameter values goes to zero.

Notice how Bayesian inference is, of course, not magically solving the problem of not having enough data to estimate our distribution. The sample mean of the data is, in the absence of informative prior knowledge, the most likely mean of the distribution and also the mode of our inferred distribution. But at least the ground truth distribution is given non-zero probability and, more importantly, datapoints that are further away from the mean are not assigned absurdly small probability. Informally, regions of the space which are given a low probability are only assigned a low probability when sufficiently data has been seen in regions of high probability to be confident in knowing what the low probability regions are.

5.5 Parametric Generative Models

In generative models, we find a function that describes the joint distribution of the observations and the quantities we are interested in. As a very simple and slightly artificial example, let us consider that we are playing the following game. I have two coins, one coin which is fair and will land with heads $(X = 1)$ and tails $(X = 0)$ with equal probability and another coin which will land with heads with probability $p(X = 1) = 0.6$. You do not know *a priori* which coin I am using, I could be using either with equal probability.

5.6 Example

You are a foreign student, newly arrived in Amsterdam. You don't really know what weather to expect, because you've got a few Spanish friends who told you that, basically, it's precisely what hell would be like if the devil had been sadistic enough to use water instead of fire, and you've got a few Scottish friends who told you the weather in Amsterdam is bloody fantastic. Now you wonder whether to bring your umbrella when coming to class. Doing so is great when it rains, but is a pain when the weather's dry, because your umbrella is one of these old-fashioned types that aren't extensible. They're great when you get caught up in a sword fight, but you figure you don't have to fear those that much. Not anymore... Well, not *here*, at least.

It did not rain when you stepped off of the plane, and in the first ten days of your stay in Amsterdam the weather was as follows: $\{\neg r, r, \neg r, \neg r, r, \neg r, \neg r, r, r, r\}$, meaning that it didn't rain the first day, it rained the second day, *etc.* What is your optimal strategy on each of those days?

First, we need to decide how we weight the different inconveniences. Let us encode the problem with the following loss function:

$$L(X, R) = \begin{bmatrix} 0 & 1 \\ 3 & 0 \end{bmatrix} \tag{5.11}$$

[2] Let us for now assume that the data has a Gaussian distribution; the only unknowns are the parameters of this distribution.

[3] This is not formally correct, as our parameters can take any value in $[-\infty, \infty]$ and our uniform PDF is therefore not properly normalised. Such a prior is called an *improper* prior, and must be handled with care: one must make sure that the posterior is normalised. In this particular case the posterior is indeed normalised.

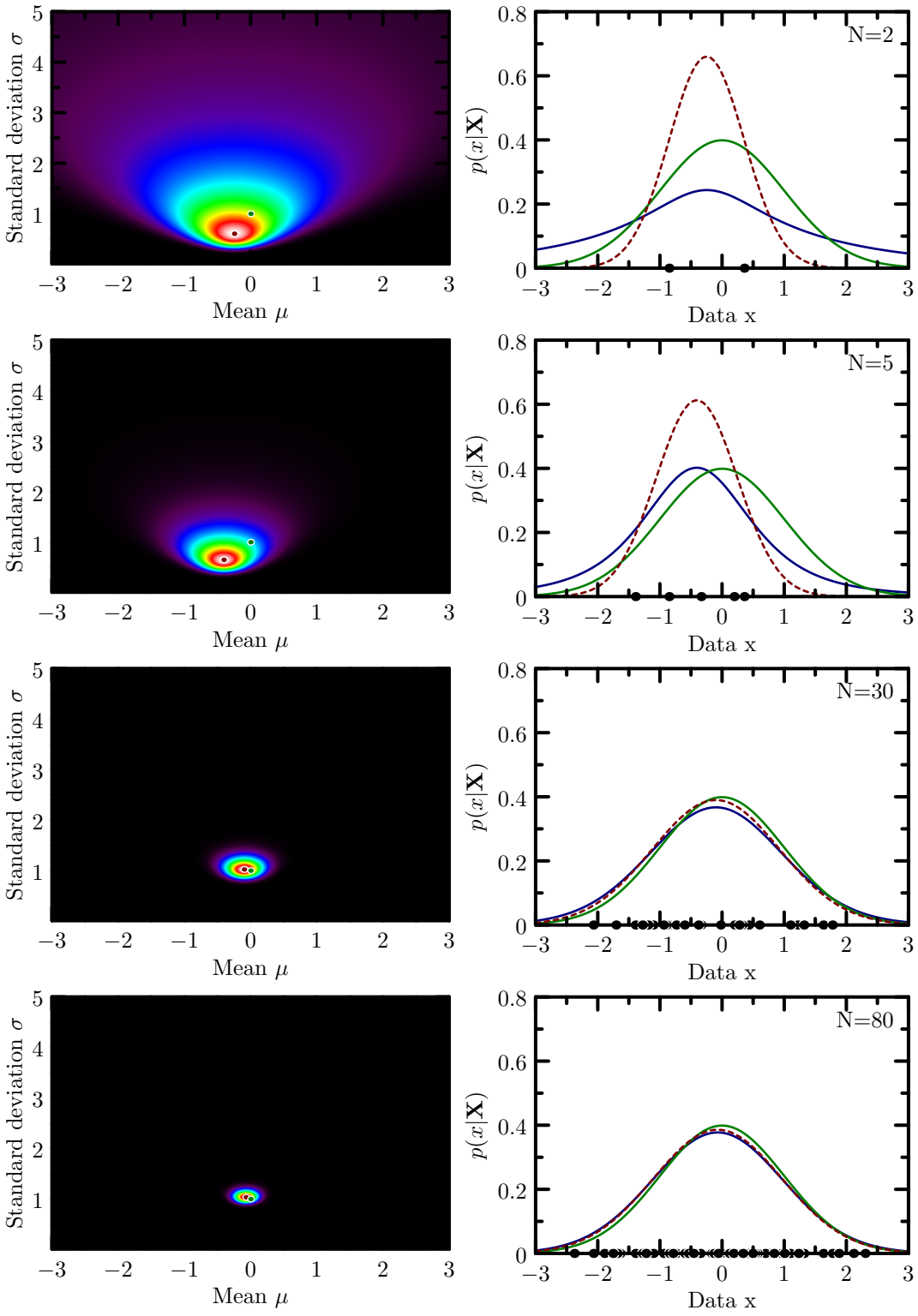


Figure 5.3: Illustration of the posterior probability of parameters μ and σ^2 for a gaussian distribution, with different amounts of training data. The left hand side shows the probability of (μ, σ) pairs given the training data. In these plots, the ML estimator is indicated with a red dot, while the ground truth is indicated with a green dot. On the right hand side, the data is shown as dots in the X axis, the ground truth data distribution is plotted in green, the ML PDF is indicated in dashed red and the PDF obtained by Bayesian inference is shown in blue.

If it rains and you don't carry your umbrella, you consider that three times more annoying than if it's dry and you have you're carrying umbrella for nothing. Should you carry your umbrella? We shall solve this problem in three ways, using maximum likelihood, using maximum a posteriori and using the Bayesian approach.

5.6.1 Expected loss

We want to minimise the expectation of loss, which is given by:

$$\mathbb{E}[L(x,r)] = \begin{bmatrix} L(x,r)p(r) + L(x,\neg r)p(\neg r) \\ L(\neg x,r)p(r) + L(\neg x,\neg r)p(\neg r) \end{bmatrix} \tag{5.12}$$

$$= \begin{bmatrix} p(\neg r) \\ 3\,p(r) \end{bmatrix} \tag{5.13}$$

In other words, we expect a loss of $p(\neg r)$ if we carry our umbrella, and a loss if $3\,p(r)$ if we don't. So now the question is: what is the probability that it will rain today. We encode this probability with a Bernoulli distribution, which takes a single parameter, μ:

$$p(r) = \mu^r(1-\mu)^{1-r} \ , \tag{5.14}$$

so that the question now is: how do we estimate μ?

5.6.2 Maximum likelihood

We assume that every day is another day, and that whether it rains or not is independent of whether it rained yesterday. The likelihood function $\ell(\mu)$ is then given by

$$p(x_1,\ldots,x_n) = \prod_{i=1}^{n} \mathrm{Bern}(x_i;\mu) \tag{5.15}$$

$$= \mu^{N_r}(1-\mu)^{N_{\neg r}} \ , \tag{5.16}$$

where N_r indicates the number of rainy days in the training set, and analogously for $N_{\neg r}$. The maximum-likelihood (ML) approach is to find the value of μ that maximises this likelihood function. It can easily be shown[4] that the ML solution for μ is given by

$$\mu = \frac{N_r}{N_r + N_{\neg r}} \tag{5.17}$$

On our first day, you have had a single experience, which was that it did not rain when you stepped off the plane. Your estimate for μ is therefore zero, and your expected loss if you take your umbrella ($= 1$) far outweighs your expected loss if you don't take it ($= 0$). This is obviously overfitting the data, and as time goes on our estimate fluctuates quite a lot (see Table 5.1). After sufficient data will have been seen (and assuming that the distribution does not change in the meantime),[5] we do eventually converge to the correct solution.

5.6.3 Maximum a posteriori

So, of course it's a bit unrealistic to rely only on our observations. After all, we had a lot of information about the weather in Amsterdam before stepping off that plane: information about its location on the globe, online information, friends' witness reports, *etc.* Let us say that, based on this knowledge, we believe that the probability that it should be dry every single day is zero: $p(\mu = 0) = 0$, that the probability that it should rain every day is zero as well ($p(\mu = 1) = 0$), and further that we believe it's more likely to rain than

[4] This is a common exam question. Do try to show it.

[5] It does actually change, of course, because the seasons change, but let's assume that for the sake of this experiment we have made the axis of rotation of the earth orthogonal to the earth's plane of rotation around the sun. We'll put things back as they were when we're done, not to upset anyone.

day	rain	μ	$\mathbb{E}[L]$
1	dry	$0/1 = 0$	$(1, 0)^\top$
2	rain	$1/2 = 0.5$	$(0.5, 1)^\top$
3	dry	$1/3 = 0.33$	$(0.67, 0.67)^\top$
4	dry	$1/4 = 0.25$	$(0.75, 0.50)^\top$
5	rain	$2/5 = 0.40$	$(0.60, 0.80)^\top$
6	dry	$2/6 = 0.33$	$(0.67, 0.67)^\top$
7	dry	$2/7 = 0.29$	$(0.71, 0.57)^\top$
8	rain	$3/8 = 0.38$	$(0.63, 0.75)^\top$
9	rain	$4/9 = 0.44$	$(0.56, 0.89)^\top$
10	rain	$5/10 = 0.5$	$(0.50, 1)^\top$

Table 5.1: Evolution of your estimate of μ and the expected loss as the days go by, using Maximum-Likelihood estimation.

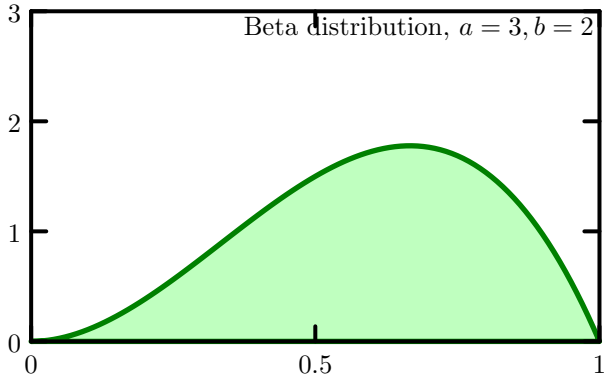


Figure 5.4: Prior over the value of μ

not. Such a prior over our parameter can be represented by a Beta distribution, for example the distribution shown in Figure 5.4, which has parameters $a = 3$ and $b = 2$. As we shall see below, this is equivalent with pretending that we have seen three rainy days and two dry days before we step out of the plane.

After our first observation ($r = 0$), we can compute the likelihood of that observation. This is given by the Bernoulli distribution, so that we can compute the posterior probability of the parameters as:

$$p(\mu|\mathbf{X}) = \frac{p(\mathbf{X}|\mu)\, p(\mu)}{p(\mathbf{X})} \tag{5.18}$$

In this, $p(\mathbf{X})$ is constant with respect to μ. Therefore, if we want to maximise $p(\mu|\mathbf{X})$, $p(\mathbf{X})$ is irrelevant and we can simply maximise $p(\mathbf{X}|\mu)\, p(\mu)$. If we wanted to compute it, it would simply be given by the partition function of the Beta distribution (see section B.2). The maximised function is given by

$$p(\mathbf{X}|\mu)\, p(\mu) \propto \mu^{N_r}(1-\mu)^{N_{\neg r}}\mu^{a-1}(1-\mu)^{b-1} \tag{5.19}$$

$$= \mu^{N_r+a-1}(1-\mu)^{N_{\neg r}+b-1} \tag{5.20}$$

which we can maximise using the usual trick of computing the logarithm and maximising that. The result is then:

$$\frac{\partial}{\partial\mu}(N_r + a - 1)\log\mu + (N_{\neg r} + b - 1)\log(1-\mu) = 0 \tag{5.21}$$

$$\frac{N_r + a - 1}{\mu} - \frac{N_{\neg r} + b - 1}{1-\mu} = 0 \tag{5.22}$$

$$(N_r + a - 1)(1-\mu) - (N_{\neg r} + b - 1)\mu = 0 \tag{5.23}$$

$$N_r + a - 1 - (N_r + a + N_{\neg r} + b - 2)\mu = 0 \tag{5.24}$$

$$\mu = \frac{N_r + a - 1}{N_r + a + N_{\neg r} + b - 2} \tag{5.25}$$

This result is illustrative of the more general facts that:

1. if you parametrise the prior over the parameters of a distribution with a conjugate prior, the MAP solution for the parameter values is equivalent with the ML solution on the training data with an additional constant amount of "virtual" data

2. the same result can trivially be obtained by Laplace smoothing. MAP learning is therefore rarely performed exactly: it does not provide anything more than Laplace smoothing does, and it is a little bit more complex to derive. MAP does, on the other hand, give us a nice explanation for why Laplace smoothing is a valuable heuristic to use.

5.7 The Bayesian approach

In the Bayesian approach, we consistently apply the rules of probability and bring them to their logical conclusion. The parameters of the distribution of the data are unknown, they cannot be determined exactly so, therefore, they should be treated like the random variables that they are. Bayes' theorem allows us to compute the probability of the values, and when we compute anything that involves the probability of the data, we just integrate out every variable that we are not interested in — including the model parameters.

Similarly to the MAP approach, we now need to specify explicitly what our prior beliefs are, which we encode in our prior distribution over our parameter values. As time goes by, we update our beliefs about the distribution's parameter values, as illustrated in Figure 5.5. Notice how the added information results in a more peaked distribution, reflecting our increased confidence in the estimate. In the limit for infinite amounts of data, we converge to the same value as the ML estimator, which then gets a probability of one.

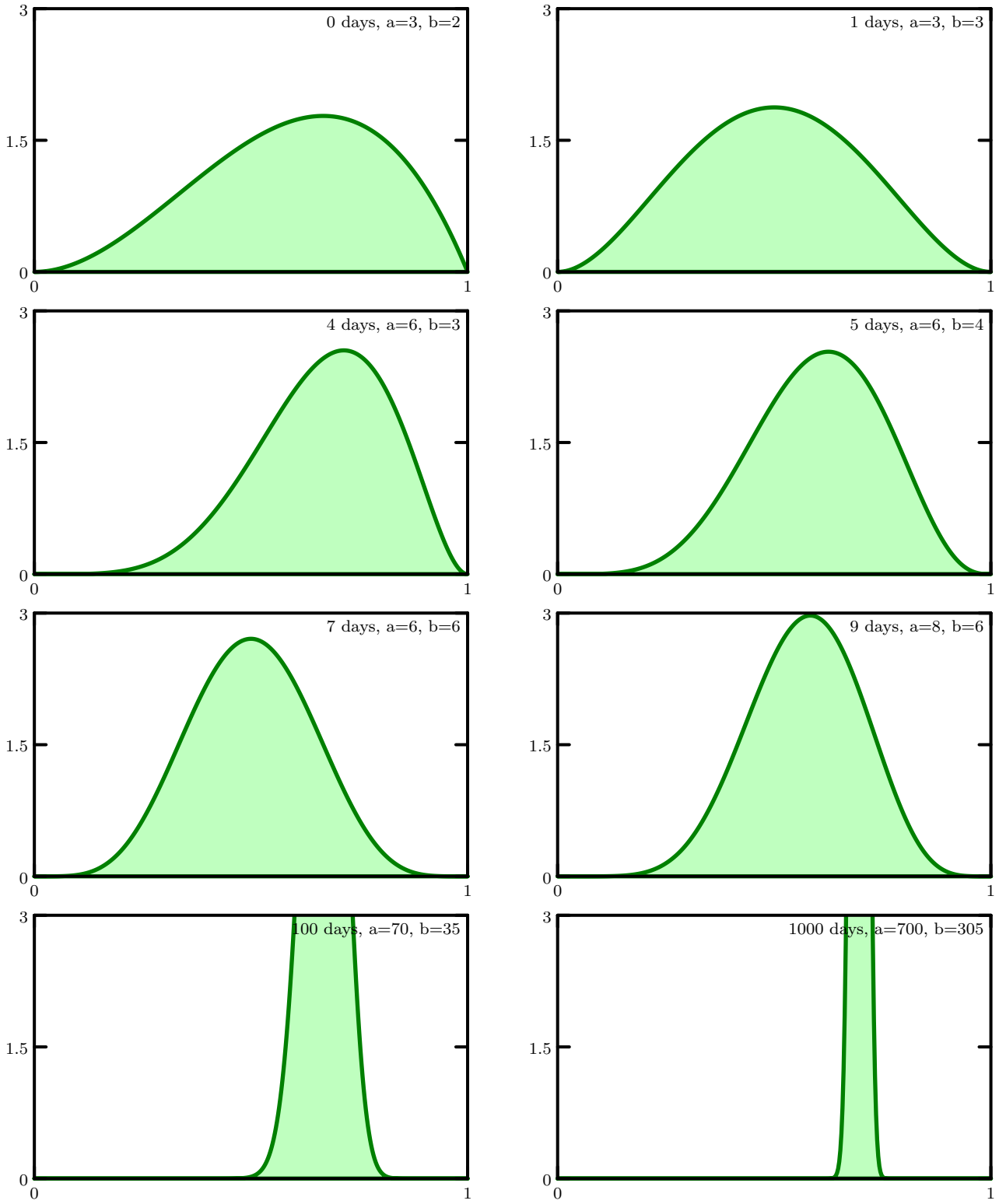


Figure 5.5: Evolution of the posterior probability distribution over the parameter μ of our Bernouli distribution, as we observe the weather on more days: before we see any data, we have a prior distribution over our parameter, which is informative but very broad. This reflects both our knowledge *and* our uncertainty. As time goes by, we observe more data, and we become more certain about the parameter values of our Bernouli PDF.

To compute our expected loss, we now need to marginalise out the variables that we are not interested in. In this case, we get that if we integrate out the parameter μ of the PDF,

$$p(x|\mathbf{X}) = \int_{-\infty}^{\infty} p(\mathbf{x}|\mu)\, p(\mu|\mathbf{X})\mathrm{d}\mu \tag{5.26}$$

$$= \int_{-\infty}^{\infty} \mu^x \,(1-\mu)^{1-x}\, \mathrm{Beta}(\mu; a, b)\mathrm{d}\mu \tag{5.27}$$

$$\tag{5.28}$$

which is the expectation of μ if $x = 1$ and one minus that expectation, if $x = 0$. The solution to this integral is therefore known, namely as the mean of the Beta distribution,

$$p(x|\mathbf{X}) = \mu^x \,(1-\mu)^{1-x} \qquad \text{where } \mu = \frac{a}{a+b} \tag{5.29}$$

Notice that despite the similarities, this result is different from the MAP solution, even for this very simple example. The ML solution sways most extremely with the data as more data are observed, the MAP solution avoids the most extreme parameter estimates by including our prior knowledge, but we can observe how the resulting estimate is always slightly more extreme than the value obtained by Bayesian inference.

This distribution is about the least complex that we can devise, since the possible values of the observations are so limited in their possible values. The difference between the different approaches is therefore also necessarily limited. Yet even in this simple case, using the Bayesian approach is guaranteed to outperform the other approaches in expectation: as the number of different datasets we observe approaches infinity, the probability approaches one, that the Bayesian approach outperforms the other approaches.

5.8 Information Theory

We introduce information theory quite informally. We want a measure for the "amount of information" obtained when observing an event. We consider that the amount of information obtained when observing the event depends on how likely we considered that event to be: for example, if the value to be observed was known beforehand, the amount of obtained information is zero.

If we observe two events, and these two events do not give information about each other (ie, they are independent), then the amount of information obtained from observing both should be equal to the sum of the information obtained by observing either of the events. We denote our measure of information as $h(x)$, so that from this last requirement, we obtain that if $p(x, y) = p(x)\, p(y)$, then $h(x, y) = h(x) + h(y)$. From this, we can derive that $h(x)$ must be proportional to the logarithm (with some arbitrary base b) of $p(x)$: $h(x) \propto \ln p(x)$. We can choose the proportionality constant as we see fit, but since we consider that observing improbable events gives us positive amounts of information (and that certain events give us zero information), this constant must be negative. The value of the constant changes the base of the logarithm, and in effect the unit in which we express the information: if we use the base-2 logarithm, we express our amount of information in 'bits', if we use the natural logarithm, we express our amount of information in 'nats'. The function for the amount of information obtained when observing x is given by:

$$h(x) = -\ln p(x) \tag{5.30}$$

in nats, or $h(x) = -\log_2 p(x)$ in bits.

If we now consider how much information a random event provides us with on average or, equivalently, how much information we need on average to transmit the real value of the variable, we compute the expected information of that variable as:

$$\mathrm{H}[x] = -\sum_x p(x)\, \ln p(x) \tag{5.31}$$

This quantity is called the *entropy* of the variable x. For a binary variable, the entropy is plotted as a function of $p(x)$ in Figure 5.7. For a discrete variable with three states, it is plotted as a function of $p(x_1)$

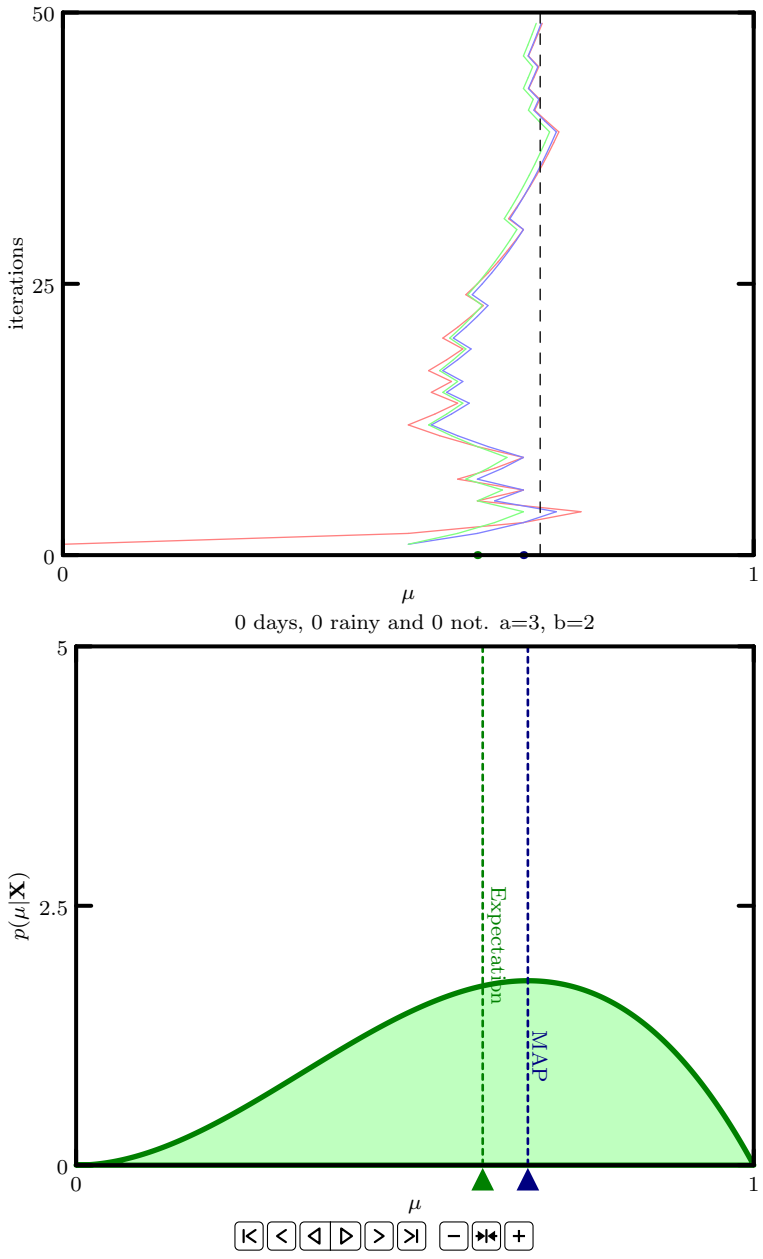


Figure 5.6: Animation of the evolution of the posterior distribution over the Bernouli parameter μ

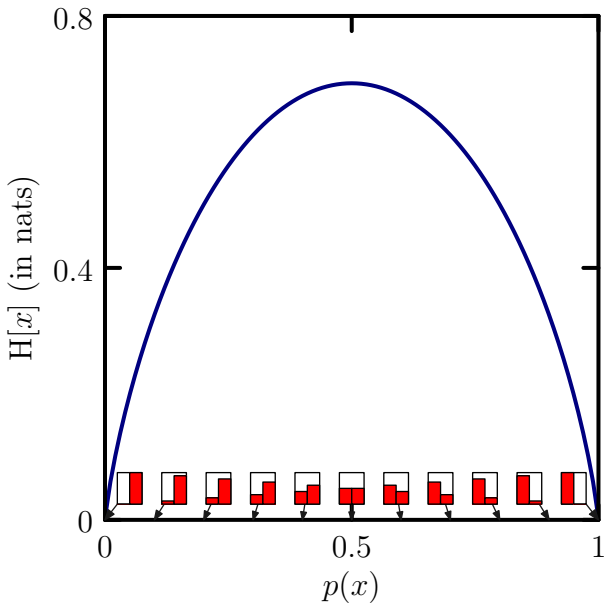


Figure 5.7: Plot of entropy for a binary random variable. The function is then $\mathrm{H}[x] = -\sum_i p(x_i) \ln p(x_i) = -p(x) \ln p(x) - (1 - p(x)) \ln(1 - p(x))$

and $p(x_2)$ in Figure 5.8. It is clear from the graph that the entropy is maximal when all states have the same probability. Intuitively, this is easy to understand: the expected amount of information is maximal when none of the states is more likely to be observed than the others. Formally, we can easily prove that this must be the case by maximising the entropy, using a Lagrange multiplier to constrain the probabilities to sum to one.

If we have two groups of variables, $\mathbf{x}$ and $\mathbf{y}$, then knowing something of the one may tell us something about the other (it will, if the variables are not independent). The *conditional entropy* tells us what the expected amount additional information is to specify one when we know the other:

$$\mathrm{H}[\mathbf{y}|\mathbf{x}] = \mathbb{E}_{p(\mathbf{x})}[\mathbb{E}_{p(\mathbf{y}|\mathbf{x})}[-\ln p(\mathbf{y}|\mathbf{x})]] \tag{5.32}$$

$$= -\iint p(\mathbf{y}|\mathbf{x}) \ln p(\mathbf{y}|\mathbf{x}) \mathrm{d}\mathbf{y} p(\mathbf{x}) \mathrm{d}\mathbf{x} \tag{5.33}$$

$$= -\iint p(\mathbf{x}, \mathbf{y}) \ln p(\mathbf{y}|\mathbf{x}) \mathrm{d}\mathbf{y} \mathrm{d}\mathbf{x} \tag{5.34}$$

From the entropy of the joint variables $\mathrm{H}[\mathbf{x}, \mathbf{y}]$, we get that

$$\mathrm{H}[\mathbf{x}, \mathbf{y}] = -\iint p(\mathbf{x}, \mathbf{y}) \ln p(\mathbf{x}, \mathbf{y}) \mathrm{d}\mathbf{y} \mathrm{d}\mathbf{x} \tag{5.35}$$

$$= -\iint p(\mathbf{x}, \mathbf{y}) \left[\ln p(\mathbf{x}) + \ln p(\mathbf{y}|\mathbf{x})\right] \mathrm{d}\mathbf{y} \mathrm{d}\mathbf{x} \tag{5.36}$$

$$= -\iint p(\mathbf{x}, \mathbf{y}) \ln p(\mathbf{x}) \mathrm{d}\mathbf{y} \mathrm{d}\mathbf{x} - \iint p(\mathbf{x}, \mathbf{y}) \ln p(\mathbf{y}|\mathbf{x}) \mathrm{d}\mathbf{y} \mathrm{d}\mathbf{x} \text{ , or} \tag{5.37}$$

$$= \mathrm{H}[\mathbf{x}] + \mathrm{H}[\mathbf{y}|\mathbf{x}] \tag{5.38}$$

where $\mathrm{H}[\mathbf{x}]$ follows from the marginalisation of $p(\mathbf{x}) = \int p(\mathbf{x}, \mathbf{y}) \mathrm{d}\mathbf{y}$.

5.8.1 Kullback-Leibler (KL) divergence

The conditional entropy relates the information content of two variables under their *real* probability distributions. In practice, however, these are never known. It is, therefore, important to be able to quantify

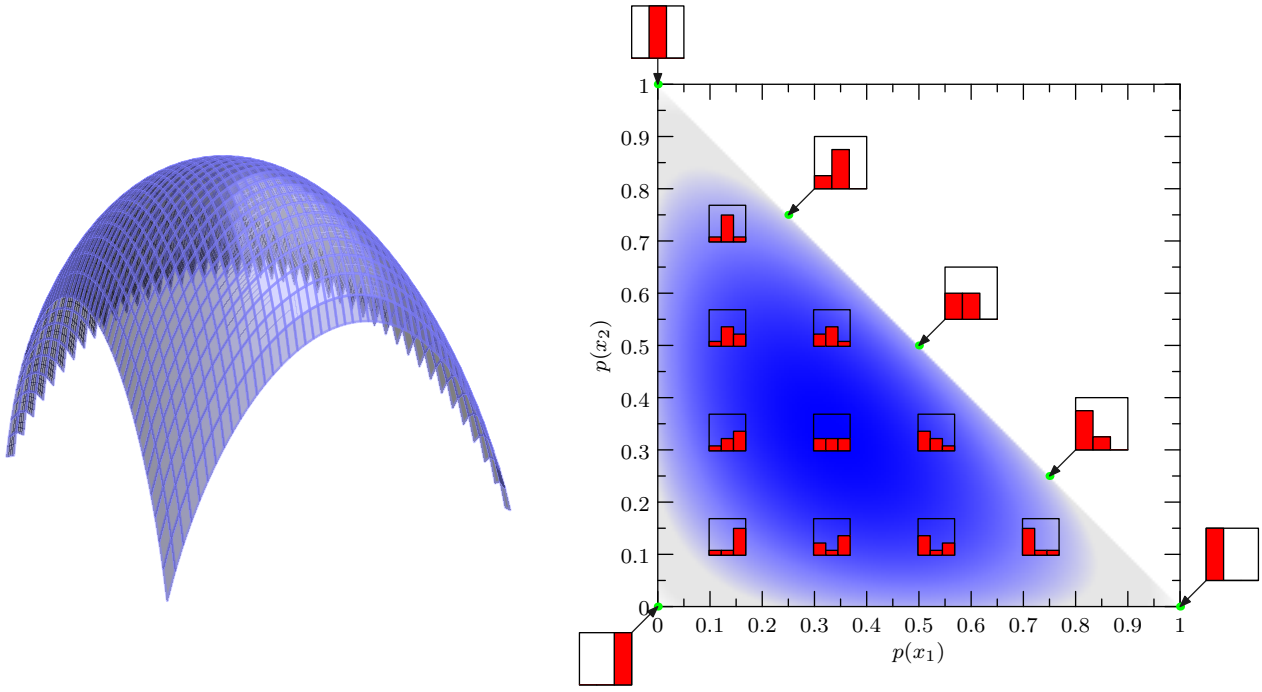


Figure 5.8: Plot of the entropy for a discrete random variable with three states. Both plots show the entropy (in 3 dimensions and as a colour gradient) as a function of $p(X = x_1)$ and $p(X = x2)$, while $p(X = x_3) = 1 - (p(x_1) + p(x_2))$. The resulting input space is a triangle, where each edge corresponds to one of the variables having probability zero (The axis labelled with $p(x_1)$ corresponds to $p(x_2) = 0$ and $p(x_3) = 1 - p(x_1)$, the diagonal edge corresponds to $p(x_3) = 0$. The ragged edge in the plot on the left hand side is an artefact of the plotting, not of the function itself.

the amount of information wasted on when using some other probability distribution instead, in order to minimise this waste. Where the *conditional entropy* relates two variables under their joint distribution, the *relative entropy* or *KL divergence* relates two distributions over a single set of variables. The conditional entropy will therefore be used in practice to measure how much information one variable provides about another (for example, how much information a feature provides about the class label we want to predict), while the KL divergence will be used to evaluate how similar one distribution is to another. This KL divergence will be particularly in chapter 7, when we learn distributions over variables which are not observed.

If we know the real distribution of a variable $\mathbf{x}$, we can compute how much information will need to transmit per measurement, on average, to specify the real value of the variable. If we don't know this real distribution, we cannot create an optimal coding scheme and we will, therefore, waste information when transmitting the real values. The Kullback-Leibler divergence measures how much information we waste by using a distribution $q(\mathbf{x})$ rather than the true distribution $p(\mathbf{x})$:

$$\mathrm{KL}(p||q) = -\int p(\mathbf{x}) \ln q(\mathbf{x}) \mathrm{d}\mathbf{x} - (-\int p(\mathbf{x}) \ln p(\mathbf{x}) \mathrm{d}\mathbf{x}) \tag{5.39}$$

$$= -\int p(\mathbf{x}) \ln \left(\frac{q(\mathbf{x})}{p(\mathbf{x})} \right) \mathrm{d}\mathbf{x} \tag{5.40}$$

It is important to note that this quantity is always positive: it is impossible to have more information, and therefore impossible to perform better at the task at hand (over a large number of measurements) than by knowing the real distribution of the data. The non-negativity of the KL-divergence is easy to show using Jensen's inequality (see Appendix D): since $-\ln(x)$ is a convex function, we have that

$$\mathrm{KL}(p||q) = -\int p(\mathbf{x}) \ln \left(\frac{q(\mathbf{x})}{p(\mathbf{x})} \right) \mathrm{d}\mathbf{x} \tag{5.41}$$

$$\leqslant -\ln \int p(\mathbf{x}) \left[\frac{q(\mathbf{x})}{p(\mathbf{x})} \right] \mathrm{d}\mathbf{x} = -\ln \int q(x) \mathrm{d}\mathbf{x} = -\ln 1 = 0 \tag{5.42}$$

5.9 Mutual information

It is often useful to quantify how informative one random variable is of another. For example, it is useful to know how informative a feature is of the class label. We can formalise this as follows: two random variables $\mathbf{x}$ and $\mathbf{y}$ are completely non-informative of each other if they are independent, *i.e.* $p(\mathbf{x}, \mathbf{y}) = p(\mathbf{x})\, p(\mathbf{y})$. A measure of how informative they are of each other is therefore the divergence of the joint probability distribution $p(\mathbf{x}, \mathbf{y})$ and the factorised distribution. This is called the mutual information between $\mathbf{x}$ and $\mathbf{y}$:

$$\mathbf{I}[\mathbf{x}, \mathbf{y}] = \mathrm{KL}(p(\mathbf{x})\, p(\mathbf{y}) \,||\, p(\mathbf{x}, \mathbf{y})) \tag{5.43}$$

which we can express in terms of entropy as follows:

$$= -\int p(\mathbf{x}, \mathbf{y}) \ln \left(\frac{p(\mathbf{x})\, p(\mathbf{y})}{p(\mathbf{x}, \mathbf{y})} \right) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \tag{5.44}$$

$$= -\int p(\mathbf{x}, \mathbf{y}) \ln p(\mathbf{x}) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} + \int p(\mathbf{x}, \mathbf{y}) \ln p(\mathbf{x}|\mathbf{y}) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \tag{5.45}$$

$$= -\int p(\mathbf{x}) \ln \mathbf{x} \mathrm{d}\mathbf{x} \int p(\mathbf{y}|\mathbf{x}) \mathrm{d}\mathbf{y} + \int p(\mathbf{x}, \mathbf{y}) \ln p(\mathbf{x}|\mathbf{y}) \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \tag{5.46}$$

$$= H[\mathbf{x}] - H[\mathbf{x}|\mathbf{y}] \tag{5.47}$$

Figure 5.9 illustrates how the conditional entropy and the mutual information evolve as a function of the distributions of $\mathbf{x}$ and $\mathbf{y}$, for two binary variables.

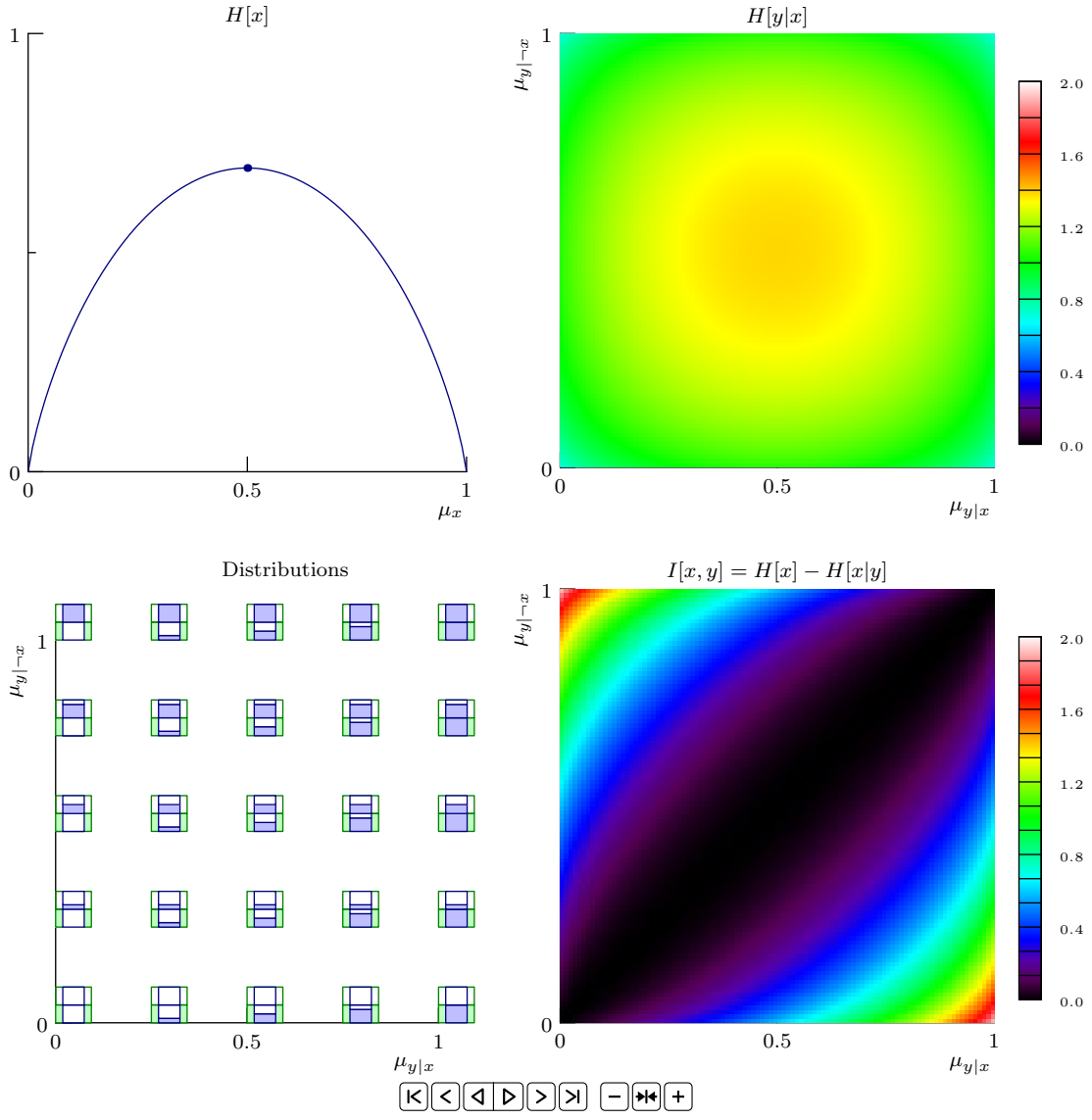


Figure 5.9: Illustration of conditional entropy and mutual information for two binary variables.

Chapter 6

Graphical Models

We have seen, when first treating probabilistic models, that we are interested in learning the joint probability distribution of different variables, in order to predict the value of some variables (say, class labels) when knowing the value of other variables (say, the observations.) In general, this distribution will become very complex when many variables are present in the model, such as when we are dealing with high-dimensional data (images, for example), or when we are dealing with data series. In such cases, it is important to analyse what variables are (conditionally) independent, as this allows us to drastically simplify the joint probability distribution.

Graphical models provide us with an intuitive depiction of the independence assumptions between the different variables in the model, and give us automated algorithms to efficiently perform computations with these variables. Graphical models have become very mainstream in machine learning, partially because many previously existing models have been shown to be specific examples of graphical models, but also because they have been instrumental in improving our insights in existing and novel techniques.

In this chapter, we introduce graphical models: we see how to interpret and build a graphical model, what information we can extract from such a model and how we can use such a model and the associated probability distributions to obtain information about unknown variables. Later, we will also see how we can learn those associated probability distributions.

6.1 Bayesian Networks

Graphical models come in two flavours: directed and undirected graphical models. Bayesian Networks (BN) are the directed type. In these graphs, nodes represent random variables and edges represent dependence relations. In Bayesian networks, the directed edges are meant to represent *causal* relations,[1] because doing

[1] Causality (the fact that one event causes another) is a fascinating concept that has subtle implications and is often avoided in statistics handbooks (except to say that there is no causation, only correlation). In fact, causality can be investigated statistically, but only if we can intervene in the system, not if we can only measure what is happening. For a fascinating read on the subject, see Pearl [2000].

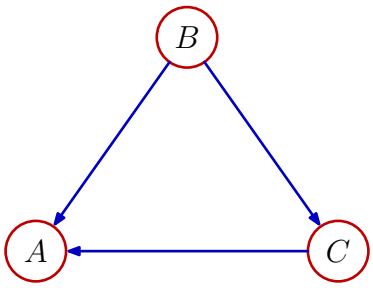


Figure 6.1: A simple graphical network

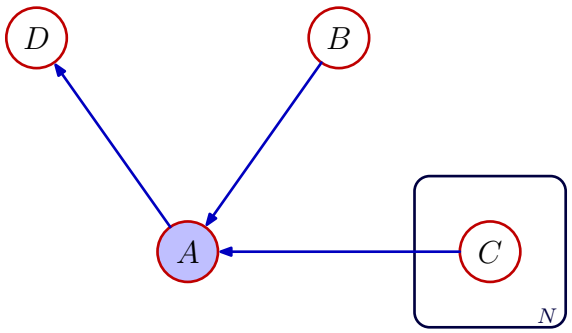


Figure 6.2: A simple graphical network illustrating the use of plates and shaded nodes

so typically leads to more compact representations of the associated distributions.

Let us consider a simple example to expose the different elements that build a graph. In this scenario, we consider the random variable A, which denotes whether you are Attending the MLPR lecture on graphical models or not. You are a student in Amsterdam, which means that you do all your displacements by bike: if your bike is Broken, denoted by B, you need to find alternative means of transportation, which means that you may not make it to the lecture. The probability of A therefore depends on B. However, whether you attend this early-morning lecture also depends on what you Consumed last night, denoted C, and this depends on whether you had the opportunity to go to your favourite coffee shop, so that C also depends on the state of your bike B. The graphical model corresponding to this scenario is depicted in Figure 6.1, and this graph indicates the factorisation of our probability distribution:

$$p(A, B, C) = p(A|B, C)\, p(C|B)\, p(B) \tag{6.1}$$

Notice that, in this case, the factorisation does not imply any independence among the variables: we could have obtained this very same factorisation by simply applying the product rule of probabilities to the joint probability distribution. We could also have chosen a different factorisation (say, $p(C|A, B)\, p(B|A)\, P(A)$) and this would have represented the same joint distribution. And here I would like to highlight two points:

1. The causality implied by the arrows does not modify the distribution of the variables, and making erroneous assumptions on the direction of the edges does not modify the joint distribution. It may, however, affect the complexity of the resulting conditional distributions (causal relations tend lead to much simpler distributions than their inverse) and, therefore, the affect the amounts of training data required to learn them.

2. The fact that we model $p(C|B)$ rather than $p(B|C)$ does not mean that B is independent of C. Indeed, if a broken bike makes it less likely that you'll go to the coffee shop, knowing that you went to the coffee shop will tell me that it's less likely that your bike is broken.

Observed variables are denoted by shaded nodes. In this case, let us imagine that I can tell whether you're present for the lecture, so that node A is shaded. Furthermore, let us assume that you could have consumed more than one thing, last night, and that, for the sake of the example, consuming one product does not affect whether you consume another. Then, instead of encoding a variable C which can take on any of a number of values $c_1, \ldots, c_n$ corresponding to a number of substances, we would have to use multiple variables $C_1, \ldots, C_n$ which can each take the value 0 or 1, denoting whether or not you consumed that particular substance. Such repeated variables can be represented compactly with plates, as depicted in Figure 6.2. If we now make the additional assumption that you are very dedicated, so that the state of your bike does not in any way affect your capacity to find a coffee shop (it still does affect your capacity to come to the lecture, though), and further assume that whether you succeed for this course and obtain your Degree (D) depends on whether you attended this lecture (A), we obtain the graphical model shown in Figure 6.2.

This example is not fully connected anymore, and contains simplifying assumptions. For example, if I know that you came to the lecture, I can compute the probability that you'll get your degree. It doesn't

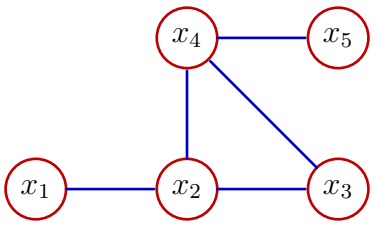


Figure 6.3: Example Markov random field. Here, the probability of the set of variables, $\mathbf{x}$, is given by $p(\mathbf{x}) = \frac{1}{Z}\psi_1(x_1, x_2)\,\psi_2(x_2, x_3, x_4)\,\psi_3(x_4, x_5)$

matter whether your bike is broken, or whether you consumed anything the evening before. A more subtle point is the following: if I know that you didn't come to the lecture, my belief that your bike is broken will increase a little, and my belief that you consumed some substance that your liver had trouble with also increases. The probability increases, for both $B = true$ and $C_n = true$. However, if after the lecture I learn for a fact that your bike is broken, my belief that you consumed anything last night will decrease again (it will still be higher than if you had come to the lecture,[2] but it will be lower than if I'd known that your bike was in perfect working order, and lower than if I didn't know about your bike either way.) In other words, as long as A is unobserved, B and C are independent. This is called *marginal independence*. If A is observed, however, then B and C become dependent, conditionally on A. The phenomenon where two variables which are marginally independent become dependent when a common descendant becomes observed is called "explaining away": in this example, the fact that your bike is broken "explains away" your inability to come to the lecture.

6.1.1 D-Separation

Finding the dependence and independence relationships between variables is, in general quite complicated. The representation of the factorisation as a graph, however, provides us with a simple and powerful algorithm for discovering these relationships automatically. D-separation states that two nodes are independent if all paths connecting the nodes are blocked. A path is blocked, conditioned on the set of observed nodes if:

1. edges meet head-to-tail ($\longrightarrow\!\bigcirc\!\longrightarrow$) or tail-to-tail ($\longleftarrow\!\bigcirc\!\longrightarrow$) at a node which is in the observed set,

2. edges meet head-to-head ($\longrightarrow\!\bigcirc\!\longleftarrow$) at a node which is not, and none of whose descendents is in the observed set

If we apply this to the graph in Figure 6.2, we see that B and C are marginally independent, but become conditionally dependent if either A or D is observed. similarly, B and D are marginally dependent, but they become independent if A is observed.

6.2 Markov Random Fields

Markov Random Fields (MRF) are represented by undirected graphs. In such models, we do not assume causal relations; instead the edges indicate marginal dependence between variables. The dependencies between variables are expressed in the form of *potential functions*: these are nonnegative functions of the variables they are defined over, which do not (necessarily) have a probabilistic interpretation. The name "potential function" stems from the origin of these models in physics. These functions are defined over *cliques* in the graph. Cliques are defined as subsets of the nodes such that all pairs of nodes in the clique have links between them. Maximal cliques are cliques of the graph such that no node could be added without the subset ceasing to be a clique. Potential functions are defined over maximal cliques.

[2] Except in the extreme case that my prior was one to begin with...

The probability of the state $\mathbf{x}$ of the network (*i.e.*, a particular instantiation for all variables in the network) is given by

$$p(\mathbf{x}) = \frac{1}{Z} \prod_C \psi_C(\mathbf{x}_C) \tag{6.2}$$

where Z is called the partition function,[3] and is a scalar that ensures that the probability of all possible states $\mathbf{x}$ sums up to one; $\mathbf{x}_C$ is the set of nodes belonging to clique C and ψ_C is the potential function defined over that clique. The partition function is computed by summing over all possible assignments to $\mathbf{x}$:

$$Z = \sum_{\mathbf{x}} \prod_C \psi_C(\mathbf{x}_C) \tag{6.3}$$

Computing Z is often not possible: the computational complexity grows exponentially (worst case) in the number of nodes in the graph, although in some cases the graph structure, *e.g.* a chain or tree structure, makes computing Z possible. In general, however, computing Z is not always necessary. The probability of two states can be compared, and conditional probabilities can be computed, without computing Z.

6.2.1 Independence

Because the potential functions are undirected, the subtle difficulties stemming from explaining away do not arise in MRF. Detecting whether two (sets of) nodes A and B are independent given C again simply boils down to checking whether all path between the nodes from A and the nodes from B are blocked. A path is blocked if any node on the path belongs to the set of observed variables.

6.3 Factor Graphs

As we have seen, both flavours of graphical models (directed and undirected) specify a factorisation of the joint probability of the variables. Factor graphs are a third, more general type of graphical representation that does not necessarily represent probability distributions. A factor graph is a graphical representation of a function, consisting of edges, variable nodes and factor nodes. The variable nodes denote the value of the variables of the function, a factor node represents any function of the variables it is connected to, and the value of the function is the *product* of the factors in the graph. Factor graphs can, therefore, represent much more than probability distributions and are not by themselves (probabilistic) graphical models although they can, of course, represent any graphical model. Since any graphical model, both directed and undirected, can be represented as a factor graph, and specific algorithms for those two types of graphical models can be seen as special cases of algorithms acting on factor graphs, in the following we shall focus on algorithms for factor graphs only.

6.3.1 Converting Graphical Models to Factor Graphs

Factor graphs are more flexible than graphical models, and in general there can be more than one factor graph that is equivalent with a given graphical model.[4] In the case of directed models, an expedient way to convert the graphical model to a factor graph is to replace each individual probability distribution with a factor. For example the graph depicted in Figure 6.2 corresponds to the factorisation

$$P(A, B, C, D) = p(B)\, p(C)\, P(A|B, C)\, P(D|A) \tag{6.4}$$

We can create an equivalent factor graph, as depicted in Figure 6.4. In this graph, the factors are as follows:

$$f_a(B) = p(B) \tag{6.5}$$
$$f_b(C) = p(C) \tag{6.6}$$
$$f_c(A, B, C) = p(A|B, C) \tag{6.7}$$
$$f_d(A, D) = p(D|A) \tag{6.8}$$

[3] It is a function of the parameters of the potential functions but not of $\mathbf{x}$.

[4] So, I probably won't ask you for the factor graph equivalent of a given graphical model at the exam. In particular, if I ask for the *factorisation* described by a graphical model, please don't draw a factor graph. Give me an equation instead!

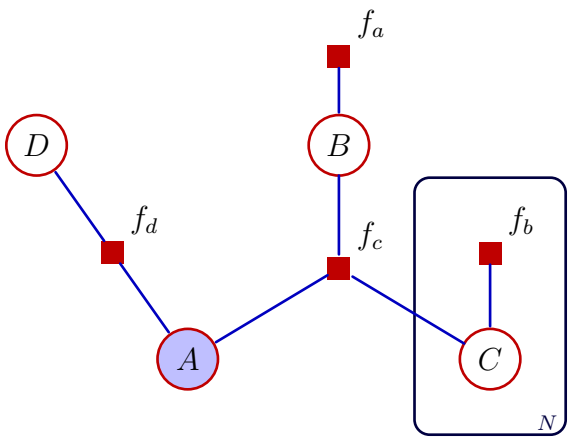


Figure 6.4: A factor graph that is equivalent to the graphical model depicted in Figure 6.2

In the case of undirected graphical models, a good choice is to substitute each potential function with a factor. For undirected models, the conversion can simply be done by assigning one factor node per potential function, but if a potential function is itself the product of two factors, it is also possible to express this in the factor graph by including a factor node for each factor of the potential function.

6.3.2 Sum-product algorithm

The purpose of probabilistic models is to compute the probability distribution of some variables, given that other variables are observed. For example, we might want to know the probability that a flower should be an Iris Versicolor, given her petal and sepal measurements. Computing such conditional probabilities requires the computation of marginal probabilities. In our iris example, we could have modelled[5] the prior probability that an iris should be of a particular variety, $p(V)$, and the probability of the petal and sepal lengths for different varieties of irises, $p(L_p, L_s|V)$. If we are now interested in knowing the probability that an iris should be of a particular variety given the length of its petals and sepals, $p(v|l_p, l_s)$, we use Bayes' rule and compute:

$$p(v|l_p, l_s) = \frac{p(l_p, l_s|v)\, p(v)}{p(l_p, l_s)} \ . \tag{6.9}$$

Notice how all terms in the numerator are known, while the denominator must be computed by marginalisation. In practice, it will often be the case that the variables we are interested in are not the only ones not to be observed. For example, we may want to know the probability that a flower should be an Iris Versicolor, given only its petal length and regardless of its sepal lengths. In that case, we need to marginalise out the other variables as well:

$$p(v|l_p) = \frac{p(l_p|v)\, p(v)}{p(l_p)} \ , \tag{6.10}$$

where we had to marginalise out L_s in both numerator and denominator. The point is, of course, that marginalisation is the key to inference in probabilistic models, and that without efficient marginalisation inference is not possible.

The sum-product algorithm is an automated procedure to compute the marginal probability of a set of variables *exactly, as efficiently as possible* for a given factorisation of the joint probability distribution. In the following, we will first explore how the marginalisation can be done as efficiently as possible and then proceed to developing this as an algorithm on graphs.

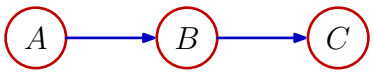


Figure 6.5: Graphical model for the factorisation of $p(A, B, C) = p(A)\, p(B|A)\, p(C|B)$

Efficient inference in factorised distributions

Consider a system with three binary variables, A, B and C, where the joint probability of the variables, $p(A, B, C)$ is factorised as $p(A)\, p(B|A)\, p(C|B)$. This factorisation is shown graphically in Figure 6.5. If we wanted to compute the marginal probability $p(B)$, since the variables are binary, we could compute

$$p(B) = p(a, B, c) + p(a, B, \neg c) + p(\neg a, B, c) + p(\neg a, B, \neg c) \tag{6.11}$$

where, following our convention of using capital letters for the variables and minuscules for instantiations, $p(B)$ denotes the probability of any of B's possible instantiations. We're using binary variables in this example, and so $p(B)$ could be seen as a vector containing the values $(p(b) \;\; p(\neg b))^\top$, but the argument holds in general and so if B was a continuous variable, for example, $p(B)$ would be a PDF encoding the probability density for any of the possible value of the variable.

Notice how, in order to marginalise out two binary variables, we use four summations. In general, this summation is exponential in the number of variables we are marginalising out. However, because according to our factorisation, C is independent from A given B, we could compute our marginalisation as

$$\begin{aligned} p(B) &= p(a)\, p(B|a)\, [p(c|B) + p(\neg c|B)] + p(\neg a)\, p(B|\neg a)\, [p(c|B) + p(\neg c|B)] && (6.12)\\ &= [p(a)\, p(B|a) + p(\neg a)\, p(B|\neg a)]\, [p(c|B) + p(\neg c|B)] && (6.13) \end{aligned}$$

which requires only two summations. We used the distributivity of the addition and the product (*i.e.*, $ab + ac = a(b + c)$) to minimise the number of operations needed in computing the marginal probability. It is this, the order in which we perform the computations, which is the key to efficient marginalisation.

Efficient marginalisation on factor graphs

We will describe the sum-product algorithm with our example depicted in Figure 6.4. The formal derivation and proof of the validity of the algorithm can be found in Section 8.4.4 of Bishop's book; here we focus on how to apply it, and show how it does indeed provide us with the desired quantities in the case at hand.

Suppose we want to know how likely it is for you to attend this lecture, $p(A)$. This probability is given by

$$p(A) = \sum_b \left[p(B = b) \sum_{c_1} \cdots \sum_{c_N} \prod_n [p(C_n = c_n)]\, p(A|B = b, C_1 = c_1 \ldots C_N = c_N) \right] \sum_d p(D = d|A) \tag{6.14}$$

This formula may seem pretty unfriendly, but let's take a look at what it does. We take the sum over all possible values of B, $C_1, \ldots, C_N$ and D of $p(A, B, C_1, \ldots, C_N, D)$, thus simply applying the sum rule of probabilities to obtain the marginal probability $p(A)$. But we do it in such a way, that every factor that does not depend on the variable that we are summing over is, that is, every variable that can be taken out of the sum, is indeed taken out. So, for example $p(B)$ depends only on B, but $p(A|B, C_1, \ldots, C_N)$ depends on $B, C_1, \ldots, C_N$. The sum-product algorithm provides a way to find this efficient order of the terms in the equation mechanically, by passing messages over in the graph.

Application of the sum-product algorithm

Each node can transmit a message on one of its links when it has received messages on all of its other links. The different types of nodes perform different actions:

[5] Remember that by convention uppercase letters denote random variables and lowercase letters denote instantiations.

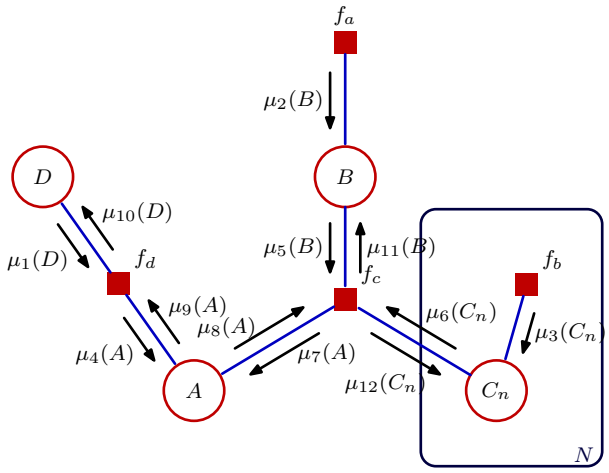


Figure 6.6: Messages passed in the factor graph for the sum-product algorithm.

A factor node takes the sum, over the possible values of the variable nodes that it receives messages from, of the product of those messages with the function it represents. The message it transmits is therefore a function of the variable it is sending to.

A Variable node takes the product of the incoming messages and transmits that. The message it transmits is therefore a function of the value of the variable itself.

Let us apply this to our example graph, where the messages are depicted in Figure 6.6. The algorithm starts at the leaves of the graph. These nodes have only one link, and can therefore transmit a message immediately. In this case, those are nodes D, f_a and f_b. Since these nodes have no incoming messages, the messages they transmit are reduced to:

$$\mu_1(D) = [1, 1] \tag{6.15}$$
$$\mu_2(B) = [p(b), p(\neg b)] \tag{6.16}$$
$$\mu_3(C_n) = [p(c_n), p(\neg c_n)] \tag{6.17}$$

The messages that arrive in factor f_d get multiplied, multiplied with $f_d(A, D)$ and summed over the possible values of D, while the messages arriving in B and C simply get multiplied. Since in all cases only one message is received, the resulting outgoing messages are:

$$\mu_4(A) = [1p(d|a) + 1p(\neg d|a), 1p(d|\neg a) + 1p(\neg d|\neg a)] \tag{6.18}$$
$$= [1, 1] \tag{6.19}$$
$$\mu_5(B) = [p(b), p(\neg b)] \tag{6.20}$$
$$\mu_6(C_n) = [p(c), p(\neg c)] \tag{6.21}$$

Now f_c takes its incoming messages, multiplies them, multiplies them with its factor and sums over the values of its incoming variables. The result is:

$$\mu_7(A) = \begin{bmatrix} \sum_{B=b,\neg b} p(B) \sum_{C_1} \cdots \sum_{C_n} p(C_1) \cdots p(C_n)\, p(a|B, C_1, \ldots, C_n) \\ \sum_{B=b,\neg b} p(B) \sum_{C_1} \cdots \sum_{C_n} p(C_1) \cdots p(C_n)\, p(\neg a|B, C_1, \ldots, C_n) \end{bmatrix} \tag{6.22}$$
$$= [p(a), p(\neg a)] \tag{6.23}$$

The result is then given by the product of the incoming messages,

$$p(A) = \mu_4(A)\, \mu_7(A) = p(A) \tag{6.24}$$

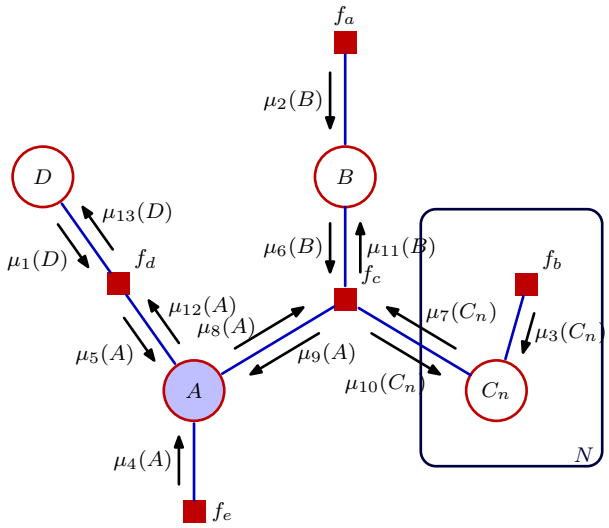


Figure 6.7: Messages passed in the factor graph for the sum-product algorithm, where $A = a$ is observed.

However, the marginal probability of all other variables can be found by keeping propagating messages, indicated by messages $\mu_8, \ldots, \mu_{12}$:

$$\mu_8(A) = [1, 1] \tag{6.25}$$
$$\mu_9(A) = [p(a), p(\neg a)] \tag{6.26}$$
$$\mu_{10}(D) = [p(a)p(d|a) + p(\neg a)p(d|\neg a), p(a)p(\neg d|a) + p(\neg a)p(\neg d|\neg a)] \tag{6.27}$$
$$= [p(d), p(\neg d)] \tag{6.28}$$
$$\mu_{11}(B) = \begin{bmatrix} \sum_A \sum_{C_1} \cdots \sum_{C_n} \prod p(C_n)p(A|b, C_1, \ldots, C_N) \\ \sum_A \sum_{C_1} \cdots \sum_{C_n} \prod p(C_n)p(A|\neg b, C_1, \ldots, C_N) \end{bmatrix} \tag{6.29}$$
$$= [1, 1] \tag{6.30}$$
$$\mu_{12}(C_n) = \begin{bmatrix} \sum_A \sum_B \sum_{C \neq C_n} \prod_{C \neq C_n} p(C)p(A|B, C_1, \ldots, C_n, \ldots, C_N) \\ \sum_A \sum_B \sum_{C \neq C_n} \prod_{C \neq C_n} p(C)p(A|B, C_1, \ldots, \neg C_n, \ldots, C_N) \end{bmatrix} \tag{6.31}$$
$$= [1, 1] \tag{6.32}$$
$$\tag{6.33}$$

so that the product of the incoming messages, in each variable node, results in the marginal probability of that variable.

Dealing with observed variables

If some variables are observed, we simply introduce an extra factor connected to those variables only, which is one for the observed value of the variable and zero otherwise. The rest of the algorithm is identical and provides us, for each latent variable, with the marginal *joint* probability of that latent variable and the observations. Figure 6.7 illustrates what the propagated messages become if I observe that you attended the

lecture, where the messages take the following values:

$$\mu_1(D) = [1, 1] \tag{6.34}$$
$$\mu_2(B) = [p(b), p(\neg b)] \tag{6.35}$$
$$\mu_3(C_n) = [p(c_n), p(\neg C_n)] \tag{6.36}$$
$$\mu_4(A) = [1, 0] \tag{6.37}$$
$$\mu_5(A) = [1, 1] \tag{6.38}$$
$$\mu_6(B) = [p(b), p(\neg b)] \tag{6.39}$$
$$\mu_7(C_n) = [p(c), p(\neg c)] \tag{6.40}$$
$$\mu_8(A) = [1 \times 1, 1 \times 0] = [1, 0] \tag{6.41}$$
$$\mu_9(A) = [p(a), p(\neg a)] \tag{6.42}$$
$$\mu_{10}(C_n) = \begin{bmatrix} \sum_B \sum_{C \neq C_n} \prod_{C \neq C_n} p(C) p(a|B, C_1, \ldots, C_n, \ldots, C_N) \\ \sum_B \sum_{C \neq C_n} \prod_{C \neq C_n} p(C) p(a|B, C_1, \ldots, \neg C_n, \ldots, C_N) \end{bmatrix} \tag{6.43}$$
$$= [p(a|C_n), p(a|\neg C_n] \tag{6.44}$$
$$\mu_{11}(B) = \begin{bmatrix} \sum_{C_1} \cdots \sum_{C_n} \prod p(C_n) p(a|b, C_1, \ldots, C_N) \\ \sum_{C_1} \cdots \sum_{C_n} \prod p(C_n) p(a|\neg b, C_1, \ldots, C_N) \end{bmatrix} \tag{6.45}$$
$$= [p(a|b), p(a|\neg b)] \tag{6.46}$$
$$\mu_{12}(A) = [p(a), 0] \tag{6.47}$$
$$\mu_{13}(D) = [p(d|a)\, p(a) + p(d|\neg a)\, 0, p(\neg d|a)\, p(a) + p(\neg d|a)\, 0] \tag{6.48}$$
$$= [p(a, d), p(a, \neg d)] \tag{6.49}$$

From the marginal distributions that we obtained, $p(a, D), p(a, B), p(a, C_n)$, we can compute conditional probabilities using Bayes' theorem. For example,

$$p(d|a) = \frac{p(a, d)}{p(a)} = \frac{\mu_{13}(d)}{\mu_{12}(a)} \tag{6.50}$$

6.3.3 max-sum algorithm

The sum-product algorithm provides us with an efficient way to compute the marginal probability of a variable. Sometimes, however, we are not interested in the conditional probability of the latent variables given the observations; rather, we would like to know what the *most* likely, *joint* state of these latent variables is. For example, I may not be interested to know what the probability is that your bike is broken, or what the probability is that you enjoyed some dodgy substance or the other. Instead, I may like to know what the most likely *combination* of factors caused you not to come to a lecture. This result is obtained by the max-sum algorithm.

The max-sum algorithm is basically the same algorithm as the sum product algorithm. The same sequence of messages is propagated, where the probabilities are replaced by log probabilities, the products by sums and sums by max operators. For each node, this provides us with the information of: (1) what the most likely state of that node is, given the most likely states of all other nodes in the graph, (2) what the probability is of that state of the graph, and (3) what state of its neighbours brought it to this most likely state. Once the last node in the graph has received a message on all its links, we then backtrack to store, for each node, what state of that variable resulted in the most probable state of the total system.

I'd work this out for our example, but since it's really the same as before, and since I feel rather woozy from all the that I've been having myself, I think I'll leave that for you to do as an exercise.

6.4 Dynamic Bayesian Networks

Until now, we have been assuming that the number of variables in the system was known and fixed beforehand. Sometimes, however, it is useful to consider systems where the number of variables is dynamic. For

example, we could consider time sequences, where the number of observations increases over time, or models of DNA sequences, where the number of base pairs that constitute a gene is not known beforehand. Such models of sequential data are just graphical models, but because of their interest and specific properties, they deserve to be considered explicitly. These models have been described long before graphical models themselves were developed as a paradigm, and the same algorithms used in graphical models (sum-product and max-sum) are known by different names in the case of sequential models, namely as the Forward-Backward and Viterbi algorithms.

Coming up...

Chapter 7

Learning in Graphical Models

In the previous chapter, we have seen how we could use the factorisation of the joint probability distribution to compute the marginal probability of variables efficiently. We have also seen automated procedures on the graph, which allow us to find the order in which to do the computations with optimal efficiency. To compute these marginal probabilities, the joint probability distribution had to be known. More specifically, the parameters of the functions describing the factorised probability distributions (or pdfs) were known.

Due to the complexity of the models that are typically described by graphical models, it is often not possible to integrate out the parameters of these distributions and the parameters are then learnt by maximum likelihood or maximum a posteriori learning. In this chapter, we investigate how this can be done. We use an example model, the Gaussian mixture model (GMM) to illustrate the techniques we develop, but of course these techniques are more widely applicable.

7.1 Fully observed model

Let us first consider a fully supervised classification problem. We are given the data depicted in Figure 7.1, where both the measurements $\mathbf{x}_i$ and the class labels z_i are given. We want to model the distribution of the data for each class independently, and choose to use a Gaussian distribution to do so. So, for each datapoint, we want the joint probability of that datapoint and its label to be given by

$$p(\mathbf{x}_i, z_i) = \begin{cases} p(\mathcal{C}_1)p(\mathbf{x}_i|\mathcal{C}_1) & \text{if } z_i = 1 \\ p(\mathcal{C}_2)p(\mathbf{x}_i|\mathcal{C}_2) & \text{otherwise (That is, if } z_i = 0) \end{cases} \tag{7.1}$$

where:

$$p(\mathcal{C}_1) = \rho \tag{7.2}$$
$$p(\mathcal{C}_2) = 1 - \rho \tag{7.3}$$
$$p(\mathbf{x}_i|\mathcal{C}_1) = \mathcal{N}(\mathbf{x}_1; \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) \tag{7.4}$$
$$p(\mathbf{x}_i|\mathcal{C}_2) = \mathcal{N}(\mathbf{x}_1; \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2) \tag{7.5}$$

The corresponding graphical model is given in Figure 7.2. Thanks to our clever encoding of the labels, we can use the same trick as in the Bernoulli distribution, and raise the relevant probability to the power of z_i. The likelihood is then given by:

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N, z_1, \ldots, z_N|\theta) = \prod_{i=1}^{N} \left(\rho\,\mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)\right)^{z_i} \left((1-\rho)\,\mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)\right)^{1-z_i} \tag{7.6}$$

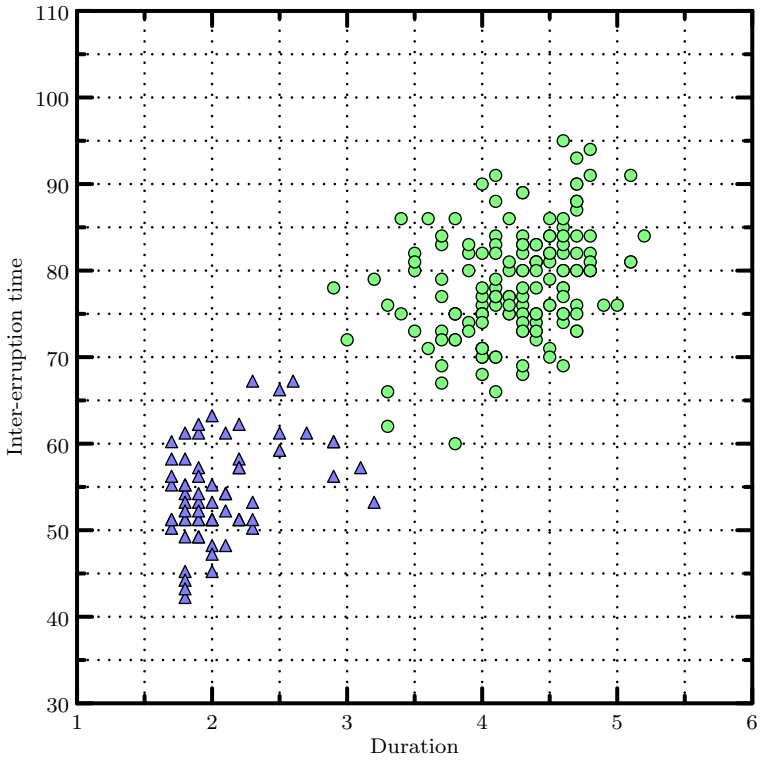


Figure 7.1: Supervised classification of continuous, 2-dimensional data

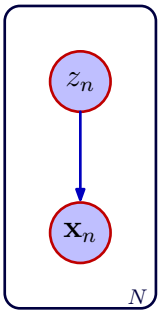


Figure 7.2: Graphical model for supervised classification

where, again, $z_i = 1$ if $\mathbf{x}_i \in \mathcal{C}_1$ and $z_i = 0$ otherwise, $\rho = p(\mathcal{C}_1)$ is the prior probability that a datapoint belongs to class $\mathcal{C}_1$[1], and $\mathcal{N}(\cdot)$ indicates the Gaussian PDF:

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{d/2}|\boldsymbol{\Sigma}|^{1/2}} \exp[-\tfrac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})] \ . \tag{7.7}$$

Notice that this can be generalised to more than two classes, by using a 1-of-N encoding for the class label $\mathbf{z}_i$. To find the maximum likelihood parameters, we take the logarithm of the likelihood, take the derivative with respect to the parameters we want to use and set this derivative equal to zero. Below, I will derive the solution in a lot of detail. This serves two purposes: (1) to show you how it's done; the same technique is used for any graphical model, and (2) I am hoping that you will follow the steps, and that they will seem understandable and perhaps even vaguely familiar, and that it may help you to see a page of scary equations and look at it, and understand it, without skipping over it. The purpose is explicitly *not* to make the page look scary.[2]

Filling in Eq. (7.7) in Eq. (7.6) and taking the logarithm results in the log-likelihood (denoted $\ell(\theta)$, because the data is considered fixed and the parameters are not):

$$\ell(\theta) = \sum_{i=1}^{N} z_i \left(\ln\rho - \tfrac{1}{2}\left(d\ln 2\pi + \ln|\boldsymbol{\Sigma}_1|\right) - \tfrac{1}{2}(\mathbf{x}_i-\boldsymbol{\mu}_1)^\top\boldsymbol{\Sigma}_1^{-1}(\mathbf{x}_i-\boldsymbol{\mu}_1)\right) + (1-z_i)\left(\ln(1-\rho) - \tfrac{1}{2}\left(d\ln 2\pi + \ln|\boldsymbol{\Sigma}_2|\right) - \tfrac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_2)^\top\boldsymbol{\Sigma}_2^{-1}(\mathbf{x}-\boldsymbol{\mu}_2)\right) \tag{7.8}$$

Taking the first derivative with respect to ρ and setting it equal to zero results in:

$$\sum_{i=1}^{N} z_i\frac{1}{\rho} + (1-z_i)\frac{1}{1-\rho}(-1) = 0 \quad \Longrightarrow \quad \sum_{i=1}^{N}\frac{z_i(1-\rho)-(1-z_i)\rho}{\rho(1-\rho)} = 0 \quad \Longrightarrow \tag{7.9}$$

$$\sum_{i=1}^{N} z_i - \rho\sum_{i=1}^{N} z_i - \rho\sum_{i=1}^{N}(1-z_i) = 0 \quad \Longrightarrow \quad \rho = \frac{N_{\mathcal{C}_1}}{N_{\mathcal{C}_1}+N_{\mathcal{C}_2}} \tag{7.10}$$

where we have introduced $N_{\mathcal{C}_1} = \sum_{i=1}^{N} z_i$ and $N_{\mathcal{C}_2} = \sum_{i=1}^{N}(1-z_i)$, the number of datapoints in class $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively.

To take the first derivative with respect to the mean, we call Petersen and Pedersen [2006] to the rescue. We get:

$$\frac{\partial\ell(\theta)}{\boldsymbol{\mu}_1} = \sum_{i=1}^{N} z_i\boldsymbol{\Sigma}_1^{-1}(\mathbf{x}_i-\boldsymbol{\mu}_1) \tag{7.11}$$

$$\frac{\partial\ell(\theta)}{\boldsymbol{\mu}_2} = \sum_{i=1}^{N}(1-z_i)\boldsymbol{\Sigma}_2^{-1}(\mathbf{x}_i-\boldsymbol{\mu}_2) \tag{7.12}$$

(using (79) in the matrix cookbook) and

$$\tag{7.13}$$

$$\frac{\partial\ell(\theta)}{\boldsymbol{\Sigma}_1^{-1}} = -\tfrac{1}{2}\sum_{i=1}^{N} z_i\left(\boldsymbol{\Sigma}_1 - \boldsymbol{\Sigma}_1^{-1}(\mathbf{x}_i-\boldsymbol{\mu}_1)(\mathbf{x}_i-\boldsymbol{\mu}_1)^\top\boldsymbol{\Sigma}_1^{-1}\right) \tag{7.14}$$

$$\frac{\partial\ell(\theta)}{\boldsymbol{\Sigma}_2^{-1}} = -\tfrac{1}{2}\sum_{i=1}^{N}(1-z_i)\left(\boldsymbol{\Sigma}_2 - \boldsymbol{\Sigma}_2^{-1}(\mathbf{x}_i-\boldsymbol{\mu}_2)(\mathbf{x}_i-\boldsymbol{\mu}_2)^\top\boldsymbol{\Sigma}_2^{-1}\right) \tag{7.15}$$

[1] We are using ρ for this rather than the more conventional π, to avoid confusion with the value of $\pi = 3.1415\ldots$ that crops up in the Gaussian distribution.

[2] Also, as a side note, always remember that you're smarter than me. I've had more time to process things than you have, but you're younger and can process new stuff faster than I can.

using (51) and (55) in the same text, where we have used that the covariance matrix is symmetric, so that $\boldsymbol{\Sigma}^\top = \boldsymbol{\Sigma}$. Setting Eq. (7.11) equal to zero, we get the matrix equivalent of the familiar computation of the data mean:

$$\begin{aligned}
\sum_{i=1}^N z_i \boldsymbol{\Sigma}^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_1) = 0 \quad &\Longrightarrow \quad \boldsymbol{\Sigma}^{-1}\sum_{i=1}^N z_i(\mathbf{x}_i - \boldsymbol{\mu}_1) = 0 \quad \Longrightarrow \\
\sum_{i=1}^N z_i(\mathbf{x}_i - \boldsymbol{\mu}_1) = 0 \quad &\Longrightarrow \quad \sum_{i=1}^N z_i\mathbf{x}_i = \sum_{i=1}^N z_i\boldsymbol{\mu}_1 \quad \Longrightarrow \\
\sum_{i=1}^N z_i\mathbf{x}_i = \sum_{i=1}^N z_i\boldsymbol{\mu}_1 \quad &\Longrightarrow \quad \boldsymbol{\mu}_1 = \frac{\sum_{i=1}^N z_i\mathbf{x}_i}{\sum_{i=1}^N z_i}
\end{aligned} \tag{7.16}$$

In other words: $\boldsymbol{\mu}_1$ is the data mean of the datapoints that belong to $\mathcal{C}_1$. The same derivation can be done for $\boldsymbol{\mu}_2$, and results in the mean of the datapoints from class $\mathcal{C}_2$.

Similarly, setting Eq. (7.14) equal to zero results in the familiar equation for the covariance matrix:

$$\begin{aligned}
-\tfrac{1}{2}\sum_{i=1}^N z_i\left(\boldsymbol{\Sigma}_1^{-1} - \boldsymbol{\Sigma}_1^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_1)(\mathbf{x}_i - \boldsymbol{\mu}_1)^\top\boldsymbol{\Sigma}_1^{-1}\right) = 0 \quad &\Longrightarrow \\
\sum_{i=1}^N z_i\boldsymbol{\Sigma}_1^{-1} = \sum_{i=1}^N z_i\boldsymbol{\Sigma}_1^{-1}(\mathbf{x}_i - \boldsymbol{\mu}_1)(\mathbf{x}_i - \boldsymbol{\mu}_1)^\top\boldsymbol{\Sigma}_1^{-1} \quad &\Longrightarrow \\
\boldsymbol{\Sigma}_1^{-1}\sum_{i=1}^N z_i = \boldsymbol{\Sigma}_1^{-1}\left[\sum_{i=1}^N z_i(\mathbf{x}_i - \boldsymbol{\mu}_1)(\mathbf{x}_i - \boldsymbol{\mu}_1)^\top\right]\boldsymbol{\Sigma}_1^{-1} \quad &\Longrightarrow \\
\boldsymbol{\Sigma}_1\left[\boldsymbol{\Sigma}_1^{-1}\sum_{i=1}^N z_i\right]\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_1\boldsymbol{\Sigma}_1^{-1}\left[\sum_{i=1}^N z_i(\mathbf{x}_i - \boldsymbol{\mu}_1)(\mathbf{x}_i - \boldsymbol{\mu}_1)^\top\right]\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\Sigma}_1 \quad &\Longrightarrow \\
\boldsymbol{\Sigma}_1\sum_{i=1}^N z_i = \sum_{i=1}^N z_i(\mathbf{x}_i - \boldsymbol{\mu}_1)(\mathbf{x}_i - \boldsymbol{\mu}_1)^\top \quad &\Longrightarrow \\
\boldsymbol{\Sigma}_1\sum_{i=1}^N z_i = \sum_{i=1}^N z_i(\mathbf{x}_i\mathbf{x}_i^\top - \mathbf{x}_i\boldsymbol{\mu}_1^\top - \boldsymbol{\mu}_1\mathbf{x}_i^\top + \boldsymbol{\mu}_1\boldsymbol{\mu}_1^\top) \quad &\Longrightarrow \\
\boldsymbol{\Sigma}_1 = \frac{\sum_{i=1}^N z_i\mathbf{x}_i\mathbf{x}_i^\top}{\sum_{i=1}^N z_i} - \frac{\sum_{i=1}^N z_i\mathbf{x}_i}{\sum_{i=1}^N z_i}\boldsymbol{\mu}_1^\top - \boldsymbol{\mu}_1\frac{\sum_{i=1}^N z_i\mathbf{x}_i^\top}{\sum_{i=1}^N z_i} + \boldsymbol{\mu}_1\boldsymbol{\mu}_1^\top \quad &\Longrightarrow \\
\boldsymbol{\Sigma}_1 = \frac{\sum_{i=1}^N z_i\mathbf{x}_i\mathbf{x}_i^\top}{\sum_{i=1}^N z_i} - \boldsymbol{\mu}_1\boldsymbol{\mu}_1^\top - \boldsymbol{\mu}_1\boldsymbol{\mu}_1^\top + \boldsymbol{\mu}_1\boldsymbol{\mu}_1^\top \quad &\Longrightarrow \\
\boldsymbol{\Sigma}_1 = \frac{1}{\sum_{i=1}^N z_i}\sum_{i=1}^N z_i\mathbf{x}_i\mathbf{x}_i^\top - \boldsymbol{\mu}_1\boldsymbol{\mu}_1^\top
\end{aligned} \tag{7.17}$$

7.2 k-means

Now let us suppose that we are not given any labels for the data. We would still like to model two clusters, but we don't know beforehand which cluster each datapoint should belong to. We could enumerate all possible assignment combinations, compute the parameters as explained above for each combination, compute the resulting likelihood and keep the assignment combination and corresponding parameters that result in the highest likelihood.

This would give us the maximum-likelihood parameters for our model, but it doesn't scale. While in our original model with known labels the datapoints were independent and we could compute the parameters in closed form, now we need to list all possible combinations of datapoint-cluster assignments, which is impossible for even moderately sized datasets. Luckily, although finding the global optimum for the

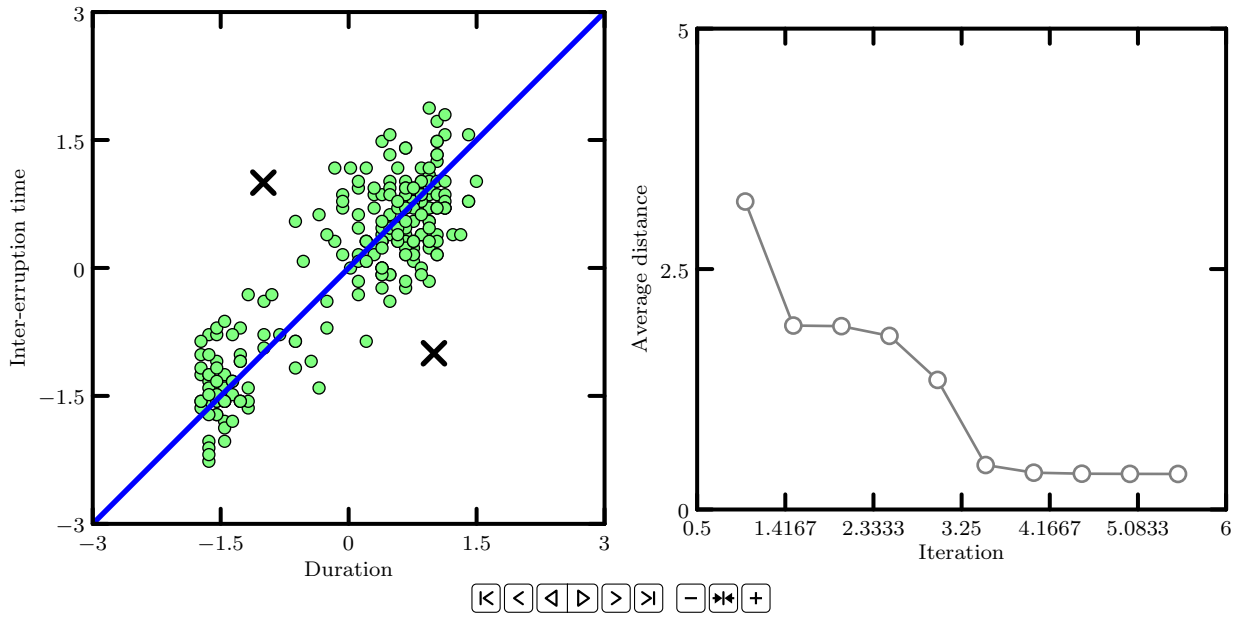


Figure 7.3: Animation illustrating the k-means algorithm on the old-faithful dataset. Five iterations are performed, with two clusters ($k = 2$). Each iteration consists of two steps: in the first step, each datapoint is assigned to the closest mean. In the second step, each mean is updated according to the datapoints that are assigned to it. At each step, the average distance from each datapoint to its assigned mean is computed and plotted in the graph on the right-hand side. After five iterations, no change happens in the assignments: the algorithm has converged (although it would need an extra, sixth iteration, to know that; this is omitted from the animation to keep it compact.)

maximum-likelihood parameters is intractable, we can still devise algorithms that may not be guaranteed to find the global optimum in the parameter space, but will in general find a good local optimum.

To illustrate this, we first consider the k-means algorithm. This clustering technique (also called "vector quantisation" in some fields,) can be seen as a special case of the Gaussian mixture model where not all parameters are optimised. Let us first investigate the properties of k-means; afterwards, we will take a closer look at the relationship between k-means and GMM below, and will then generalise the algorithm to unconstrained GMMs and to other graphical models.

The algorithm for k-means is as follows. Given the number of clusters, k:

1. Initialise the means of the clusters at random
2. Assign each datapoint to the closest mean
3. Update the mean of the cluster by computing the mean of its assigned datapoints
4. Repeat from step 2 until the update results in no change.

The effect of this algorithm is illustrated on the "old faithful" dataset with two clusters, in Figure 7.3. The data has first been normalised to have zero mean and unit variance along each dimension: this is not necessary, but ensures that right angles are preserved in the graphic. If I hadn't normalised the data, the discriminant would be correct but look incorrect with respect to the plotted means.

7.2.1 Notes on k-means

Some things to node about the algorithm are:

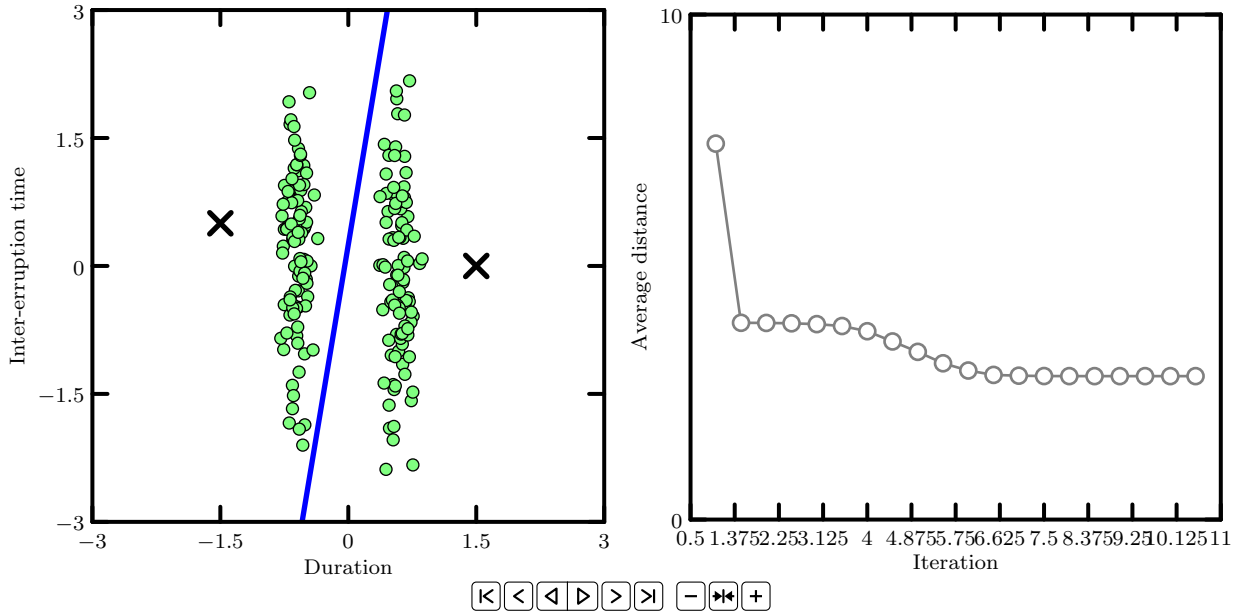


Figure 7.4: Illustration of how k-means fails on elongated clusters, due to the use of Euclidean distance. The solution we converge to is the global optimum, but notice how the algorithm can converge to other, locally optimal solutions depending on the initial parameter values. For example, if we initialise the means to be the means of the elongated clusters, the algorithm does not find a better (higher-likelihood) solution.

Distance measure Kmeans uses the Euclidean distance. This limits the applicability of the algorithm to some types of data, and suffers from the same problem as the kNN classifier described earlier. If clusters are elongated more than they are distant from each other, the resulting clustering will not separate the original clusters. This is illustrated in Figure 7.4. Moreover, the use of Euclidean distance additionally makes kmeans sensitive to outliers. Other distance metrics can be used, of course, but the choice of distance metric is very data-dependant, and we would therefore want to learn such things automatically.

Hard assignment Each datapoint is assigned to one and only one cluster. Datapoints that are distant from the cluster mean carry the same weight as datapoints which are close to the mean, and affect the updates as much. Especially when the data distribution is complex, one would prefer to have soft assignments, where each datapoint contributes to the update of a cluster's parameters according to the probability that it does indeed belong to that cluster. The EM algorithm for GMM, described next, does just that.

7.3 Gaussian Mixture Model

The GMM is a linear combination of Gaussian distributions. Such a combination can serve two purposes:

1. To describe complex probability densities
2. For clustering

In the first case, which Gaussian a datapoint is "assigned" to has no further meaning.[3] The combination of Gaussians can approximate any density arbitrarily closely, given sufficient Gaussians, and it is the final distribution that is important. This is illustrated in Figure 7.5. In the second case, the purpose is to

[3]As we will see, in reality each datapoint is assigned to all clusters, each with a certain probability. There are no real assignments.

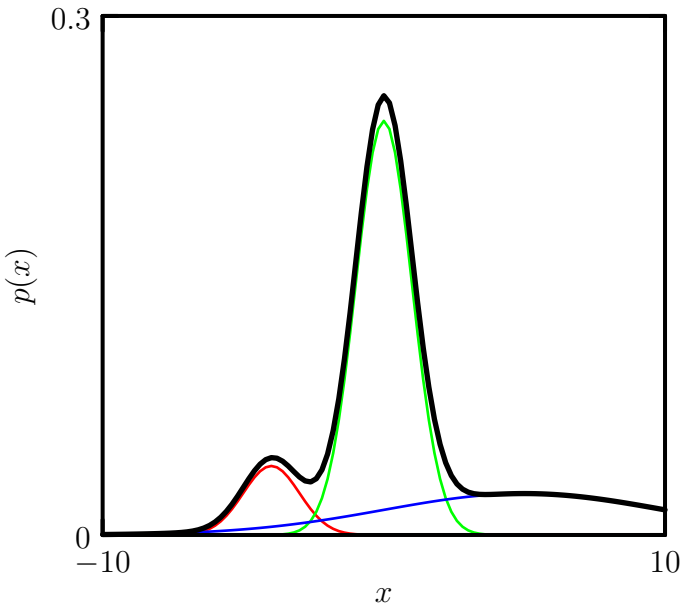


Figure 7.5: Representing a complex distribution (black line) with a mixture of three Gaussians (red, green and blue lines)

find underlying structure in the data, and which cluster a datapoint "belongs to" can be important. The underlying model is the same, however, so we shall not dwell further over these trivia.

The graphical model corresponding to the GMM is depicted in Figure 7.6. It is equivalent to the model of Figure 7.2, except that the assignment of a datapoint to a Gaussian is not observed. Mathematically, the *complete likelihood* function (that is, the probability of the data and the labels given the parameters as opposed to the incomplete likelihood, which is the probability of the data — only – given the parameters) is identical to the supervised case. We repeat it here, but will take the opportunity to consider the case of K Gaussians, for generality. To do this, we represent the assignment of datapoint $\mathbf{x}_i$ by vector $\mathbf{z}_i$, which contains all zeroes, except $\mathbf{z}_i(j) = 1$ for the Gaussian $\mathcal{C}_j$ it is assigned to. (We keep using the notation $\mathcal{C}_j$ to indicate the jth Gaussian, to avoid introducing new, superfluous symbols. But remember that, as discussed before, the Gaussians may not represent different classes at all.) We can then write that

$$p(\mathbf{x}_i, \mathbf{z}_i) = \begin{cases} p(\mathcal{C}_1)\, p(\mathbf{x}_i|\mathcal{C}_1) & \text{if } \mathbf{z}_i(j) = 1 \\ \quad \vdots & \\ p(\mathcal{C}_K)\, p(\mathbf{x}_i|\mathcal{C}_K) & \text{if } \mathbf{z}_i(K) = 1 \end{cases} \tag{7.18}$$

where we parametrise the different probabilities as

$$p(\mathcal{C}_1) = \rho_1 \qquad \dots \qquad p(\mathcal{C}_K) = \rho_K \tag{7.19}$$

$$p(\mathbf{x}_i|\mathcal{C}_1) = \mathcal{N}(\mathbf{x}_1; \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1) \qquad \dots \qquad p(\mathbf{x}_i|\mathcal{C}_K) = \mathcal{N}(\mathbf{x}_1; \boldsymbol{\mu}_K, \boldsymbol{\Sigma}_K) \tag{7.20}$$

with the additional constraint that the prior probabilities must sum up to one: $\sum_{k=1}^{K} \rho_k = 1$. Based on this, we can write down the complete likelihood:

$$p(\mathbf{x}_i, \mathbf{z}_i) = \prod_{k=1}^{K} \left(\rho_k\, \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right)^{z_i(k)} \tag{7.21}$$

Since we don't observe $\mathbf{z}_i$, we are really interested in $p(\mathbf{x}_i)$, which we can get by marginalisation: $p(\mathbf{x}_i) =$

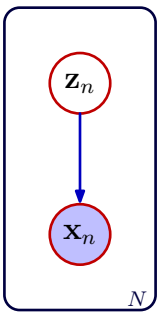


Figure 7.6: Graphical model for the GMM

$\sum_{\mathbf{z}_i} p(\mathbf{x}_i, \mathbf{z}_i)$. And since $\mathbf{z}_i$ is a vector of length K with only one non-zero element, we get:

$$p(\mathbf{x}_i) = \sum_{k=1}^{K} \rho_k \, \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{7.22}$$

This is the quantity that is plotted with a black line in Figure 7.5, for the possible values of a single, one-dimensional datapoint x. For the complete dataset we get, therefore,

$$p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \prod_{i=1}^{N} \sum_{k=1}^{K} \rho_k \, \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \ . \tag{7.23}$$

This is the likelihood[4], and this is the quantity that we want to optimise.

Optimising the likelihood is not trivial, because the product of sums means that when we take the first derivative with respect to one parameter, this derivative will be a non-linear function of all the other parameters, so that no closed-form solution can, in general, be found.

7.4 Gradient Descent

Let us first take the first derivative of the log-likelihood with respect to ρ_k. The log-likelihood is given by

$$\ln p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \sum_{i=1}^{N} \ln \sum_{k=1}^{K} \rho_k \, \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \ . \tag{7.24}$$

Using the chain rule of derivatives, we get:

$$\frac{\partial}{\partial \rho_k} \ln p(\mathbf{x}_1, \ldots, \mathbf{x}_N) = \sum_{i=1}^{N} \frac{1}{\sum_{k=1}^{K} \rho_k \, \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)} \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{7.25}$$

Notice how this derivative depends on $\rho_1, \ldots, \rho_K, \boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K, \boldsymbol{\Sigma}_1, \ldots, \boldsymbol{\Sigma}_K$. I chose to use ρ_k because that parameter is outside of the normal distribution, and therefore the simplest to optimise for, but the same holds when we try finding the derivative with respect to any other parameter. And hence, setting the derivative equal to zero and solving for the desired parameters does not yield a closed-form solution.

We can, however, compute the derivative at any point in the parameter space: for given values of the parameters, we can compute what the gradient is of the log-likelihood. We can use this to optimise the log-likelihood iteratively, by computing and following the gradient at each step. This is a valid method, which is indeed applied in practice. However, care must be taken when performing gradient descent:

1. As always in gradient descent, the step size can be hard to choose, and convergence can be slow

[4]The distinction between complete and incomplete likelihood is not always made explicit, but the term "likelihood" should always denote the probability of the observations given the parameters.

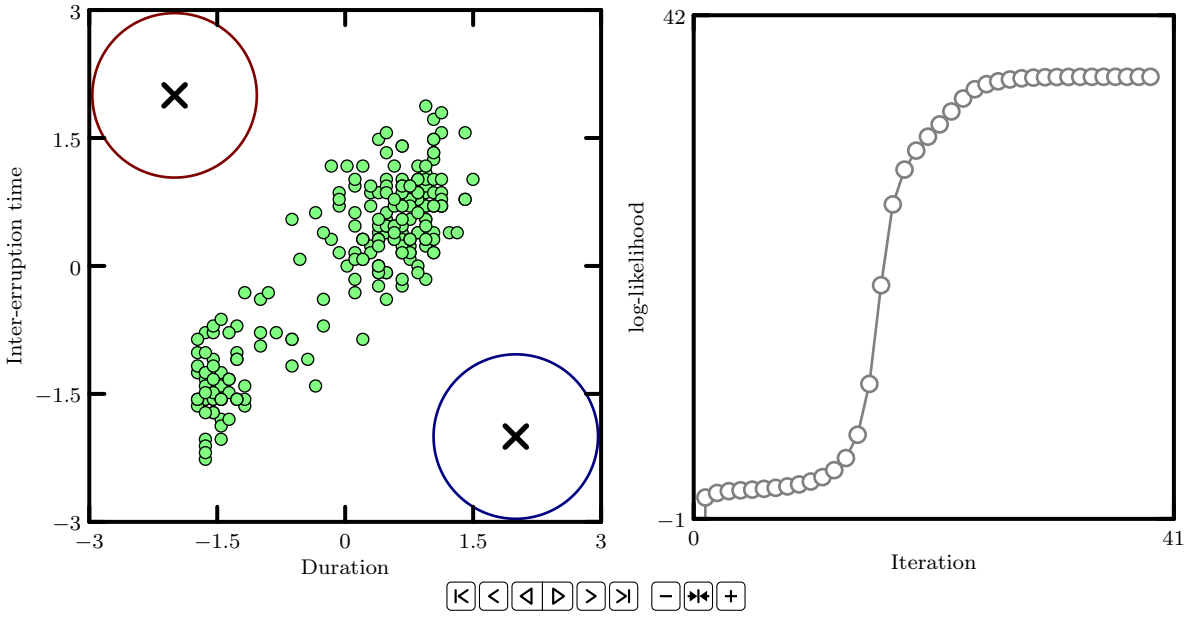


Figure 7.7: Expectation maximisation for the Gaussian mixture model

2. Constraints are not enforced. Simply doing gradient descent using the gradient of the log-likelihood will not result in a valid optimum, because no constraint is enforced on $\rho_1, \ldots, \rho_K$: these are probabilities, and must therefore be non-negative and sum up to one. In order to enforce this, one can parametrise the priors with the softmax function, as $\rho_k = \frac{\exp \rho_k}{\sum_j \exp \rho_j}$, and use the chain rule to compute the derivative with respect to ρ_k instead.

7.5 Expectation-Maximisation

The intuition behind the EM algorithm is as follows. We know the form of the complete log-likelihood and we know how to optimise it, but we don't have all the required observations, so we can't optimise that directly. We also know the form of the (incomplete) log-likelihood, and we do have all the observations for that one, but optimising it is hard. The idea is then that we could compute what the *expectation* of the complete log-likelihood (as this is our best estimate of what the real complete log-likelihood would be), and optimise that instead. The expectation of the complete log-likelihood has the same form as the complete log-likelihood itself, so optimising it should be easy. Of course, when we mention "expectation", some probability distribution is implied, and the purpose of the whole thing here is to *find* the parametrisation of that distribution, so we have a bit of a chicken-and-egg problem. The key idea of the EM algorithm is to start with some joint distribution over the variables (which may be quite unlike the real distribution), use this to compute the expectation of the joint log-likelihood, find the parameters that optimise this expectation and iteratively use the new parameters to recompute the expectation. This iterative procedure is shown in Figure 7.7 for a GMM with two Gaussian distributions: on the right-hand side, the data and the Gaussian distributions are plotted; on the left-hand side, the (incomplete) log-likelihood is plotted for each iteration.

Below, we shall first develop the EM optimisation procedure for the GMM, after which we will explore in some more detail why the algorithm works and why it is guaranteed to converge to some (local) optimum of the log-likelihood function. We are optimising a function $\mathcal{Q}$ with two different sets of parameters: the parameters from the previous iteration $\boldsymbol{\theta}^{\text{old}}$, which are known, and the parameters of the distribution $\boldsymbol{\theta}$, that

we want to find in this iteration:

$$\mathcal{Q}(\boldsymbol{\theta}^{\text{old}}, \boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\theta}^{\text{old}}}[\ln p(\mathbf{x}_1, \ldots, \mathbf{x}_N, \mathbf{z}_1, \ldots, \mathbf{z}_N)] \tag{7.26}$$

$$= \sum_{\mathbf{z}_1, \ldots, \mathbf{z}_N} p(\mathbf{z}_1, \ldots, \mathbf{z}_N | \mathbf{x}_1, \ldots, \mathbf{x}_N, \boldsymbol{\theta}^{\text{old}}) \sum_{i=1}^{N} \ln p(\mathbf{x}_i, \mathbf{z}_i | \boldsymbol{\theta}) \tag{7.27}$$

where the two sums can be swapped because the datapoints are independent:

$$= \sum_{i=1}^{N} \sum_{\mathbf{z}_i} p(\mathbf{z}_i | \mathbf{x}_i, \boldsymbol{\theta}^{\text{old}}) \ln p(\mathbf{x}_i, \mathbf{z}_i | \boldsymbol{\theta}) \tag{7.28}$$

Here, we have that $p(\mathbf{z}_i | \mathbf{x}_i, \boldsymbol{\theta}^{\text{old}})$ can be written as

$$p(\mathbf{z}_i | \mathbf{x}_i, \boldsymbol{\theta}^{\text{old}}) = \sum_{k=1}^{K} z_i(k) p(\mathcal{C}_k | \mathbf{x}_i, \boldsymbol{\theta}^{\text{old}}) \ , \tag{7.29}$$

the probability of the assignment vector $\mathbf{z}_i$ is the probability that the datapoint belongs to the Gaussian k, for the nonzero dimension $\mathbf{z}_i(k)$.[5] The posterior probability $p(\mathcal{C}_k | \mathbf{x}_i, \boldsymbol{\theta}^{\text{old}})$ is sometimes called the "responsibility" of Gaussian k for the datapoint, and is then denoted as $\gamma_k(\mathbf{x}_i)$. It is computed using Bayes' rule.

$$\gamma_k(\mathbf{x}_i) = \frac{\rho_k \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_j \rho_j \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j)} \tag{7.30}$$

The other quantity in Eq. (7.28) is $\ln p(\mathbf{x}_i, \mathbf{z}_i | \boldsymbol{\theta})$, the logarithm of the complete log-likelihood given in Eq. (7.21):

$$\ln p(\mathbf{x}_i, \mathbf{z}_i) = \sum_{k=1}^{K} z_i(k) \left[\ln \rho_k + \ln \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\right] \tag{7.31}$$

Filling these quantities into Eq. (7.28) and remembering that all $\mathbf{z}_i(k) = 0$ except for one which is one, we can combine the sums over k and obtain:

$$\mathcal{Q}(\boldsymbol{\theta}^{\text{old}}, \boldsymbol{\theta}) = \sum_{i=1}^{N} \sum_{k=1}^{K} \gamma_k(\mathbf{x}_i) \big[\ln \rho_k + \ln \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)\big] \tag{7.32}$$

7.5.1 Maximisation

Now we're getting there. All we need to do now, is compute the values of the parameters $\boldsymbol{\theta} = \{\rho, \boldsymbol{\mu}, \boldsymbol{\Sigma}\}$ that maximise $\mathcal{Q}(\boldsymbol{\theta}^{\text{old}}, \boldsymbol{\theta})$. We take the derivative and set it equal to zero, and obtain:

$$\frac{\partial \mathcal{Q}(\boldsymbol{\theta}^{\text{old}}, \boldsymbol{\theta})}{\partial \boldsymbol{\mu}_k} = 0 \qquad \Longrightarrow \qquad \sum_{i=1}^{N} \gamma_k(\mathbf{x}_i) \boldsymbol{\Sigma}_k^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_k) = 0 \qquad \Longrightarrow \tag{7.33}$$

$$\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i) \mathbf{x}_i = \sum_{i=1}^{N} \gamma_k(\mathbf{x}_i) \boldsymbol{\mu}_k \qquad \Longrightarrow \qquad \boldsymbol{\mu}_k = \frac{\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i) \mathbf{x}_i}{\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i)} \tag{7.34}$$

[5] We used a sum rather than a product, because we will not want to take the logarithm of this quantity, later on. The effect is the same, though: only one of the terms matters.

and similarly for the covariance,

$$\frac{\partial \mathcal{Q}(\boldsymbol{\theta}^{\text{old}}, \boldsymbol{\theta})}{\partial \boldsymbol{\Sigma}_k} = 0 \qquad \Longrightarrow \tag{7.35}$$

$$\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i) \Big[\boldsymbol{\Sigma}_k^{-1} - \boldsymbol{\Sigma}_k^{-1} (\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^\top \boldsymbol{\Sigma}_k^{-1} \Big] = 0 \qquad \Longrightarrow \tag{7.36}$$

$$\boldsymbol{\Sigma}_k \sum_{i=1}^{N} \gamma_k(\mathbf{x}_i) = \sum_{i=1}^{N} \gamma_k(\mathbf{x}_i) \Big[(\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^\top \Big] \qquad \Longrightarrow \tag{7.37}$$

$$\boldsymbol{\Sigma}_k = \frac{\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i) \Big[(\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^\top \Big]}{\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i)} \qquad \Longrightarrow \tag{7.38}$$

$$\boldsymbol{\Sigma}_k = \frac{\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i) \Big[\mathbf{x}_i \mathbf{x}_i^\top - 2\mathbf{x}_i \boldsymbol{\mu}_k^\top + \boldsymbol{\mu}_k \boldsymbol{\mu}_k^\top \Big]}{\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i)} \qquad \Longrightarrow \tag{7.39}$$

$$\boldsymbol{\Sigma}_k = \frac{\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i) \mathbf{x}_i \mathbf{x}_i^\top}{\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i)} - 2\boldsymbol{\mu}_k \frac{\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i) \mathbf{x}_i^\top}{\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i)} + \boldsymbol{\mu}_k \boldsymbol{\mu}_k^\top \qquad \Longrightarrow \tag{7.40}$$

$$\boldsymbol{\Sigma}_k = \frac{\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i) \mathbf{x}_i \mathbf{x}_i^\top}{\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i)} - \boldsymbol{\mu}_k \boldsymbol{\mu}_k^\top \tag{7.41}$$

Finally, to find the optimal priors ρ_k, we need to enforce the rules of probability: the priors must sum up to one. As always, we do that with a Lagrange multiplier (Appendix A). The Lagrangian is then given by

$$\sum_{i=1}^{N} \sum_{k=1}^{K} \gamma_k(\mathbf{x}_i) \big[\ln \rho_k + \ln \mathcal{N}(\mathbf{x}_i; \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \big] + \lambda (\sum_k \rho_k - 1) \tag{7.42}$$

Taking the derivatives results in the following K equations:

$$\begin{cases} \sum_{i=1}^{N} \gamma_k(\mathbf{x}_i) \Big/ \rho_k + \lambda = 0 \quad \forall k \in [1 \dots K] & \text{(A)} \\ \sum_{k=1}^{K} \rho_k = 1 & \text{(B)} \end{cases} \tag{7.43}$$

From (A), we get

$$\lambda \rho_k = -\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i) \tag{7.44}$$

and by summing both sides over all K Gaussians, we get

$$\lambda \underbrace{\sum_{k=1}^{K} \rho_k}_{1} = -\sum_{i=1}^{N} \underbrace{\sum_{k=1}^{K} \gamma_k(\mathbf{x}_i)}_{1} \tag{7.45}$$

$$\lambda = -N \tag{7.46}$$

$$\tag{7.47}$$

And so we get:

$$\rho_k = \frac{\sum_{i=1}^{N} \gamma_k(\mathbf{x}_i)}{N} \tag{7.48}$$

7.6 Why EM works

In the above, we have given the intuitive meaning of the EM algorithm, namely that the closest thing we can find to the incomplete likelihood is the expectation of the complete likelihood under the posterior distribution

of the latent variables. The idea was then that, since that's the best we could do, we might as well do it. In this section, we will see that this is indeed a valid thing to do, and that iteratively optimising the expectation of the complete likelihood does indeed maximise the (incomplete) likelihood.

To prove that, we will have to do some magical shuffling around with equations. It may not be clear where we're going in the beginning, but do read on and try to follow the steps: I hope that it will make sense in the end.

The idea is the following: from the product rule of probabilities, we have that:

$$p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) = p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})\, p(\mathbf{X}|\boldsymbol{\theta}) \quad \text{and, therefore:} \tag{7.49}$$

$$p(\mathbf{X}|\boldsymbol{\theta}) = \frac{p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})}{p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})} \tag{7.50}$$

If we introduce a random distribution over the latent variables $\mathbf{Z}$, we have that:

1. The distribution $q(\mathbf{Z})$ is normalised, so that

$$\sum_{\mathbf{Z}} q(\mathbf{Z}) = 1 \tag{7.51}$$

2. we can do the following:

$$p(\mathbf{X}|\boldsymbol{\theta}) = \frac{p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})}{q(\mathbf{Z})} \frac{q(\mathbf{Z})}{p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})} \tag{7.52}$$

and, therefore:

$$\ln p(\mathbf{X}|\boldsymbol{\theta}) = \ln \frac{p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})}{q(\mathbf{Z})} - \ln \frac{p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})}{q(\mathbf{Z})} \tag{7.53}$$

Leaving this aside for a moment, let's have a look at $\ln p(\mathbf{X}|\boldsymbol{\theta})$. Since $\ln p(\mathbf{X}|\boldsymbol{\theta})$ does not depend on $\mathbf{Z}$ and $q(\mathbf{Z})$ is normalised, we have:

$$\ln p(\mathbf{X}|\boldsymbol{\theta}) = \ln p(\mathbf{X}|\boldsymbol{\theta}) \sum_{\mathbf{Z}} q(\mathbf{Z}) \tag{7.54}$$

$$= \sum_{\mathbf{Z}} q(\mathbf{Z}) \ln p(\mathbf{X}|\boldsymbol{\theta}) \tag{7.55}$$

Filling in Eq. (7.53), this gives us:

$$\ln p(\mathbf{X}|\boldsymbol{\theta}) = \sum_{\mathbf{Z}} q(\mathbf{Z}) \ln \frac{p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})}{q(\mathbf{Z})} \underbrace{- \sum_{\mathbf{Z}} q(\mathbf{Z}) \ln \frac{p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})}{q(\mathbf{Z})}}_{\mathrm{KL}(p \,||\, q)} \tag{7.56}$$

And now things get interesting, because we recognise the formula for the Kullback-Leibler divergence in the second term of the right-hand part of Eq. (7.56). We have seen in subsection 5.8.1, relying on Jensen's inequality (see Appendix D), that the KL-divergence between any two distributions is always non-negative. As a consequence,

$$\ln p(\mathbf{X}|\boldsymbol{\theta}) \geqslant \underbrace{\sum_{\mathbf{Z}} q(\mathbf{Z}) \ln \frac{p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})}{q(\mathbf{Z})}}_{\mathcal{L}(q, \boldsymbol{\theta})} \tag{7.57}$$

In other words, $\mathcal{L}(w, \boldsymbol{\theta})$ is a lower bound for $p(\mathbf{X}|\boldsymbol{\theta})$, the quantity that we really want to optimise. In the EM algorithm, it is this lower bound that we optimise iteratively.

The EM algorithm alternates two steps:

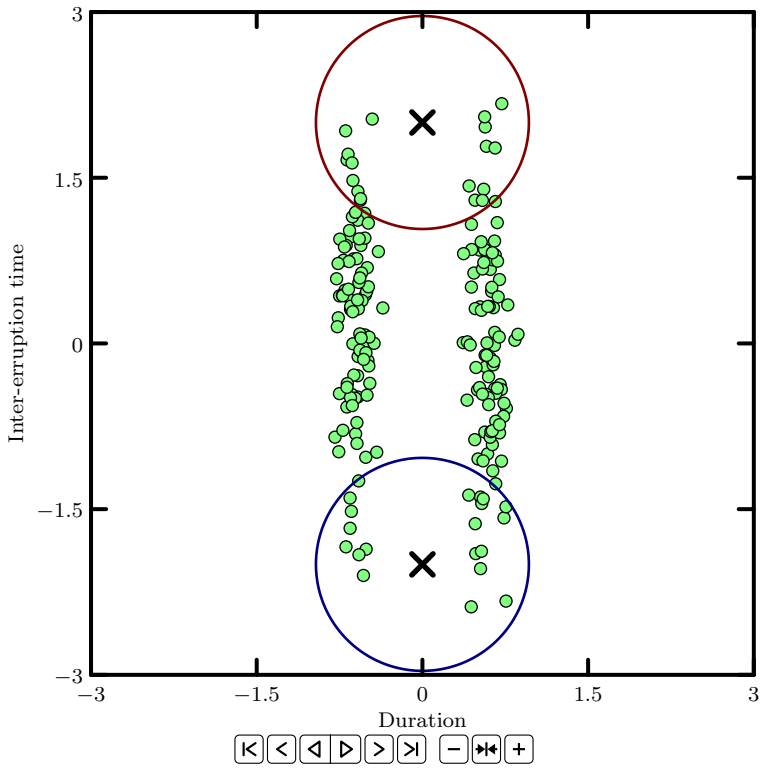


Figure 7.8: Expectation maximisation for the Gaussian mixture model on the same elongated clusters as Figure 7.4. Notice how despite the bad initialisation, the algorithm still converges to a solution where both clusters are correctly modelled.

E-step During the E-step, we set $q(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}})$. By doing this, we make $\text{KL}(p \,||\, q)$ equal to zero, so that $\mathcal{L}(q, \boldsymbol{\theta}) = \ln p(\mathbf{X}|\boldsymbol{\theta})$: the lower bound is then tight.

M-step During the M-step, we maximise $\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$ with respect to $\boldsymbol{\theta}$. Since we have set $q(\mathbf{Z}) = p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta})$ during the E-step, the lower bound has become:

$$\mathcal{L}(q, \boldsymbol{\theta}) = \sum_{\mathbf{Z}} q(\mathbf{Z}) \ln \frac{p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta})}{q(\mathbf{Z})} \tag{7.58}$$

$$= \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}}) \ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) - \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}}) \ln p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}}) \tag{7.59}$$

$$= \sum_{\mathbf{Z}} p(\mathbf{Z}|\mathbf{X}, \boldsymbol{\theta}^{\text{old}}) \ln p(\mathbf{X}, \mathbf{Z}|\boldsymbol{\theta}) - const \tag{7.60}$$

In other words: maximising the expectation of the complete log-likelihood ($\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{\text{old}})$) is equivalent to maximising the lower bound on the incomplete log-likelihood.

Moreover, since the KL-divergence $\text{KL}(p||q)$ is only zero when p equals q and q does not change in the M-step but p does (unless it is already at a maximum), the incomplete log-likelihood is guaranteed to increase *more* than the lower bound did. This makes the EM algorithm a particularly effective optimisation procedure.

7.7 k-means versus GMM

The k-means algorithm is a special case of the EM algorithm for GMM, where we keep the covariance fixed, for all clusters, as $\mathbf{\Sigma} = \epsilon \mathbf{I}$ in the limit of $\epsilon \to 0$. The assignment step of k-means is the E-step of EM where, due to the zero covariances, the responsibilities are zero or one, and the update step is the maximisation step, where we maximise the expectation of the complete log-likelihood with respect to the means.

Both algorithms have some complementary strengths and weaknesses.

Singularities The greatest weakness of the GMM, is that the problem is inherently ill-posed. Because the parameters of the Gaussian distributions are unconstrained, a Gaussian modelling a single point will have a covariance equal to zero, and contribute a density of $+\infty$ at that point. Since the other densities are necessarily non-negative, the maximum incomplete likelihood will also be $+\infty$. But this does not model the real distribution of the data well at all; it is an example of the extreme overfitting that maximum-likelihood methods are prone to.

Because it constrains the covariances to be zero for *all* Gaussians, k-means does not suffer this weakness

Unbalanced sizes We have seen in Figure 7.4 that the k-means algorithm could not handle clusters well if they were very elongated, because it cannot take the shape of the cluster into account. The GMM performs much better in such cases, as shown in Figure 7.8. In effect, because we take the covariance matrix into account, we obtain a distance measure (called the Mahalanobis distance) where not all dimensions are weighted equally. This distance measure is automatically adapted to the dataset when we adjust the covariance matrix.

Speed of convergence Because k-means updates fewer parameters, its search space is much more limited than the general algorithm for GMM. It will therefore typically require fewer iterations to converge to an optimum, and will have fewer local optima. The individual iterations are also less complex and therefore slightly faster.

Identifiability An issue that arises in both k-means and GMM is that there are multiple ($K!$ to be precise) equivalent solutions for a given dataset: the identity of the clusters does not affect the likelihood and all of those solutions are equivalent. For density estimation this does not matter, but when trying to interpret the results or compare different solutions, this becomes an issue.

Chapter 8

Neural Networks

In chapter 4, we have seen how perceptrons, a crude mathematical model of how biological neurons work, could learn parameters to perform linear classification using a surprisingly simple learning rule. The problem with perceptrons was that the were limited to linear classification: if we try to combine perceptrons so that one perceptron would take the output of the other as input, then such a construct could compute more complex, non–linear functions but the perceptron learning algorithm could not be applied and, due to the discontinuous activation function, the derivative cannot in general be used to find the model parameters.

In that same chapter, we have also seen that it is possible to transform a problem by representing it in a different vector space, using fixed basis functions, in such a way that a problem that is hard in the original space becomes easy in the new space. The problem, then, was that although such transformations can be found, it was not clear how the precise parametrisation of these functions could be found.

Multi–layer perceptrons (MLP) are inspired from this, and provide a way to combine multiple layers of perceptrons (albeit with a different activation function), so that one layer in effect provides a set of basis functions to transform the data, while the next layer performs the actual regression or classification task in this new space.

8.1 Feed–forward Neural Networks

Each node j in a Multi–layer perceptron computes an output z_j which is function of its inputs of the form:

$$z_j = f(a_j) \ , \tag{8.1}$$

where $f(\cdot)$ is a differentiable, non-linear activation function[1] and the activation a_j is weighted sum of the inputs:

$$a_j = \sum_{i=1}^{M_j} w_{ji} z_{n_i^j} + w_{j0} \tag{8.2}$$

Here, we sum over the M_j nodes that are connected to provide input to node j, w_{ji} is the weight associated with input i of node j and w_{j0} is a fixed bias to node j, while n_i^j is the index of the node connected to the ith input of node j. As usual, we can use a vector inner product to represent this sum, and we can extend the vector of inputs with a dimension that is fixed to one to incorporate the bias in the vector sum. Using $\mathbf{w}_j$ to denote the vector of weights $(w_{j0}, w_{j1}, \ldots, w_{jM})^\top$ and $\mathbf{z}_j$ to denote the vector $(1, z_{jn_1^j}, \ldots, z_{jn_M^j})^\top$, this becomes

$$a_j = \mathbf{w}_j^\top \mathbf{z}_j \tag{8.3}$$

The input nodes are not computational nodes, and the "output" z_i of these nodes is simply the input value x_i.

[1] Output nodes are not restricted to non-linear activation functions, but the other nodes are

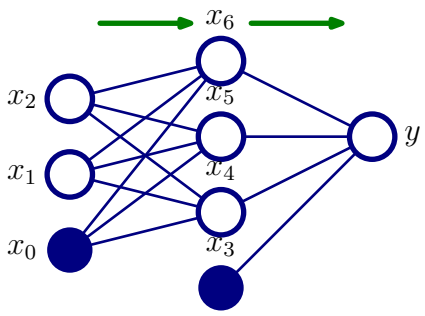


Figure 8.1: Example of a feed–forward neural network with two input nodes, one hidden layer with three nodes and a single output node. The shaded nodes are included to indicate the extra weight used for the bias; their value is fixed to one.

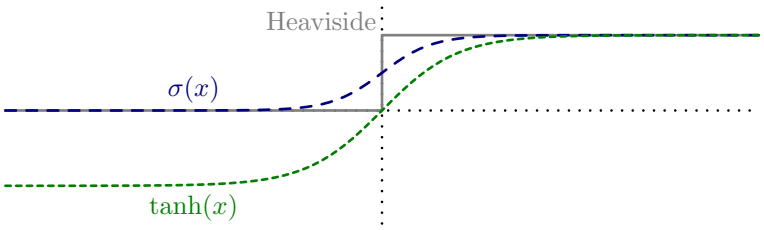


Figure 8.2: Different activation functions: the Heaviside step function (black, plain) used in perceptrons, and the logistic sigmoidal (dashed blue) and hyperbolic tangent (green, dashed-dotted) used in neural networks

Such multi–layer perceptrons can be visualised graphically, and an example of this is depicted in Figure 8.1. In this case, we have a 2-layer network[2]: a layer of three hidden nodes and a layer of one output node. The two input variables are also depicted as nodes in the graph, but they don't perform any computations, and the shaded nodes are not computational nodes. Their value is fixed to one, so that the corresponding weight for the nodes they are connected to is a fixed bias.

These networks are constrained, in that the connections are "one-way": nodes cannot have input values which depend on their own output. As we will see in a moment, this constraint is required to make training possible. Such networks are, therefore, called feed-forward neural networks. Notice that in a multi-layer structure, not all nodes from one layer need be connected to all layers of the next, nor is there a constraint that the inputs of a node must come from the previous layer: connections can skip layers.

The outputs of the network are then computed by propagating the input values forward through the network, and the outputs of the current layer are used as inputs to the future nodes, until the output nodes are reached and the output values are computed. The activation functions of the nodes in the hidden layer must be non-linear functions in order for the final function to be non-linear (a linear function of a linear function is itself a linear function, and is therefore not very interesting). For the output nodes, however, this constraint does not apply and, which is more, the use of a "squashing function" in the output nodes reduces the range in which the output values can lie. Depending on the desired output, it is therefore often useful to use a linear activation function for the output nodes.

8.2 Training

The major change that made MLP possible is the use of a differentiable activation function. Where the perceptron used a discontinuous step function, MLPs use a continuous function such as the logistic sigmoidal function $\sigma(x)$ or the hyperbolic tangent. These functions are a typical choice because of the ease with which their derivatives can be computed, and are compared to the Heaviside step function in Figure 8.2. They are

[2]The way layers are counted in the literature is a bit vague. By convention, in these lecture notes, we consider the number of layers of computing nodes

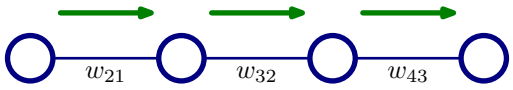


Figure 8.3: A first, very simple neural network

defined as:

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{8.4}$$

It is a good exercise to verify that the derivatives of these functions are given by

$$\frac{\mathrm{d}\,\sigma(x)}{\mathrm{d}x} = \sigma(x)(1-\sigma(x)) \qquad \frac{\mathrm{d}\tanh(x)}{\mathrm{d}x} = 1 - \tanh^2(x) \tag{8.5}$$

which are both particularly cheap to compute, since the value of the function itself is typically already computed during the evaluation of the network function.

During training, we want to find the values for the weights w_{ji}, for all nodes, which makes the outputs optimal in some sense. Since the output of the network is a non-trivial function of the inputs and of the parameters, this is not possible in closed form. However, thanks to the structure of the network and since the activation functions are differentiable, we can compute the gradient of the output with respect to the parameters. We can therefore train the network by gradient descent. The question is then how we can effectively compute this gradient for any given neural network. We address this question in the next section.

8.3 Backpropagation

The structure of an MLP is constrained in two important ways:

1. Each node computes a weighted sum of its inputs.
2. The output of a node does not affect its inputs (there are no loops).

These constraints are important, because they provide us with a functional structure in which it is easy to compute the derivative of the output (or some function of the output) with respect to any of the weights in the network. The algorithm to compute these derivatives is called backpropagation (because it is based on propagating information back from the outputs towards the inputs). In the following, we gradually build up more complex networks to explain how this works.

8.3.1 Introduction by example

First, let us consider a simple network of one input, one hidden layer and one output. Such a network is depicted in Figure 8.3. This network computes the following function if the input x:

$$f(x) = h_3\left(w_{43}\, h_2\left(w_{32}\, h_1(w_{21}\, x)\right)\right) \tag{8.6}$$

We have training inputs $x_1 \ldots x_N$ and matching training labels $t_1 \ldots t_N$. Training the network consists of adjusting the weights such that the output is as close as possible to the labels for the given training data. We do this by minimising the sum-squared error function:

$$E(x_1 \ldots x_N) = \frac{1}{2}\sum_{i=1}^{N}(f(x_i) - t_i)^2 \tag{8.7}$$

$$= \frac{1}{2}\sum_{i=1}^{N}\left(h_3\left(w_{43}\, h_2\left(w_{32}\, h_1(w_{21}\, x)\right)\right) - t_i\right)^2 \tag{8.8}$$

This function can be minimised by gradient descent, if we can compute the gradient. In this case, we have three parameters, w_{21}, w_{32} and w_{43}. The derivatives are given by:

$$\begin{aligned}
\frac{\partial}{\partial w_{43}} E(x_1 \dots x_N) &= \frac{\partial}{\partial w_{43}} \frac{1}{2} \sum_{n=1}^{N} (h_3(w_{43}\, h_2(w_{32}\, h_1(w_{21}\, x_i))) - t_i)^2 \\
&= \sum_{n=1}^{N} \underbrace{(h_3(w_{43}\, h_2(w_{32}\, h_1(w_{21}\, x_i))) - t_i)}_{z_3^{(n)} - t_i} \underbrace{h_3'(w_{43}\, h_2(w_{32}\, h_1(w_{21}\, x_i)))}_{h_3'(a_3^{(n)})} \underbrace{h_2(w_{32}\, h_1(w_{21}\, x_i))}_{z_2^{(n)}} \\
&= \sum_{n=1}^{N} \delta_3^{(n)}\, z_2^{(n)} \qquad (8.9) \\
\frac{\partial}{\partial w_{32}} E(x_1 \dots x_N) &= \frac{\partial}{\partial w_{32}} \frac{1}{2} \sum_{n=1}^{N} (h_3(w_{43}\, h_2(w_{32}\, h_1(w_{21}\, x_i))) - t_i)^2 \\
&= \sum_{n=1}^{N} \underbrace{(h_3(w_{43}\, h_2(w_{32}\, h_1(w_{21}\, x_i))) - t_i)}_{z_3 - t_i} \underbrace{h_3'(w_{43}\, h_2(w_{32}\, h_1(w_{21}\, x_i)))}_{h_3'(a_3)} \\
&\qquad w_{43}\, h_2'(w_{32}\, h_1(w_{21}\, x_i)) \underbrace{h_1(w_{21}\, x_i)}_{z_1} \\
&= \sum_{n=1}^{N} \delta_3\, w_{43}\, h_2'(w_{32}\, h_1(w_{21}\, x_i))\, z_1 \\
&= \sum_{n=1}^{N} \delta_2 z_1 \qquad (8.10) \\
\frac{\partial}{\partial w_{21}} E(x_1 \dots x_N) &= \sum_{n=1}^{N} (h_3(w_{43}\, h_2(w_{32}\, h_1(w_{21}\, x_i))) - t_i)\, h_3'(w_{43}\, h_2(w_{32}\, h_1(w_{21}\, x_i))) \\
&\qquad w_{43}\, h_2'(w_{32}\, h_1(w_{21}\, x_i))\, w_{32}\, h_1'(w_{21}\, x_i) \underbrace{x_i}_{z_0} \\
&= \sum_{n=1}^{N} \delta_2\, w_{32}\, h_1'(w_{21}\, x_i)\, z_0 \\
&= \sum_{n=1}^{N} \delta_1 z_0 \qquad (8.11)
\end{aligned}$$

We obtained this result using the chain rule of derivatives, and we can readily distinguish that the application of the chain rule results in some terms being identical for all derivatives. In fact, we can define the δ-terms recursively:

$$\begin{aligned}
\delta_3 &= (z_3 - t_i)\, h_3'(a_3) \\
\delta_2 &= \delta_3\, w_{43}\, h_2'(a_2) \\
\delta_1 &= \delta_2\, w_{32}\, h_1'(a_1) \qquad (8.12)
\end{aligned}$$

These terms are called the "errors", which we "back-propagate" through the network. The derivative of the output error with respect to the weights is obtained by multiplying the back-propagated error with the corresponding, forward-propagated, network response. Now the interesting thing is that if we make the network more complex by adding more nodes in any of the layers, the complexity of the backpropagation algorithm does not increase: all the nodes compute a weighted sum so that, since the derivative of a sum is the sum of the derivatives, the terms that do not depend on the weight at hand drop away. To illustrate this, consider the network depicted in Figure 8.4. In this case, the error we want to minimise is the sum of

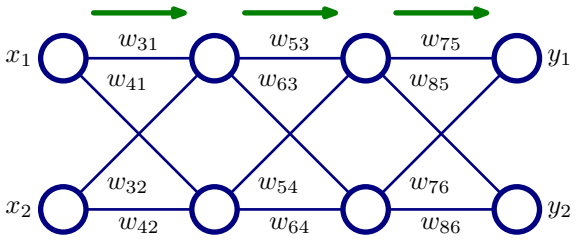


Figure 8.4: A second, slightly more complex neural network. There are two output nodes, with activation functions h_7 and h_8, and the hidden layers contain two nodes each.

the errors on the two outputs, so that we want to compute the derivative, with respect to every weight, of

$$E = \frac{1}{2}\sum_{n=1}^{N}(z_4^{(n)} - t_1^{(n)})^2 + (z_8^{(n)} - t_2^{(n)})^2 \tag{8.13}$$

where the superscript (n) indicates the value that corresponds to datapoint n.

During forward-propagation, we get

$$\begin{aligned}
a_3 &= w_{31}x_1 + w_{32}x_2 & z_3 &= h_3(a_3)\\
a_4 &= w_{41}x_1 + w_{42}x_2 & z_4 &= h_4(a_4)\\
a_5 &= w_{53}z_3 + w_{54}z_4 & z_5 &= h_5(a_5)\\
a_6 &= w_{63}z_3 + w_{64}z_4 & z_6 &= h_6(a_6)\\
a_7 &= w_{75}z_5 + w_{76}z_6 & y_1 &= z_7 = h_7(a_7)\\
a_8 &= w_{85}z_5 + w_{86}z_6 & y_2 &= z_8 = h_8(a_8)
\end{aligned} \tag{8.14}$$

Using these, we can compute the derivatives of the error with respect to each of the weights as:

$$\begin{aligned}
\frac{\partial E^{(n)}}{\partial w_{75}} &= (y_1^{(n)} - t_1^{(n)})\, h_7'(a_7)\, z_5 = \delta_7\, z_5\\
\frac{\partial E^{(n)}}{\partial w_{85}} &= (y_2^{(n)} - t_2^{(n)}) h_8'(a_8) z_5 = \delta_8 z_5\\
\frac{\partial E^{(n)}}{\partial w_{54}} &= (z_8^{(n)} - t_2^{(n)}) h_8'(a_8) w_{85} h_5'(a_5) z_4 + (z_7^{(n)} - t^{(n)}) h_7'(a_7) w_{75} h_5'(a_5) z_4\\
&= (\delta_8\, w_{85} h_5'(a_5) + \delta_7\, w_{75} h_5'(a_5)) z_4\\
\frac{\partial E^{(n)}}{\partial w_{31}} &= (z_7^{(n)} - t_1^{(n)})\, h_7'(a_7)\, w_{75}\, h_5'(a_5)\, w_{53}\, h_3'(a_3)\, z_1\\
&\quad + (z_7^{(n)} - t_1^{(n)})\, h_7'(a_7)\, w_{76}\, h_6'(a_6)\, w_{63}\, h_3'(a_3)\, z_1\\
&\quad + (z_8^{(n)} - t_2^{(n)})\, h_8'(a_8)\, w_{85}\, h_5'(a_5)\, w_{53}\, h_3'(a_3)\, z_1\\
&\quad + (z_8^{(n)} - t_2^{(n)})\, h_8'(a_8)\, w_{86}\, h_6'(a_6)\, w_{63}\, h_3'(a_3)\, z_1\\
&= \delta_3\, z_1
\end{aligned} \tag{8.15}$$

Notice how the derivative does not become unmanageably complex. Thus is because the derivative of a term that is constant with respect to the relevant weight is zero, and because the derivative of a sum equals the sum of the derivatives. This leads us to define the backpropagation recursively. We introduce the so-called "errors", δ_i, as

$$\delta_i^{(n)} = h_i'(a_i)\sum_j w_{ji}\,\delta_j \tag{8.16}$$

Using these, the derivative of the error with respect to a weight w_{ij} can be computed as $\delta_i \, z_j$. The recursive definition of the errors allows us to compute them in a single pass through the nodes, starting at the outputs and progressing backwards towards the inputs, hence the name of "Error backpropagation" or simply "backpropagation".

Chapter 9

Gaussian Processes

Where the Dirichlet process used a Dirichlet distribution to specify a discrete probability distribution over continuous functions, the Gaussian process (GP) provides a continuous distribution over continuous functions. The concept is slightly arcane, but is very elegant and worth understanding. The "bible" of Gaussian processes is Rasmussen and William's Rasmussen and Williams [2006], which provides a brilliant overview of the history, derivations, and practical considerations of Gaussian processes. Gaussian processes have a long history, and have been well-known in, *e.g.*, meteorology and geostatistics (where GP prediction is known as "kriging") for 40 years. A good overview of the use of Gaussian processes for spatial data can be found in Cressie [1993].

Gaussian Processes have recently been used successfully to extract human pose Ek et al. [2007], a problem that is known to be strongly multimodal and hence hard to model Ek et al. [2008], and to create complex models of human motion Wang et al. [2008]. The flexibility of GPs comes from the wide range of kernel functions that can be used, which makes it possible to model very complex signals, with multiple levels of superposition. As early as 1969, GPs were used to model the firing patterns of human muscle Clamann [1969].

The idea of Gaussian processes is to consider a function $f(\mathbf{x})$ as an infinite vector of function values $\mathbf{f}$, one for every possible value of x. In this representation, we can define a distribution over the function as a distribution over this vector, rather than as a distribution over the parameters of some parametrisation of the function. For the purpose of this explanation, we will consider one-dimensional functions, but the concept easily extends to multiple dimensions. It turns out that, if we assume that this distribution is Gaussian, it is actually possible to manipulate such an object in practice, and the resulting distribution is closely related to a distribution over parametric functions.

The Gaussian process defines the conditional joint distribution of all elements in our infinite-dimensional vector $\mathbf{f}$ as being Gaussian, with some mean and covariance.[1] The question, then, is of course how we could represent such a distribution. The answer relies on two key observations:

1. We only ever need to evaluate a function for a finite number of function values. Figure 9.1 illustrates this idea by showing a single sample from some Gaussian process. In this case, a 49-dimensional sample of $f(\mathbf{x})$ is drawn from the process, for a given set of 49 one-dimensional inputs.

2. One of the properties of the multivariate Gaussian distribution is that the marginal distribution of a subset of its dimensions is again a Gaussian distribution, with mean and covariance the relevant elements of the original means and covariances. Therefore, the distribution returned by the Gaussian process over the points where the function is evaluated is the same whether we take all other points into account or not, as long as we know how our finite set of points covary.

We cannot define a mean vector and covariance matrix over all the points in our function vector, as an unfortunate side effect of its infinite size, but we can provide a functional description of its mean vector and covariance matrix conditionally on the input values $\mathbf{x}$ at hand. These functions lie at the heart of the Gaussian process, and define its properties. The capacity to ignore the points where the function is not

[1] To be more formally exact, it is defines the distribution over any finite subset of the variables in that vector to be Gaussian.

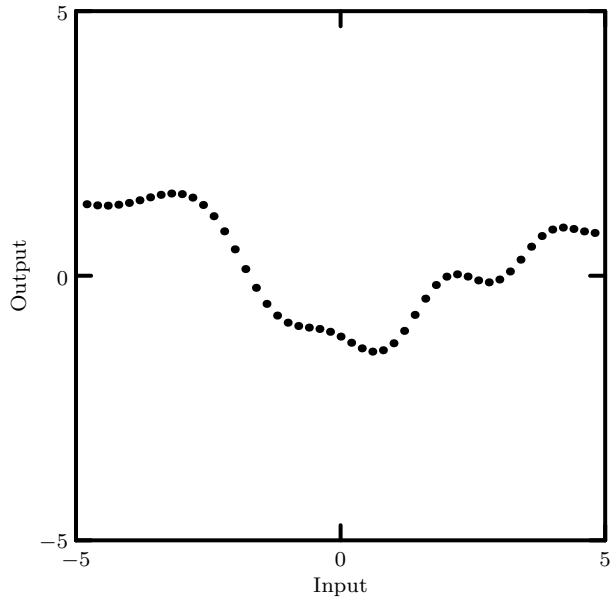


Figure 9.1: Illustration of a single sample drawn from a Gaussian process

evaluated is what makes Gaussian Processes tractable and, indeed, very efficient for small datasets. The combination of computational tractability and a formally consistent Bayesian treatment of the model makes Gaussian processes very appealing.

9.0.1 Gaussian process as Bayesian Linear Regression

Consider standard linear regression with Gaussian noise:

$$y = f(\mathbf{x}) + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, \sigma^2) \tag{9.1}$$

$$= \varphi(\mathbf{x})^\top \mathbf{w} + \epsilon \tag{9.2}$$

The target output y is a noisy version of a linear function of the parameters $\mathbf{w}$ and the feature representation $\varphi(\mathbf{x})$ of the input datapoint $\mathbf{x}$. We assume that the output noise is Gaussian, with zero mean and σ^2 variance.

We are given a set of training datapoints $\mathbf{X}$, which we represent in a design matrix, $\mathbf{\Phi}$, and a set of targets $\mathbf{y}$. From this, we want to learn to predict the output $f(\mathbf{x}_*)$ for a given input $\mathbf{x}_*$. We specify the design matrix as

$$\mathbf{\Phi} \triangleq \begin{bmatrix} \varphi(\mathbf{x}_1)^\top \\ \vdots \\ \varphi(\mathbf{x}_N)^\top \end{bmatrix} \tag{9.3}$$

so that each row contains the features of one datapoint. In the Bayesian framework, we need to introduce a prior distribution over the parameters $\mathbf{w}$, and choose to use a zero-mean Gaussian prior, with covariance matrix $\mathbf{\Sigma}$: $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$. Since both the likelihood $p(\mathbf{y}|\mathbf{\Phi}, \mathbf{w})$ and the prior $p(\mathbf{w})$ are Gaussian, the posterior $p(\mathbf{w}|\mathbf{y}, \mathbf{\Phi})$ is also Gaussian. Moreover, for a given input value $\mathbf{x}_*$, the predictive distribution over the function values $\mathbf{f}(\mathbf{x}_*)$ is given by

$$p(f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{\Phi}, \mathbf{y}) = \int p(f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{w})\, p(\mathbf{w}|\mathbf{\Phi}, \mathbf{y})\, \mathrm{d}\mathbf{w} \tag{9.4}$$

This predictive distribution over function values is again Gaussian, and is given by:

$$p(f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{\Phi}, \mathbf{y}) = \mathcal{N}(\frac{1}{\sigma^2}\varphi(\mathbf{x}_*)^\top \mathbf{A}^{-1}\mathbf{\Phi}\mathbf{y}, \varphi(\mathbf{x}_*)^\top \mathbf{A}^{-1}\varphi(\mathbf{x}_*)) \tag{9.5}$$

where $\mathbf{A} = \frac{1}{\sigma^2}\mathbf{\Phi}^\top\mathbf{\Phi} + \mathbf{\Sigma}^{-1}$, and can be rewritten as

$$p(f(\mathbf{x}_*)|\mathbf{x}_*, \mathbf{\Phi}, \mathbf{y}) = \mathcal{N}(\varphi_*^\top\mathbf{\Sigma}\varphi\,(\mathbf{\Phi}\mathbf{\Sigma}\mathbf{\Phi}^\top + \sigma^2 I)^{-1}\mathbf{y}, \\ \varphi_*^\top\mathbf{\Sigma}\varphi_* - \varphi_*^\top\mathbf{\Sigma}\mathbf{\Phi}^\top(\mathbf{\Phi}\mathbf{\Sigma}\mathbf{\Phi}^\top + \sigma^2 I)^{-1}\mathbf{\Phi}\mathbf{\Sigma}\varphi_*) \quad (9.6)$$

where we used φ_* as shorthand for $\varphi(\mathbf{x}_*)$. This last form is advantageous when the number of features is larger than the number of datapoints and, since $\mathbf{\Sigma}$ is positive-definite, we can rewrite the multiplications of the form $\varphi(\mathbf{x})^\top\mathbf{\Sigma}\varphi(\mathbf{x}')$ as $\boldsymbol{\psi}(\mathbf{x}) \cdot \boldsymbol{\psi}(\mathbf{x}')$ for some vector of feature functions $\boldsymbol{\psi}(\mathbf{x})$. Notice that $\varphi(\mathbf{x})$ only occurs in multiplications of that form in Eq. (9.6), so that we can use the *kernel trick* and fully specify our predictive distribution with a kernel function $k(\mathbf{x}, \mathbf{x}') = \boldsymbol{\psi}(\mathbf{x}) \cdot \boldsymbol{\psi}(\mathbf{x}')$. For every set of feature functions, we can compute the corresponding kernel function. Moreover, for every kernel function there exists a (possibly infinite) expansion in feature functions. This infinite expansion is not a problem in a Bayesian framework, because the (implicit) integration over the parameter values prevent the model from overfitting on the training data.

The Gaussian process is fully specified by its mean function, $\mu(\mathbf{x})$, and its covariance or kernel function $k(\mathbf{x}, \mathbf{x}')$. The functions $f(\mathbf{x})$ are distributed as:

$$f(\mathbf{x}) \sim \mathcal{GP}(\mu(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \quad (9.7)$$

where, by definition, $\mu(\mathbf{x}) \triangleq \mathbb{E}[f(x)]$ and $k(\mathbf{x}, \mathbf{x}') \triangleq \mathbb{E}[(f(\mathbf{x}) - \mu(\mathbf{x}))(f(\mathbf{x}') - \mu(\mathbf{x}'))]$. In practice, the mean function is often taken to be zero, for simplicity as much as for symmetry in the function space. This is not a restriction of the Gaussian process itself, however, and sometimes a non-zero mean function is indeed specified. Notice that a zero mean does not make the meak of a particular function equal to zero; rather, for every point in the input space the expectation over the value of all functions at that point, is zero. If we are given a set of inputs $\mathbf{X}$, a set of targets $\mathbf{y}$ and a set of test datapoints $\mathbf{X}_*$, we can directly specify the covariance matrix of our joint distribution over the function values for the training datapoints and the test datapoints as

$$\begin{bmatrix}\mathbf{y} \\ \mathbf{f}_*\end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix}\mathbf{K}(\mathbf{X}, \mathbf{X}) + \sigma^2 I & \mathbf{K}(\mathbf{X}, \mathbf{X}_*) \\ \mathbf{K}(\mathbf{X}_*, \mathbf{X}) & \mathbf{K}(\mathbf{X}_*, \mathbf{X}_*)\end{bmatrix}\right) \quad (9.8)$$

where each element (i, j) of the matrix $\mathbf{K}(\mathbf{X}, \mathbf{X}') \triangleq k(\mathbf{x}_i, \mathbf{x}'_j)$ for $\mathbf{x}_i$ being datapoint i in the set $\mathbf{X}$. Using the standard result for the Gaussian conditional distribution, we can compute

$$p(\mathbf{f}_*|\mathbf{X}, \mathbf{y}, \mathbf{X}_*) = \mathcal{N}(\boldsymbol{\mu}, \mathbf{\Sigma}), \text{ where} \quad (9.9)$$

$$\boldsymbol{\mu} = \mathbf{K}(\mathbf{X}_*, \mathbf{X})(\mathbf{K}(\mathbf{X}, \mathbf{X}) + \sigma^2 I)^{-1}\mathbf{y} \text{ and} \quad (9.10)$$

$$\mathbf{\Sigma} = \mathbf{K}(\mathbf{X}_*, \mathbf{X}_*) - \mathbf{K}(\mathbf{X}_*, \mathbf{X})(\mathbf{K}(\mathbf{X}, \mathbf{X}) + \sigma^2 I)^{-1}\mathbf{K}(\mathbf{X}, \mathbf{X}_*) \quad (9.11)$$

Notice how the mean of the predictive distribution is non-zero, and depends on the training data and targets. The covariance consists of a term representing the prior covariance of the test datapoints, which is diminished by a term that depends on the training and test datapoints, and on the noise of the targets; it does not depend on the value of the training targets. Also notice how the matrix inversion, which dominates the computational complexity of Eq. (9.10) and Eq. (9.11), does not depend on the test data: it needs only be computed once. Training a Gaussian process, therefore, consists of: 1) selecting the correct properties of the function of interest (by selecting the right kernel function,) and 2) computing the said matrix inverse.

9.0.2 Kernel functions

The kernel function fully specifies the (zero-mean) Gaussian process prior. It captures our prior beliefs about the type of function we are looking at; most importantly, we need to specify beforehand how "smooth" we believe the underlying function to be. Figure 9.2 illustrates how different kernel function result in different styles of functions: the plots on the left hand side show samples from the prior, while the corresponding plots on the right hand side show samples from a GP with the same covariance function, conditional on the three depicted training datapoints. The kernel functions depicted here are stationary, meaning that they

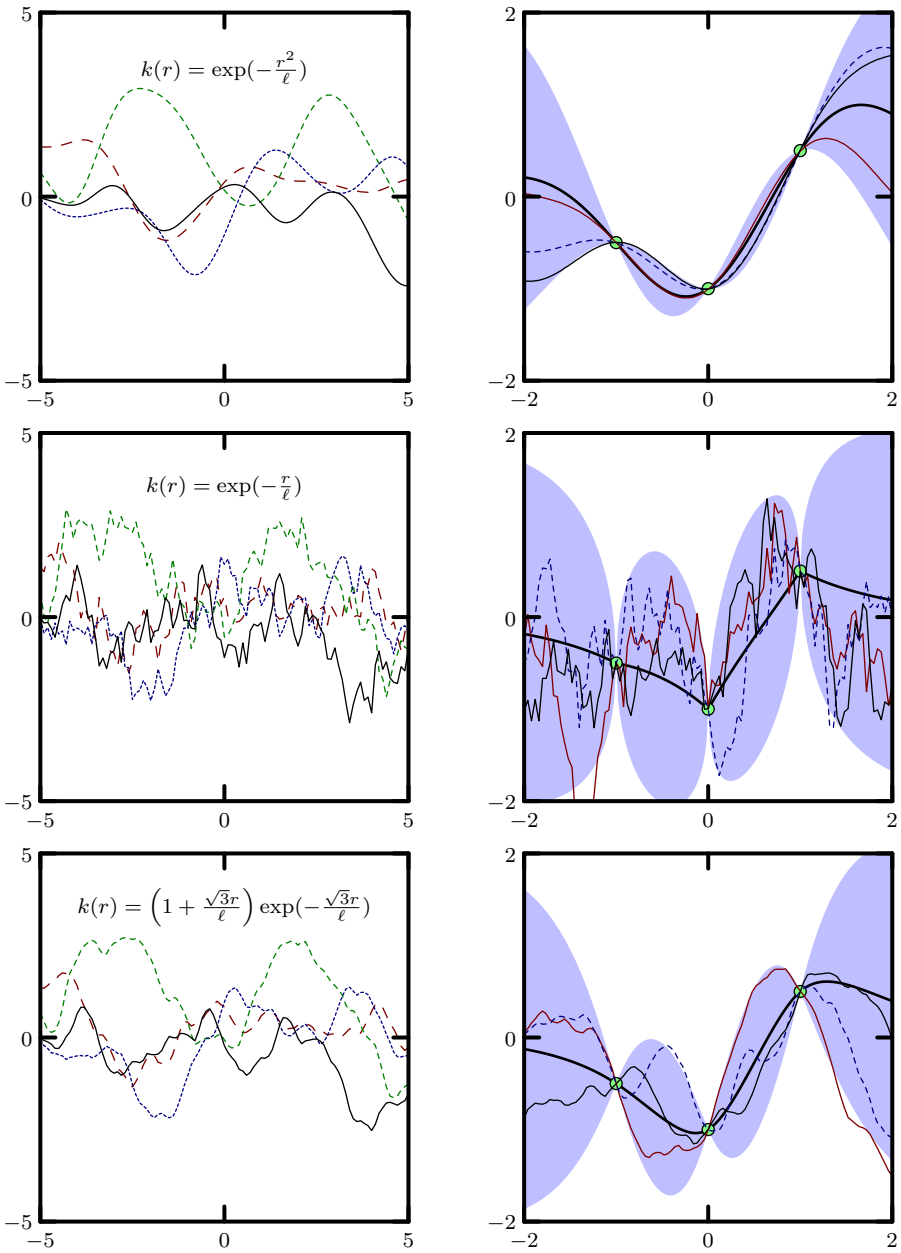


Figure 9.2: The kernel function specifies the properties of the modelled function. The plots on the left hand side show samples from the Gaussian process prior (before we observe any data). The corresponding plots on the right hand side show samples of the posterior distribution, given the training data (thin lines), as well as the mean function (thick black line) and two standard deviations around the mean (shaded area). In all plots the length scale ℓ was set to one, and $r = |\mathbf{x} - \mathbf{x}'|$

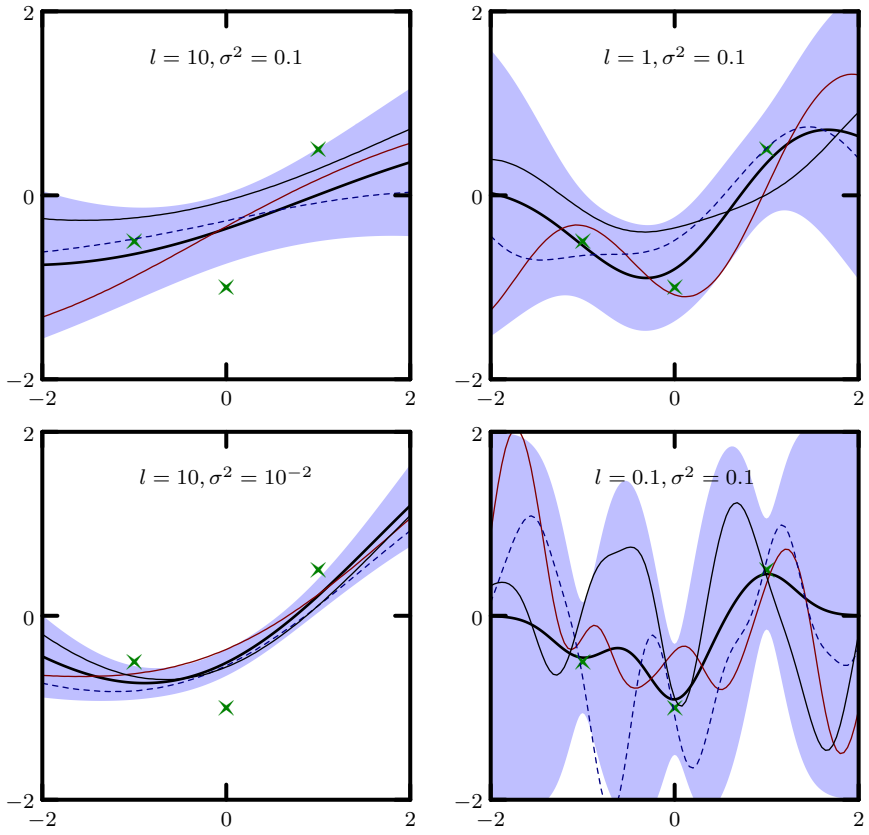


Figure 9.3: Illustration of Gaussian process regression, with one-dimensional input and one-dimensional output, and three training datapoints

depend solely on the difference between the two input vectors (and are therefore invariant to translations). Many other kernel functions are possible, including non-stationary ones: the only requirement of a function for it to be a valid kernel function is that it should be positive semi-definite. Informally, this means that it must be a function that leads to a positive semi-definite covariance matrix. The easiest way to derive new kernel functions is to modify known kernel functions using operations which are known to preserve the positive-semidefiniteness of the function Bishop [2007]

Kernel functions often have parameters. These parameters are not affected by the Gaussian process, and are part of its specification. One recurrent parameter is the length scale ℓ, which basically sets how the distance between datapoints affect how the corresponding function outputs covary. The length scale, therefore, affects how smoothly the function varies. Figure 9.3 shows samples from four different Gaussian processes, all with squared exponential kernel function and different length scales, illustrating how this parameter affects the distribution over the functions. A short length scale increases the variance in areas away from the training data, and consequently also increases the effect of the prior in those areas.

The parameters of the kernel function, also called hyper-parameters, are fixed for a particular GP but can, of course, themselves be learnt from data. In the Bayesian framework, the way to do this is to place a prior distribution over the hyperparameters, and to integrate out the hyperparameters. This integral typically cannot be done analytically, and approximations are then required. When the posterior distribution over the parameters is strongly peaked, one acceptable approximation to the posterior is the Dirac impulse: the integral then becomes the maximum-likelihood function. Since the posterior distribution over the hyperparameters is more often strongly peaked than the posterior over the parameters, the values of the hyperparameters are often found by maximising the marginal likelihood of the training data with respect to the hyperparameters. Such an optimisation is called type II Maximum Likelihood and, although this procedure re-introduces a risk of overfitting, this risk is far lower than with maximum-likelihood optimisation of

the parameters.

9.0.3 Classification

Gaussian processes lend themselves very naturally for regression, but can also be used very effectively for classification. Instead of having an unrestricted function output as in the case of regression, the output of a two-class classifier is constrained to lie in the range $[0 \dots 1]$, so that it can be interpreted as the probability of one of the classes given the inputs. This is typically done using a "squashing function", such as the logistic sigmoidal:

$$\sigma(a) = \frac{1}{1 + e^{-a}} \tag{9.12}$$

For multi-class problems, this can be extended very naturally to the soft-max function.

Training a GP for classification is more involved than for regression, however. Because the output is pushed through a non-linear function, the posterior distribution is not Gaussian anymore, and the posterior distribution cannot be derived in closed form anymore. Approximation schemes are therefore required, and both variational approximation schemes, as well as sampling methods, are routinely used for this.

Chapter 10

Dimensionality Reduction

10.1 Principal Component Analysis

Principal Component Analysis or PCA is a dimensionality reduction technique based on simple linear projection: a number of linearly independent vectors are chosen, and the data is projected on those. As we know, we can project a vector on another vector using the inner product of the two vectors (see chapter 2), and so the major problem to solve is to decide on how to choose the projection vectors.

When projecting a vector (the datapoint) on another vector (the projection direction), we obtain a scalar. If we project the datapoint on as many linearly independent vectors as the dimensionality of the original datapoint, we can use the obtained scalars and the chosen projection directions to reconstruct the datapoint exactly. If we keep fewer projection directions as we had dimensions originally, then the reconstruction is not exact. This is illustrated in Figure 10.1: in this case, we have two-dimensional data which we can reconstruct exactly using two linearly independent basis vectors, or with a reconstruction error by projecting on a single vector. Which leads us to PCA: different projection directions will result in different reconstruction errors, as illustrated in Figure 10.2. PCA chooses the projection vectors so as to minimise the sum-squared error between the original data and the reconstructed data or, equivalently, so as to maximise the variance of the projected data.

The question then becomes: how can we find the projection directions which maximise the variance of the projected data? It turns out that these directions can be found efficiently using eigenvector-eigenvalue decomposition of the covariance matrix of our original data. We can derive this as follows. The function that we want to maximise is the variance of the projected data, with respect to the projection direction $\mathbf{u}_1$, where $\mathbf{u}_1$ is a vector of unit length:

$$f(\mathbf{u}_1) = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{u}_1^\top \mathbf{x}_n - \boldsymbol{\mu})^2 \ , \tag{10.1}$$

where $\boldsymbol{\mu}$ is the mean of the projected data, which is given by

$$\boldsymbol{\mu} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{u}_1^\top \mathbf{x}_n \ . \tag{10.2}$$

We can bring the product out of the sum, so that the mean of the projected data is identical to the projection of the mean of the original data:

$$\boldsymbol{\mu} = \mathbf{u}_1^\top \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n = \mathbf{u}_1\bar{\mathbf{x}} \ , \tag{10.3}$$

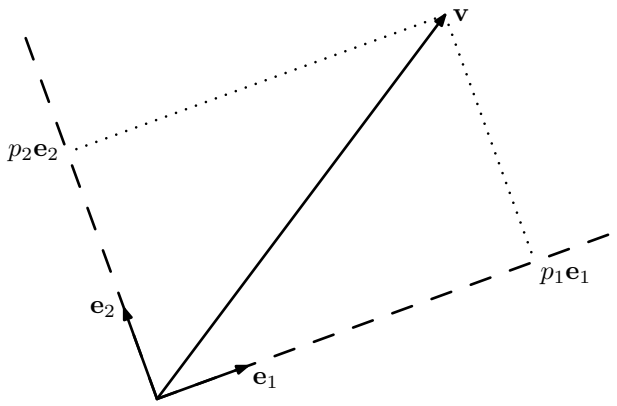


Figure 10.1: Illustration of projection in two dimensions. The vector $\mathbf{v}$ is projected on the two linearly independent unit vectors $\mathbf{e}_1$ and $\mathbf{e}_2$, resulting in $p_1 = \mathbf{v}^\top \mathbf{e}_1$ and $p_2 = \mathbf{v}^\top \mathbf{e}_2$. For each of these projections, the reconstruction is given by $p_i \mathbf{e}_i$. The original vector can be recovered by summing the reconstructions, $\mathbf{v} = p_1\mathbf{e}_1 + p_2\mathbf{e}_2$

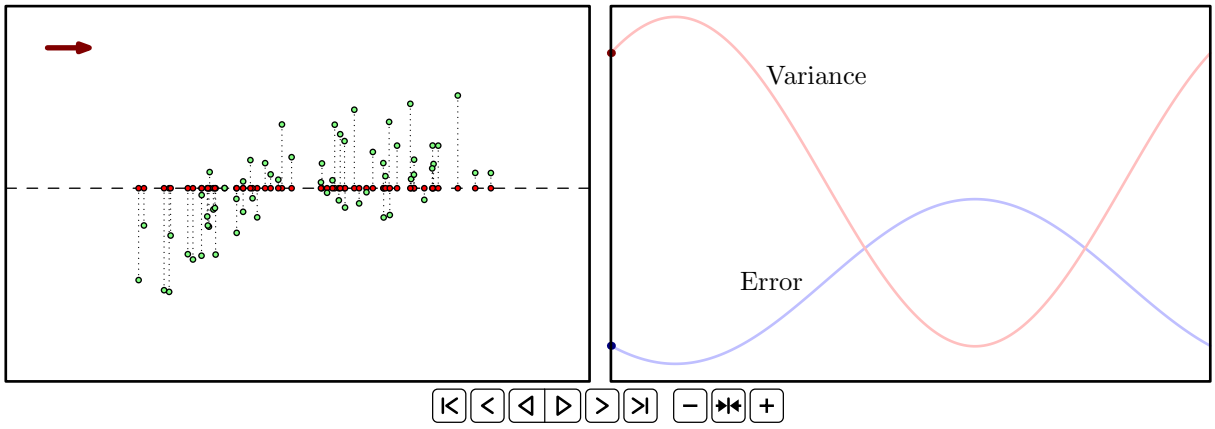


Figure 10.2: Illustration of the reconstruction error with linear projection. In the left graph, the data are plotted as green dots, the projection direction is indicated by the red arrow and the dashed line, while the reconstructed data are indicated by the bright red dots. The right graph plots the variance of the projected data (red) and the total sum-squared error of the reconstructed data (blue)

where $\bar{\mathbf{x}}$ is the sample mean of the data. Putting this together, we get:

$$f(\mathbf{u}_1) = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{u}_1^\top \mathbf{x}_n - \mathbf{u}_1\bar{\mathbf{x}})^2 \tag{10.4}$$

$$= \frac{1}{N}\sum_{n=1}^{N}(\mathbf{u}_1^\top (\mathbf{x}_n - \bar{\mathbf{x}}))^2 \tag{10.5}$$

By introducing a matrix $\mathbf{X}$, where each row n corresponds to the mean-subtracted datapoint $\mathbf{x}_n - \bar{\mathbf{x}}$, we obtain the vector of the projections of all datapoints as $\mathbf{X}\mathbf{u}_1$, and we can rewrite Eq. (10.5) as:

$$f(\mathbf{u}_1) = \frac{1}{N}\mathbf{u}_1^\top \mathbf{X}^\top \mathbf{X}\mathbf{u}_1 \tag{10.6}$$

$$= \mathbf{u}_1^\top \mathbf{S}\mathbf{u}_1 \tag{10.7}$$

where $\mathbf{S}$ is the data covariance matrix. Ok. So now we want to maximise this function, subject to the constraint that $\mathbf{u}_1$ is a unit-length vector. We encode this constraint in a Lagrangian function:

$$L(\mathbf{u}_1, \lambda) = \mathbf{u}_1^\top \mathbf{S}\mathbf{u}_1 - \lambda(\mathbf{u}_1^\top \mathbf{u}_1 - 1) \tag{10.8}$$

Taking the first derivative and setting it equal to zero results in:

$$\mathbf{S}\mathbf{u}_1 = \lambda\mathbf{u}_1 \tag{10.9}$$

which is, of course, the definition of an eigenvector-eigenvalue problem. The vectors which maximise our function $f(\mathbf{u}_1)$, $\mathbf{u}_1$ are eigenvectors of the data covariance matrix $\mathbf{S}$, with corresponding eigenvalue λ. Large eigenvalues correspond to large variance, so we select the eigenvectors with the largest eigenvalues.

10.1.1 PCA with very high-dimensional data

When the data is very high-dimensional, the covariance matrix $\mathbf{S}$ becomes impossible to work with. For example, consider working with images of 256×256 pixels: each image is then a vector with $65k$ elements, and the covariance matrix is a $65k \times 65k$ matrix. Such a matrix requires 48GB to simply store (using double precision floating point variables), which is not realistic on most current hardware. However, in practice we rarely have enormous amounts of high-dimensional data. For PCA of images of faces (so-called eigenfaces), for example, some 300 faces seem to span the complete space of human faces quite well. In such cases, we can do better than computing the complete covariance matrix. Indeed, we know that in a d-dimensional space, two datapoints will define exactly one single line that goes through both: this is also the line which, when used for projection, results in zero reconstruction error. Projecting these two datapoints on any vector which is orthogonal to this line will result in both points having the same projection, so that the variance of the projected data will be zero. If we have three datapoints, we span a plane and therefore two vectors, and so on: we always have, for N datapoints (of dimensionality d larger than N), at most $N-1$ vectors with non-zero variance of the projected data, $N-1$ eigenvectors with non-zero corresponding eigenvalue. As a consequence, we're allowed to do the following: instead of considering the eigenvalue decomposition of the covariance matrix

$$\frac{1}{N}\mathbf{X}^\top \mathbf{X}\mathbf{u}_1 = \lambda\mathbf{u}_1 \ , \tag{10.10}$$

we consider the eigenvalue decomposition of another, related, matrix

$$\frac{1}{N}\mathbf{X}\mathbf{X}^\top \mathbf{v}_1 = \nu\mathbf{v}_1 \ , \tag{10.11}$$

Notice that if this latter eigenvalue decomposition is easier than the previous one, as long as we have fewer datapoints than dimensions. In our previous example, the matrix $\mathbf{X}^\top\mathbf{X}$ would be a $65k \times 65k$ matrix, while

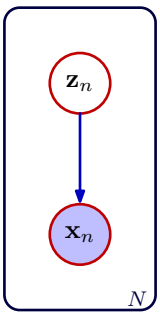


Figure 10.3: Graphical model of probabilistic PCA

$\mathbf{X}\mathbf{X}^\top$ would be a 300×300 matrix, which is vastly easier to work with. So what is the relationship between these two matrices? Well, if we take Eq. (10.11) and pre-multiply both sides by $\mathbf{X}^\top$, we get:

$$\frac{1}{N}\mathbf{X}^\top\mathbf{X}\mathbf{X}^\top\mathbf{v}_1 = \nu\mathbf{X}^\top\mathbf{v}_1 \ , \tag{10.12}$$

so that, by setting $\mathbf{u} = \mathbf{X}^\top\mathbf{v}_1$, we recover our original eigenvalue decomposition of the covariance matrix. In other words, if $\mathbf{v}_1$ is an eigenvector of $\mathbf{X}\mathbf{X}^\top$, then $\mathbf{X}^\top\mathbf{v}_1$ is an eigenvector of $\mathbf{X}^\top\mathbf{X}$. Doing this eigenvector decomposition and subsequent matrix multiplication is a lot cheaper than directly doing the eigenvector decomposition of covariance matrix.

Practical side note Notice that in the eigenvalues of the two matrices are not the same, and that if $\mathbf{v}_1$ is a unit-length vector, $\mathbf{X}^\top\mathbf{v}_!$ is (most likely) not. When using this technique, remember to normalise the computed eigenvectors: $\mathbf{u}_1 = \mathbf{u}/||\mathbf{u}||^2$.

10.2 Probabilistic PCA

Instead of considering PCA as a deterministic algorithm for finding projection directions which minimise the reconstruction error, we can instead consider it as a generative probabilistic algorithm. This means that we consider the low-dimensional projection of the data to be some hidden, continuous random variable $\mathbf{z}$ with some distribution, while the original data is an observed random variable $\mathbf{x}$ which depends on $\mathbf{z}$. PCA defined the low-dimensional representation to be the projection of the (mean-subtracted) high-dimensional data on a limited number of projection vectors: in other words, we could reconstruct the original mean subtracted data (approximately) by scaling the projection vectors by the resulting projections and summing these (as explained in Figure 10.1):

$$\mathbf{x} - \boldsymbol{\mu} = \mathbf{z}_1\mathbf{u}_1 + \cdots + \mathbf{z}_k\mathbf{u}_k \tag{10.13}$$

which we can write as a matrix product

$$\mathbf{x} = \mathbf{W}\mathbf{z} + \boldsymbol{\mu} \tag{10.14}$$

where we introduced the matrix $\mathbf{W}$ whose columns contain the projection vectors,

$$\mathbf{W} = \begin{bmatrix}\mathbf{u}_1 & \cdots & \mathbf{u}_k\end{bmatrix} \tag{10.15}$$

Now PCA was defined as minimising the sum-squared error between the original data and the reconstruction which, as we've seen before, is equivalent with maximising the probability of the data under the assumption that the noise has a Gaussian distribution with zero mean and spherical covariance:

$$p(\mathbf{x}|\mathbf{z}) = \mathcal{N}(\mathbf{x}; \mathbf{W}\mathbf{z} + \boldsymbol{\mu}, \sigma^2\mathbf{I}) \tag{10.16}$$

Now we still need to specify a prior distribution over the latent variables $\mathbf{z}$, to ensure that our model is not underspecified. If we did not do this, we could scale our projections $\mathbf{z}$ by any constant, and scale $\mathbf{W}$ by the inverse of that constant, to obtain the exact same reconstructions. We specify that $\mathbf{z}$ must be normally distributed, with zero mean and unit covariance:

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I}) \tag{10.17}$$

The corresponding graphical model is depicted in Figure 10.3. With this model, we can optimise the free parameters to maximise the probability of the data. The maximum-likelihood estimate of the mean, $\boldsymbol{\mu}$, is the sample mean of the data, while the projection matrix $\mathbf{W}$ and the data variance σ can be optimised by gradient descent. This has the downside of not finding the optimal value of the parameters in closed form (that is, it can take a few iterations and does not find the exact optimum) but means that no matrix inversion is required. For large, high-dimensional datasets, where standard PCA is not possible in practice, probabilistic PCA is a solution.

10.3 More to come...

There are more things I want to talk about here, including factor analysis, ICA and different forms of non-linear dimensionality reduction.

10.3.1 t-distributed Stochastic Neighbour Embedding

One technique that is very useful for visualising data in two or three dimensions is t-distributed stochastic neighbour embedding or t-SNE. This technique is similar to LLE, in that it attempts to preserve the neighbourhood relationships in the original high-dimensional space as best as possible in the low-dimensional space

Chapter 11

Sampling

When working with probabilistic models, there is one main issue that comes back over and over again: integration. Whether it be to compute the marginal probability of some variables, conditional probabilies or the expected value of some function, all of these require the computation of an integral. When variables are discrete, this integration becomes a sum over all possible values of the variables and in the case of continuous variables, the integrals can sometimes be computed in closed form. Most often, however, this will not be possible and tractable approximations are then required.

In chapter 6 subsection 6.3.2, we have seen algorithms that leverage the factorisation of the joint distribution of different variables to minimise the number of integrations required to compute a marginal probability. In this chapter, we see how we can approximate the integration process itself.

11.1 Numerical integration

The general form of the problem we want to solve is therefore of the form

$$\mathbb{E}[f(\mathbf{x})] = \int f(\mathbf{x})\, p(\mathbf{x}) \mathrm{d}\mathbf{x} \tag{11.1}$$

where for discrete variables the integral is replaced by a sum. The function $f(\mathbf{x})$ can be any function. For example, if the x-axis represents the size of the tomatoes you're growing on your balcony, the function may represent the price you can ask for the different tomato sizes should you choose to sell them. The expectation (under the probability distribution over tomato sizes) of this function would then be how much money you would expect to get from growing and selling tomatoes. If we are interested in computing the marginal distribution $p(\mathbf{z})$ from the joint distribution $p(\mathbf{x}, \mathbf{z})$, this expectation becomes

$$p(\mathbf{z}) = \int p(\mathbf{x}, \mathbf{z}) d\mathbf{x} = \int p(\mathbf{z}|\mathbf{x})\, p(\mathbf{x}) \mathrm{d}\mathbf{x} = \mathbb{E}[p(\mathbf{z}|\mathbf{x})] \tag{11.2}$$

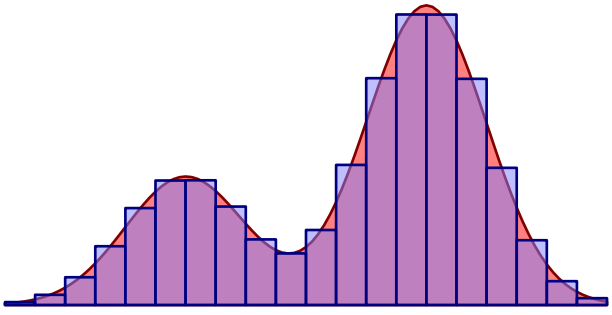


Figure 11.1: Illustration of numerical integration using the midpoint (rectangle) rule, where the area under the original function (red curve) is approximated by the sum of the areas covered by the rectangles.

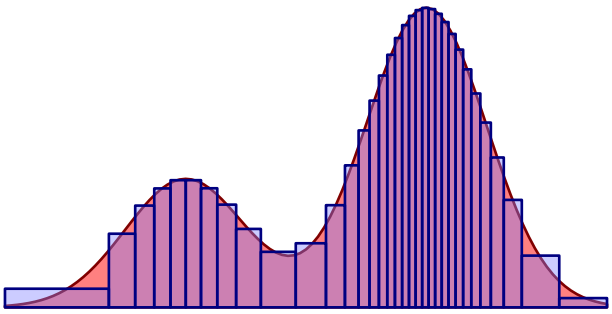


Figure 11.2: Illustration of how sampling approximates the area under the PDF. Notice that this is only an illustration, in that in this figure the boxes are located so as to have an area of exactly one Nth (samples, in contrast, are stochastic) and that the PDF is of infinite extent (the area of the leftmost and rightmost boxes corresponds to the area under the curve between infinity and the boundary of the next box).

where $f(\mathbf{x}) = p(\mathbf{z}|\mathbf{x})$ is a conditional PDF. We focus, therefore, on the general case of the computation of an expectation.

If our integral cannot be solved in closed form and is definite (*i.e.*, between known bounds), an obvious solution might be to perform numerical integration using interpolation, as illustrated in Figure 11.1. Yet this solution has many disadvantages: (1) it requires a definite integral, while the domain of most PDF is unbounded; (2) it does not scale to multiple dimensions (as the number of required hypercubes grows exponentially with the number of dimensions), and (3) the computation of the area of each rectangle requires the same amount of resources, while some rectangles contribute a lot to the end result and others have a negligible contribution.

This leads us to the idea of sampling. Imagine that we had a number N of values $\mathbf{x}_i$ in the domain of a function $g(\mathbf{x}) = f(\mathbf{x})\,p(\mathbf{x})$, such that the number of such values in the interval $\mathbf{x} + \mathrm{d}\mathbf{x}$ was proportional to $p(\mathbf{x})$. Then since the probability density $p(\mathbf{x})$ cannot be negative and is normalised, we could approximate the proportion of the area under $g(\mathbf{x})$ numerically as

$$\int g(\mathbf{x})\mathrm{d}\mathbf{x} \approx \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{x}_i) \tag{11.3}$$

Instead of computing the function "everywhere" (*i.e.*, at uniform intervals between the bounds of the integral), we can simply evaluate it at a finite number of locations (even if the integral is indefinite), to obtain an approximation to the desired integral. This works, because we know that the total area under the curve must add up to one, so that each of the N samples are associated with one Nth of the area. In effect, instead of splitting the domain of the PDF into segments of equal width[1] — with different corresponding areas under the curve — we instead split it up into segments of different width, but with the same area under the curve. This is illustrated in Figure 11.2.

11.1.1 An example

As an example illustrating what sampling is, why it's useful and how to use it, let us consider the following situation. Let us consider that the function we are computing the expectation of is the benefit, measured in marks on the exam, of your reading my lecture notes. This benefit depends on when you read the notes: if you start reading them from the start of the course, you get the time to process the material and benefit most. If you start later, you still benefit from them — for these notes truly are excellent — but you have less time to process the material and therefore benefit less. However, as the exam gets near, reading the notes yields renewed benefit, because although you haven't got time to process it all, at least it's all fresh in your memory and you can find the information you need fast (the exam is open-book).

So now, I'd like to know whether writing these notes is worth the effort. This depends on when you start reading them, so the answer to my question depends on the probability distribution, over time, of your

[1] Width in one dimension, area in two, etc.

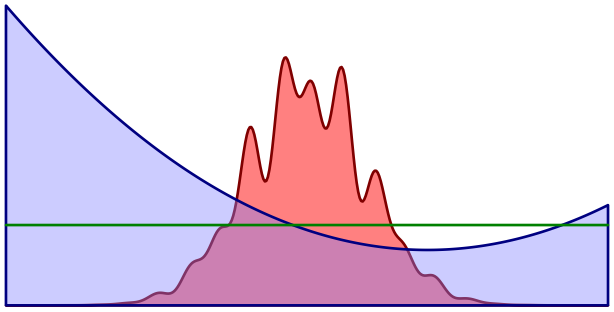


Figure 11.3: An artificial example to illustrate the use of sampling. In the bottom axis, we have time. The red function indicates the probability that you will start reading the lecture notes, at any time over the duration of the course. The blue function is the amount of benefit that you will get from reading the notes in function of time: if you read them at the start of the course, you benefit most. If you read them during the course you benefit less, but if you read them right before the exam, your benefit increases slightly. The question now is: as the author of these notes, how much can I expect you to benefit from them (and is it worth my time writing them)? The expectation is hard to compute analytically, due to the complex form of the PDF, but we can compute it by sampling. The result is indicated by the horizontal green line.

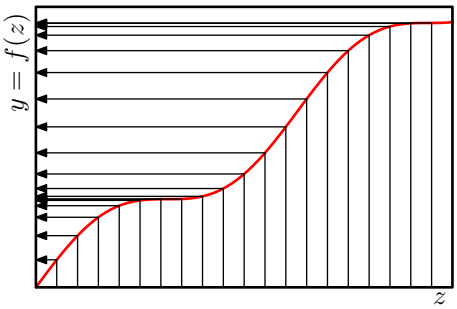


Figure 11.4: Transforming variables through a nonlinear function results in a changed probability density for the transformed variable.

reading my notes. This is plotted in Figure 11.3: the benefit of reading the notes evolves over time as the blue curve, the probability of your reading them at some point in time is indicated by the red curve (this is a complex function with local optima around the time of the actual lectures). The computation of the expectation of the benefit is indicated by the green line, which was computed numerically.

11.2 Basic sampling algorithms

The question now is how to get such "sample points" such that the number of points in a region is proportional to the probability of that region. When transforming continuous variables, the probability density is transformed by the determinant of the Jacobian. This can be seen intuitively for one-dimensional variables, as illustrated in Figure 11.4: we want that for any value of the original variable x, the probability of falling in an infinitesimally small region around x be the same as falling in an infinitesimally region around the corresponding $y = f(x)$ in the transformed space. Therefore, we have

$$p(y)\delta y \simeq p(x)\delta x \tag{11.4}$$

In the limit for infinitesimally small regions, we get

$$p(y) = p(x)\left|\frac{dx}{dy}\right| \ . \tag{11.5}$$

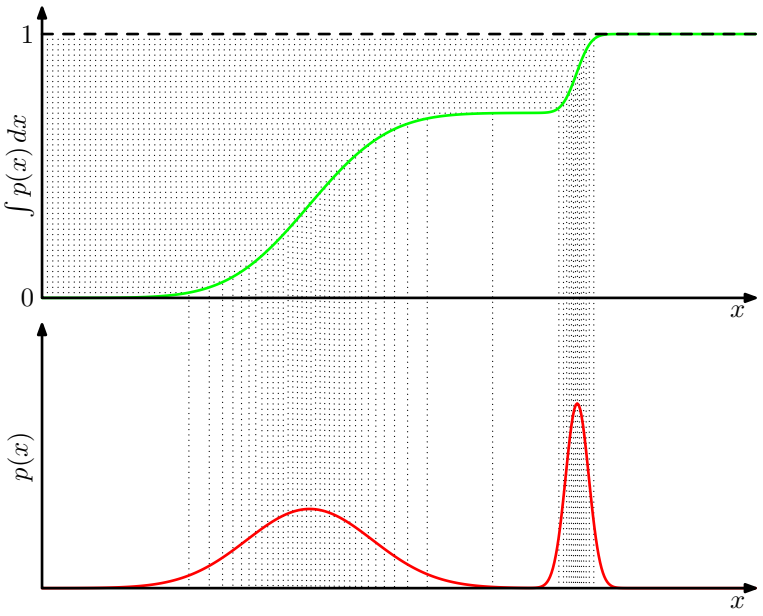


Figure 11.5: Transforming uniformly distributed samples in the interval $[0, 1]$ by the inverse CDF of the desired PDF results in samples with the desired distribution. See text for more details.

As an aside, notice that this only holds as long as the function is bijective (maps each value of x to one and only one value of y, and vice versa), but this can be extended by summing over sections of the function which are bijective. This is not an issue in our case, however. If we integrate both sides of the equation, we get

$$\int_{-\infty}^{y} p(y')\mathrm{d}y' = p(x)\,x \tag{11.6}$$

If we take x to have a uniform distribution in the range $[0, 1]$, then $p(x) = 1$ for all values of x, so that $x = h(y)$, the cumulative distribution function (CDF) of y. We can therefore transform a sample by the inverse CDF, $y = h^{-1}(x)$, to obtain samples y with the desired density function.

This is illustrated in Figure 11.5

11.3 Rejection Sampling

Rejection sampling is a method to generate samples from an unnormalised probability density function. The idea is as follows. Imagine that we should have a function that describes the distribution of a variable of interest, but our function is not a proper PDF because we do not know the normalisation constant that would make the area under the curve equal to one. Such functions are very common, for example, in Markov Random Fields (section 6.2), where the use of potential functions makes the computation of the normalisation constant exponential in the number of variables. Let us call this function $\tilde{p}(x)$, such that

$$p(x) = \frac{1}{Z}\tilde{p}(x) \tag{11.7}$$

This function $\tilde{p}(x)$ is depicted by a red line in Figure 11.6.

We can then devise a scheme to generate samples from the distribution $p(x)$ by taking any other distribution $q(x)$ from which we know how to sample, for example by transforming uniform samples using the basic sampling algorithm described above. We now find a constant k, such that $kq(x) \leq \tilde{p}(x)$ for all x.

We first sample from $q(x)$, and for each of the generated samples x, we generate a uniformly distributed sample u in the range $[0, kq(x)]$. If the sample $u > \tilde{p}(x)$, we reject the sample, otherwise we accept it and

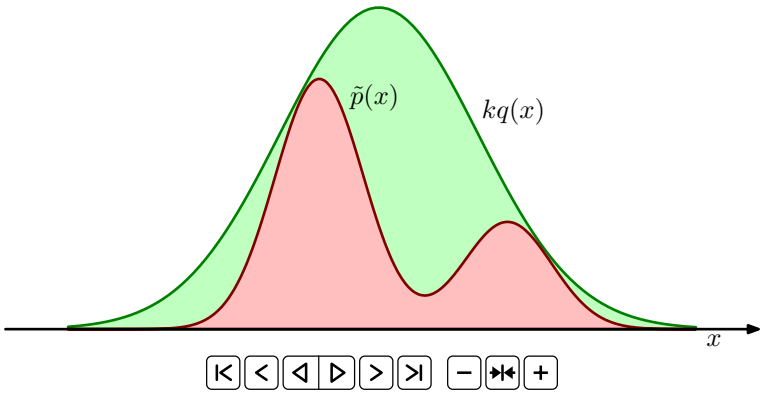


Figure 11.6: Illustration of rejection sampling

keep a sample $z = x$. The probability of a sample z is then given by

$$p(z) = q(x)\, p(z|x) \tag{11.8}$$

$$= q(x)\, \frac{\tilde{p}(x)}{kq(x)} \tag{11.9}$$

$$= \frac{Z}{k} p(x) \tag{11.10}$$

so that $p(z) \propto p(x)$: the distribution of z is the same as the distribution of x, and the probability of a sample z is the probability of x multiplied by the probability that it should be rejected uselessly. If $Z = k$, then $p(z) = p(x)$ and no samples are rejected. This is only possible if $p(x) = q(x)$, as we must scale $q(x)$ to be as least $p(x)$ for any value of x. Notice how the value of k affects the efficiency of the algorithm: the larger k, the lower the probability that a sample gets accepted.

This problem gets exacerbated in higher dimensions. TODO: link back to intuitions about high-dimensional spaces.

11.3.1 Adaptive Rejection Sampling

A clear problem with rejection sampling is that we need to keep k as small as possible, but that finding the minimum of k for which $k\, q(x) \geq \tilde{p}(x)$ is not always analytically possible. For distributions which are log-concave, a different approach can be used: instead of finding a fixed upper bound for the density that we want to sample from, we iteratively refine an upper bound while sampling. This works as follows: the (possibly unnormalised) log-probability function $\ln \tilde{p}(x)$ is bounded by piecewise linear segments, initially computed by the derivative to $\ln \tilde{p}(x)$ in a number of locations. These linear segments in log-probability result in exponential distributions in probability space, as illustrated in Figure 11.7. Rejection sampling is used with this proposal distribution, and whenever a sample gets rejected, the point is added to the set of centres where the proposal distribution is tight with $\tilde{p}(x)$. As a result, the proposal distribution will approximate the real distribution better, and the number of rejections diminishes as the number of samples increases.

11.4 Importance Sampling

Rejection sampling compensates for the difference between the proposal distribution $q(x)$ and the real distribution $p(x)$ through an extra sampling step which rejects samples proportionally to the difference between the distributions. As a consequence, there are, in a sense, two approximations: we approximate the distribution $q(x)$ by sampling, and approximate the difference between $p(x)$ and $q(x)$ by sampling again. Instead,

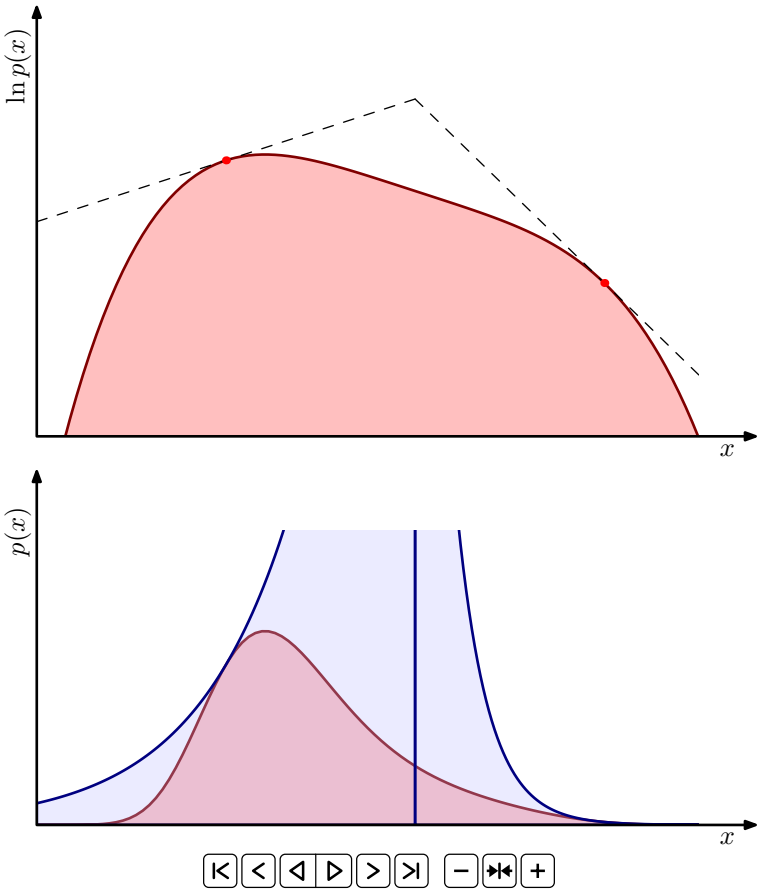


Figure 11.7: Illustration of how a log-concave distribution can be iteratively approximated by piecewise log-linear (exponential) distributions

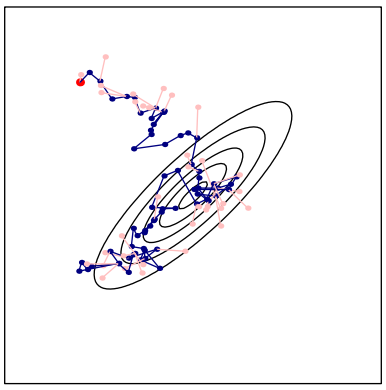


Figure 11.8: Illustration of Markov Chain Monte Carlo sampling with the Metropolis algorithm, where the desired distribution is a multivariate Gaussian with full covariance matrix, and the proposal distribution is a spherical Gaussian distribution. The connections between the samples show the order in which they were proposed; blue samples were accepted while red samples were rejected.

it is possible to replace the second sampling step by a weight. The expectation that we compute can be approximated as:

$$\mathbb{E}[f] \simeq \frac{1}{L} \sum_{l=1}^{L} \frac{p(\mathbf{z}^{(l)})}{q(\mathbf{z}^{(l)})} f(\mathbf{z}^{(l)}) \tag{11.11}$$

Doing so has two advantages: (1) we don't need to find a constant k by which to scale the proposal distribution, and (2) the weights are an exact correction for the difference between the proposal distribution and the target distribution, that rejection sampling approximates by sampling. On the other hand, importance sampling may cause problems which are hard to diagnose: if our proposal distribution is very different from our target distribution, we will obtain many sample with very low weight (and probably no samples with a large weight). Our estimated expectation will therefore be very different from the true expectation. In contrast, in the same situation rejection sampling would basically reject all samples.

11.5 Markov Chain Monte Carlo

Markov Chain Monte Carlo (MCMC) methods are sampling methods where the proposal distribution $q(\mathbf{z})$ forms a Markov chain, so that the next proposal depends on the previous sample. As a result, samples are not independent so that we need many more samples to obtain a good estimate of our distribution. However, the effectiveness of the sampling method does not decrease with the dimensionality of the space, so that MCMC methods are actually useful in many practical applications, where we have multiple variables, and where basic sampling algorithms are not applicable.

Let us create a Markov chain of samples, where the probability of the next sample given the previous sample is given by $q(\mathbf{z}_t|\mathbf{z}_{t-1})$. We are interested in the overall distribution of the samples, $p(\mathbf{z}_{1:T})$, and we want this distribution to remain constant as we get more samples. As usual, we may not be able to represent this distribution, but we do need to be able to compute it up to an unknown normalisation constant for any value of $\mathbf{z}$. Similarly to rejection sampling, we will accept or reject the samples obtained from the proposal distribution $q(\mathbf{z}_t|\mathbf{z}_{t-1})$ according to a criterion that will make the overall distribution of the accepted samples converge to our desired distribution $p(\mathbf{z})$.

11.5.1 The Metropolis Algorithm

The Metropolis algorithm is such a sampling algorithm. It assumes that the proposal distribution is symmetric — that is, $q(\mathbf{z}|\mathbf{z}') = q(\mathbf{z}'|\mathbf{z})$ — and that $q(\mathbf{z}|\mathbf{z}') \neq 0$ for all $\mathbf{z}$ and $\mathbf{z}'$. Based on these assumptions,

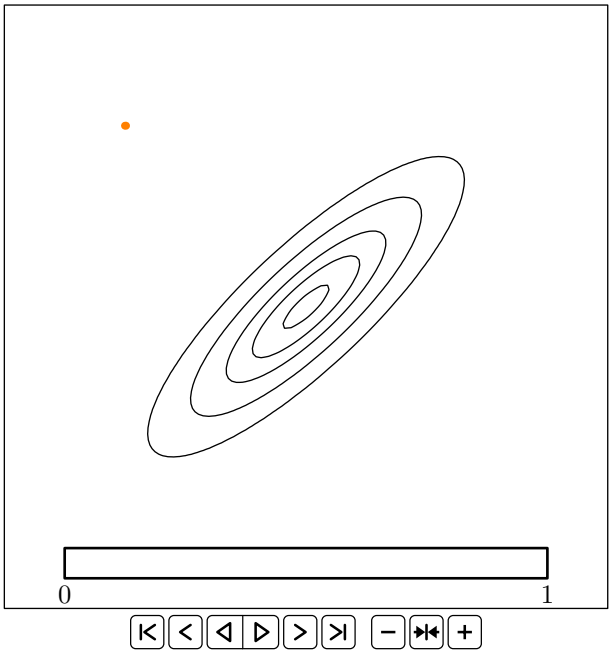


Figure 11.9: Animation illustrating how MCMC sampling explores the sample space. We start at a random location, indicated by the dark red dot. We then sample from the proposal distribution, and if the ratio between the new sample and the old sample is larger than one (indicated by a green arrow bar), we accept the sample unconditionally. If the ratio is smaller than one (indicated by a split red/green bar), we generate a uniform sample in the range $(0 \ldots 1)$, and accept the proposed sample if the value of the uniform sample falls below the computed ratio.

we can design a scheme for accepting samples from the proposal distribution where we accept a proposed sample $\mathbf{z}^*$ with probability

$$A(\mathbf{z}^*|\mathbf{z}_{t-1}) = \min\left(1, \frac{\tilde{p}(\mathbf{z}^*)}{\tilde{p}(\mathbf{z}_{t-1})}\right) . \tag{11.12}$$

That is, we always accept a sample if the probability of the proposed sample is higher than the probability of the previous sample, and only accept it with probability equal to the ratio between the probabilities of the new and the old sample if it's lower. This is crucial, as it means that we don't just find the areas with highest probability, but also that we end up in areas of lower probability (proportionally to the density in those areas). Importantly, if the sample is not accepted, it is not simply rejected: instead, a new sample $\mathbf{z}_t$ with the value of the previous sample $\mathbf{z}_{t-1}$ is created. This is illustrated in Figure 11.9, where the radius of the dot depicting each sample indicates how often the sample is included in the chain.

We can see that, using this scheme, the chain will be invariant, meaning that if the probability distribution of the samples is $p(\mathbf{z})$, sampling further from the chain will result in new samples with the same distribution $p(\mathbf{z})$. The way to see this is that our Markov chain exhibits detailed balance: $p(\mathbf{z})\, q(\mathbf{z}'|\mathbf{z})\, A(\mathbf{z}'|\mathbf{z}) = p(\mathbf{z}')\, q(\mathbf{z}|\mathbf{z}')\, A(\mathbf{z}|\mathbf{z}')$. This is easy to prove, since

$$\begin{aligned}
p(\mathbf{z})\, q(\mathbf{z}'|\mathbf{z})\, \min\left(1, \frac{p(\mathbf{z}')}{p(\mathbf{z})}\right) &= \min\left(p(\mathbf{z})\, q(\mathbf{z}'|\mathbf{z}), \frac{p(\mathbf{z}')\, p(\mathbf{z})\, q(\mathbf{z}'|\mathbf{z})}{p(\mathbf{z})}\right) && (11.13)\\
&= \min\left(p(\mathbf{z})\, q(\mathbf{z}'|\mathbf{z}), p(\mathbf{z}')\, q(\mathbf{z}'|\mathbf{z})\right) && (11.14)\\
&= \min\left(p(\mathbf{z})\, q(\mathbf{z}'|\mathbf{z}), p(\mathbf{z}')\, q(\mathbf{z}|\mathbf{z}')\right) && (11.15)\\
&= p(\mathbf{z}')\, q(\mathbf{z}|\mathbf{z}')\, \min\left(\frac{p(\mathbf{z})}{p(\mathbf{z}')}, 1\right) && (11.16)
\end{aligned}$$

In this derivation, we used normalised probabilities $p(\mathbf{z})$ instead of the unnormalised versions used in Eq. (11.12), which we can do since the normalisation constant is, by definition, a constant, so that $\frac{\tilde{p}(\mathbf{z})}{\tilde{p}(\mathbf{z}')} = \frac{p(\mathbf{z})}{p(\mathbf{z}')}$. Detailed balance ensures that the Markov chain is invariant:

$$\sum_{\mathbf{z}} p(\mathbf{z})\, q(\mathbf{z}'|\mathbf{z})\, A(\mathbf{z}'|\mathbf{z}) = \sum_{z} p(\mathbf{z}')\, q(\mathbf{z}|\mathbf{z}')\, A(\mathbf{z}|\mathbf{z}') = p(\mathbf{z}') \underbrace{\sum_{z} q(\mathbf{z}|\mathbf{z}')\, A(\mathbf{z}|\mathbf{z}')}_{=1} = p(\mathbf{z}') \tag{11.17}$$

Of course, just because our Markov chain is invariant once it has reached the desired distribution does not mean that it will converge to that distribution from any initialisation. This is where our second assumption — that the proposal distribution be non-zero for every $\mathbf{z}, \mathbf{z}'$ — comes into play, as it ensures that as the number of samples n approaches infinity, the distribution of the samples approaches the desired distribution Neal [1993].[2]

11.5.2 The Metropolis-Hastings algorithm

The Metropolis algorithm can be relaxed such that the proposal distribution need not be symmetric, by adjusting the probability so as to compensate for the asymmetry of the proposal distribution. The modified acceptance probability is given by

$$A(\mathbf{z}|\mathbf{z}') = \min\left(1, \frac{\tilde{p}(\mathbf{z})\, q(\mathbf{z}'|\mathbf{z})}{\tilde{p}(\mathbf{z}')\, q(\mathbf{z}|\mathbf{z}')}\right) \tag{11.18}$$

It is easy to show that the Markov chain exhibits detailed balance, without the symmetry constraint of the Metropolis algorithm. Notice, however, that the rejection rate is different, so that both sampling algorithms are not necessarily equally efficient.

[2] In fact, the constraint is a little looser than described here: it is sufficient for the proposal distribution $q(\mathbf{z}|\mathbf{z}')$ to be nonzero for any value $\mathbf{z}$ for which $p(\mathbf{z})$ is nonzero.

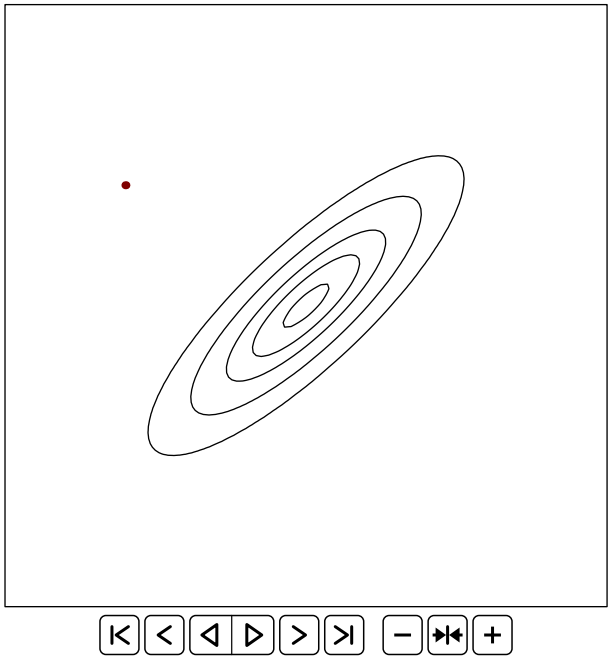


Figure 11.10: Illustration of Gibbs sampling. We sample each dimension in turn, from the conditional distribution over that dimension, given all other dimensions. In the animation, this is indicated by the grey line (which really should be drawn in three dimensions, of course). All samples are kept, see text for details.

11.5.3 Gibbs sampling

The Metropolis-Hastings algorithm can be applied to a particular form of sampling, where the proposal distribution is given by the conditional distribution of one dimension z_i of the variable given all other dimensions, $p(z_i|\mathbf{z}_{\neg i})$, under the distribution we are sampling from. If this conditional distribution can be computed (up to a normalisation constant) and sampled from, we obtain a very effective sampling scheme, since the acceptance probability of the Metropolis-Hastings algorithm becomes one:

$$\frac{p(\mathbf{z})\,p(z'_i|\mathbf{z}_{\neg i})}{p(\mathbf{z}')\,p(\mathbf{z}_i|\mathbf{z}'_{\neg_i})} = \frac{p(z_i)\,p(\mathbf{z}_{\neg i})\,p(z'_i|\mathbf{z}_{\neg i})}{p(z'_i)\,p(\mathbf{z}'_{\neg i})\,p(\mathbf{z}_i|\mathbf{z}'_{\neg_i})} \tag{11.19}$$

$$= \frac{p(z_i|\mathbf{z}'_{\neg i})\,p(\mathbf{z}_{\neg i})\,p(z'_i|\mathbf{z}_{\neg i})}{p(z'_i|\mathbf{z}'_{\neg i})\,p(\mathbf{z}'_{\neg i})\,p(\mathbf{z}_i|\mathbf{z}'_{\neg_i})} \tag{11.20}$$

$$= 1 \text{ since } \mathbf{z}_{\neg i} = \mathbf{z}'_{\neg i} \tag{11.21}$$

11.5.4 MCMC in practice

MCMC sampling has a few particularities when compared to the other sampling algorithms that we've seen. These need to be taken into account in any practical implementation.

1. Although the chain is guaranteed to converge to the desired distribution, the initial samples of the chain can have a very different distribution, especially if the chain is started in a low-probability region. In practice, it is therefore important to have a *burn-in* period of samples that are discarded.

2. It is typically not necessary to store the samples. If we are computing an expectation, we just keep the sum of the function values for each sample that is used, and discard the samples themselves. If we do need to keep the samples, in the case of the Metropolis and Metropolis-Hastings algorithms, it is often advantageous to store samples and counts representing how often a sample is used (due to rejections), rather than to keep duplicate samples.

3. Especially in the case of Gibbs sampling, if different dimensions of our probability distribution are strongly correlated, the sampling scheme becomes very inefficient. Consider the case of a 2D Gaussian distribution with strong correlations between the dimensions: in this case, Gibbs sampling will result in extremely steps being taken, and it will take very many samples to move from one end of the distribution to the other.

4. If our probability distribution is multi-modal (it has multiple maxima) separated by areas of very low probability, it will be very unlikely for the chain to transition from one area of high probability to the other. One area will be well explored, and the samples will seem to have give a good representation of the distribution, but in reality their distribution will be very different from the distribution we want to obtain. This is a real problem that has no good general solution, but multiple starts from random initialisations can at least indicate whether this is happening.

Appendix A

Lagrange multipliers

This text represents my attempt at explaining Lagrange multipliers. It is largely based on Bishop's book *Pattern Recognition and Machine Learning* but endeavours to make things short and easily understood by means of illustrative examples.

Consider a multivariate function that we would like to optimise. In machine learning, such a function will typically be some objective function of the data. For example, we may want to optimise the parameters of a probability density function (PDF) representing the distribution of the data. Often, we do not simply want to find the maximum of the function, but we also want to apply some constraints. For example, in the case of parameter optimisation of a PDF, we will want to ensure that the rules of probabilities are not violated.

In this explanation, we consider only a 2D case to help visualisation, but of course the technique is more widely applicable to higher-dimensional cases.

A.1 The function

Consider a two-dimensional feature space. The two features, x_1 and x_2 form the domain of the function we want to optimise. In this case, let us consider the function

$$f(\mathbf{x}) = \exp(-\mathbf{x}^\top \mathbf{x}) \tag{A.1}$$

which is shown in Figure A.1. Remember that the third dimension in this figure is not part of the domain of the function. It should rather be considered as a property of the input space and it is, therefore, perhaps more intuitive to consider the function as a colouring of the input space rather than as a third dimension: this is shown in the right-hand plot of Figure A.1.

In this latter figure, the gradient of the function, $\nabla f(\mathbf{x})$, is also depicted in the form of a vectorfield. Following the gradient in the input space will lead us to increasing values of $f(\mathbf{x})$. This is, of course, important when we want to find the optimum of $f(\mathbf{x})$. We will see in a minute that it is also important to find the constrained optimum of the function.

A.2 The constraint

The constraint we want to impose must be of the form $g(\mathbf{x}) = 0$. It imposes a relationship between the dimensions of the input space, so that one degree of freedom is taken away: if we know $n-1$ dimensions, we can compute the nth dimension using the constraint function. The constraint, therefore, defines an $(n-1)$-dimensional manifold in the n-dimensional input space.

Let us, in this 2D case, consider the 1D manifold given by the constraint $x_1 = 1/x_2$. In the form $g(\mathbf{x}) = 0$, this becomes:

$$x_1 x_2 - 1 = 0 \tag{A.2}$$

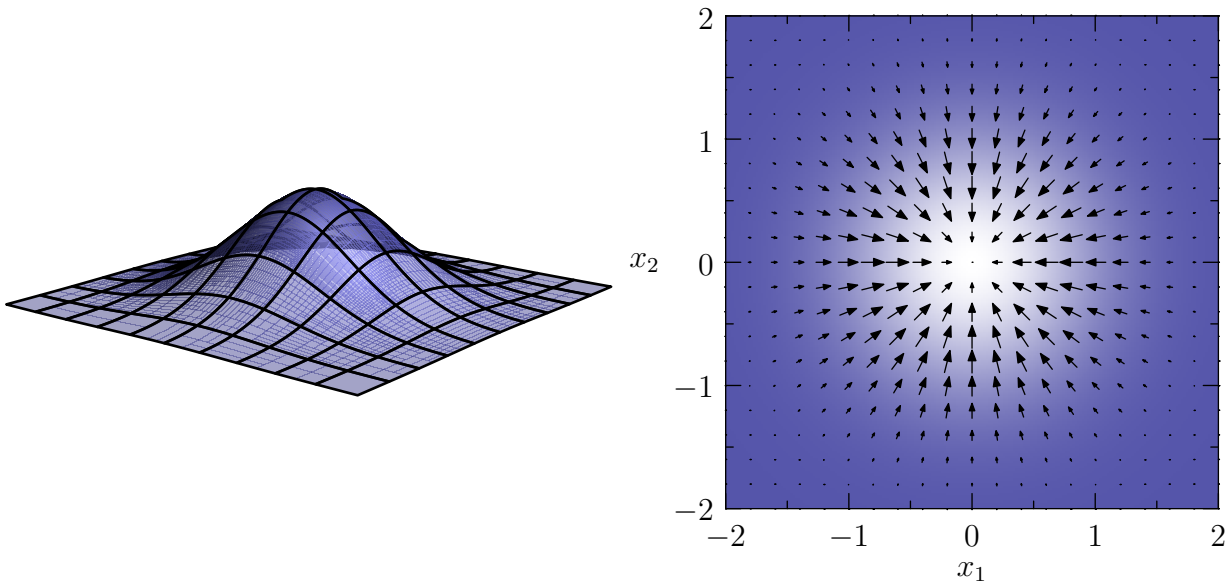


Figure A.1: Plot of the function $f(\mathbf{x})$ to be optimised

This function defines a line in our two-dimensional space, shown in dark red in Figure A.2. The value of $g(\mathbf{x})$ is represented by the colour intensity in this figure, so that a lighter colour indicates a higher value. If we look at the gradient of this function (shown by black arrows in the same figure), we see that it is always perpendicular to the line $g(\mathbf{x}) = 0$. It must be, of course, since it indicates in which direction to go in order to increase $g(\mathbf{x})$, and our line is defined as $g(\mathbf{x})$ equal to a constant.

Now we want to find the maximum of the function $f(\mathbf{x})$ on the manifold defined by $g(\mathbf{x}) = 0$. At the location of that maximum, that the gradient of $f(\mathbf{x})$ must be perpendicular to the manifold $g(\mathbf{x}) = 0$. We can see that this is the case, because if the gradient wasn't perpendicular, we could project it on the manifold $g(x) = 0$ and follow that projection to get to a higher point (if we were looking for a maximum, or go in the opposite direction to get to a lower point, if we were looking for a minimum). The original point where the gradient is not perpendicular to the manifold cannot, therefore, be an optimum. As a consequence, the gradient of $f(\mathbf{x})$ and the gradient of $g(\mathbf{x})$ must be parallel to each other at the location of the optimum, as illustrated in Figure A.3.

A.3 Finding the constrained optimum

We must find a point where the gradients of $f(\mathbf{x})$ and $g(\mathbf{x})$ are parallel (but can be of different length, and different sign), which we can express as follows:

$$\nabla f(\mathbf{x}) + \lambda \nabla g(\mathbf{x}) = 0, \tag{A.3}$$

where λ is a scalar constant. We now have a system with $D + 1$ unknowns (the D dimensions of $\mathbf{x}$, and λ). For convenience, we can introduce the following function called the *Lagrangian*:

$$f(\mathbf{x}) + \lambda g(\mathbf{x}) \tag{A.4}$$

Taking the first derivative of the Lagrangian with respect to each of the elements of $\mathbf{x}$ and setting the resulting quantities equal to zero, results in D equations. Doing the same with respect to λ results in our original constraint $g(\mathbf{x}) = 0$, so that we have $D + 1$ equations with $D + 1$ unknowns.

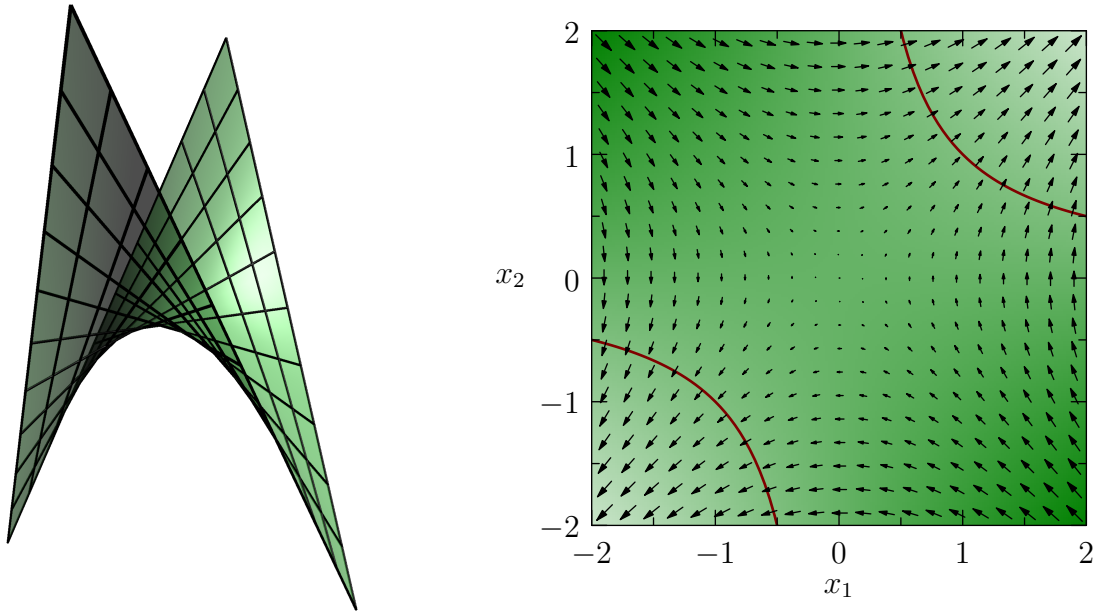


Figure A.2: Constraint function $g(\mathbf{x})$. The left plot depicts the function as an extra dimension, while the right plot uses colour intensity to indicate the function's value. The dark red line is defined by $x_1 x_2 - 1 = 0$

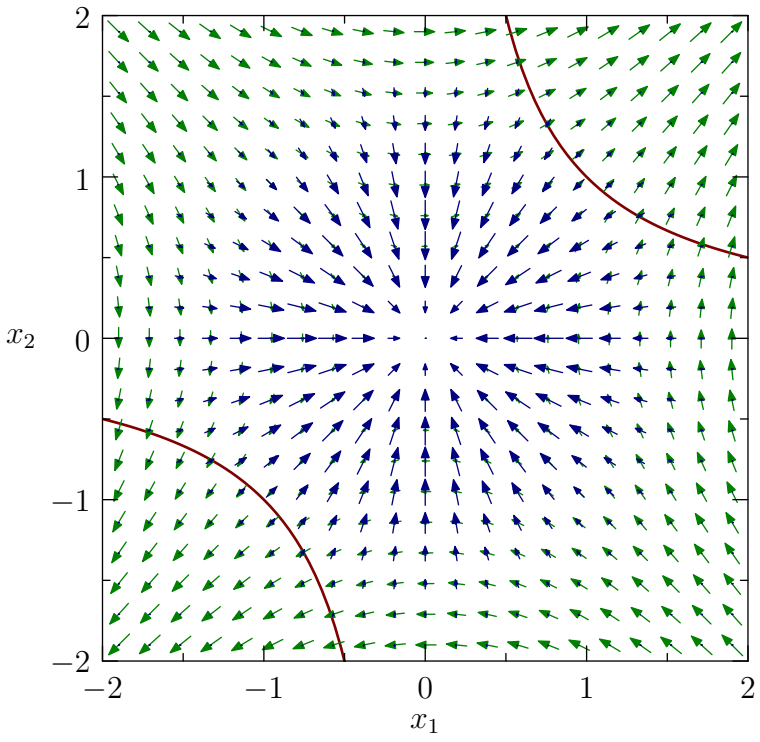


Figure A.3: Gradient of the function $f(\mathbf{x})$ to be optimised (blue) and gradient the constraint function $g(\mathbf{x})$ (green). $g(\mathbf{x}) = 0$ is shown by the dark red line.

A.3.1 Worked out example

In the case of the problem we used throughout this explanation, we have the following function and constraint:

$$f(\mathbf{x}) = \exp(-\mathbf{x}^\top \mathbf{x}) \tag{A.5}$$
$$g(\mathbf{x}) = x_1 x_2 - 1 = 0 \tag{A.6}$$

The Lagrangian $\mathcal{L}$ is therefore given by:

$$\mathcal{L} = \exp(-\mathbf{x}^\top \mathbf{x}) + \lambda(x_1 x_2 - 1). \tag{A.7}$$
$$= \exp(-x_1 x_1 - x_2 x_2) + \lambda(x_1 x_2 - 1) \tag{A.8}$$

Taking the partial derivatives and setting them equal to zero results in the following set of equations:

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial x_1} = 2\exp(-\mathbf{x}^\top \mathbf{x})(-x_1) + \lambda x_2 = 0 & \text{(A)} \\ \frac{\partial \mathcal{L}}{\partial x_2} = 2\exp(-\mathbf{x}^\top \mathbf{x})(-x_2) + \lambda x_1 = 0 & \text{(B)} \\ \frac{\partial \mathcal{L}}{\partial \lambda} = x_1 x_2 - 1 = 0 & \text{(C)} \end{cases} \tag{A.9}$$

So all that's left to do is to solve this set of equations for x_1 and x_2, to find the value of $\mathbf{x}$ on $g(\mathbf{x}) = 0$ for which $f(\mathbf{x})$ is maximal. In this case, we can do this by solving (A) and (B) for λ, resulting in:

$$\lambda = 2\exp(\mathbf{x}^\top \mathbf{x})\ \frac{x_1}{x_2} \tag{A.10}$$
$$\lambda = 2\exp(\mathbf{x}^\top \mathbf{x})\ \frac{x_2}{x_1} \tag{A.11}$$

From this, we get that

$$\frac{x_1}{x_2} = \frac{x_2}{x_1} \tag{A.12}$$
$$x_1^2 = x_2^2 \tag{A.13}$$
$$x_1 = \pm x_2 \tag{A.14}$$

However, the solution $x_1 = -x_2$ violates our constraint (C), so that it is discarded. If $x_1 = x_2$, we get that

$$x_1^2 = 1 \tag{A.15}$$
$$x_1 = x_2 = \pm 1, \tag{A.16}$$

and our function is maximal at both $(1, 1)^\top$ and $(-1, -1)^\top$, as indicated by the red dots in Figure A.4.

A.4 Inequality constraints

It is sometimes necessary to impose inequality constraints rather than equality constraints, *i.e.*, constraints in the form $g(\mathbf{x}) \geqslant 0$. In this case, our problem falls into two possible categories: either $g(\mathbf{x}) > 0$, or $g(\mathbf{x}) = 0$. If $g(\mathbf{x}) > 0$, we find that the maximum of $f(\mathbf{x})$ falls in a region where $g(\mathbf{x})$ is larger than zero. In other words, we must simply find the maximum of $f(\mathbf{x})$ (where $\nabla f(\mathbf{x}) = 0$, and the constraint does not affect the result: the constraint is then said to be inactive. If $g(\mathbf{x}) = 0$, we need to solve the same problem as described above; the constraint is then said to be active.

Notice that in the case where the constraint is inactive, this also correspond to a stationary point in the Lagrangian function; where $\lambda = 0$. If the constraint is active, the gradients of $f(\mathbf{x})$ and $g(\mathbf{x})$ are non-zero-length parallel vectors, so that λ must be non-zero. In both cases, therefore, $\lambda g(\mathbf{x}) = 0$. The direction of the gradients is now important, however, because the constraint will only be active if the gradients have opposite signs, as can easily be seen in the 2D case illustrated in Figure A.3. We can therefore optimise the function with inequality constraints by optimising the Lagrangian function, subject to the constraints

$$\begin{cases} g(\mathbf{x}) & \geqslant 0 \\ \lambda & \geqslant 0 \\ \lambda g(\mathbf{x}) & = 0 \end{cases} \tag{A.17}$$

These are known as the *Karush-Kuhn-Tucker* conditions.

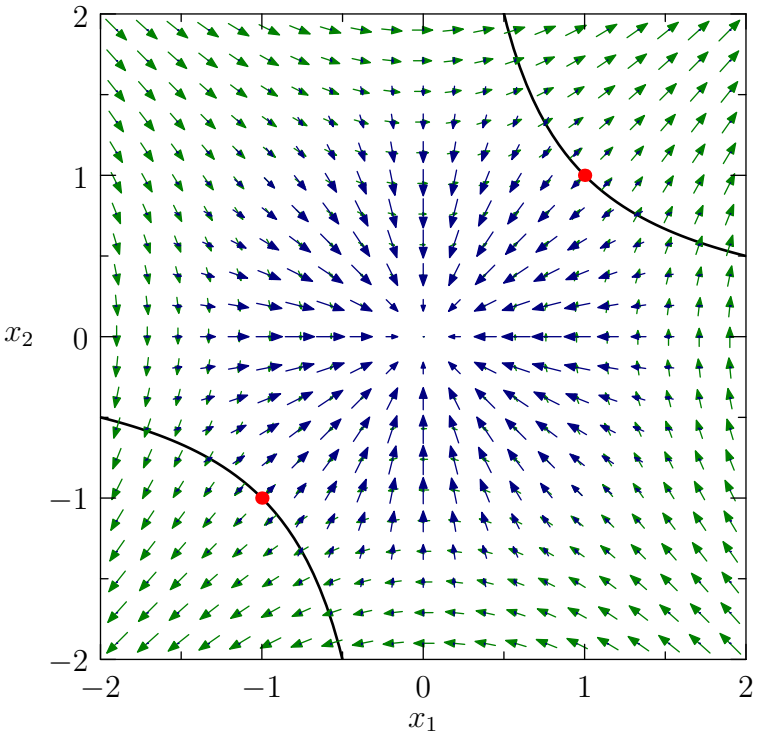


Figure A.4: Maximum of $\exp(\mathbf{x}^\top\mathbf{x})$ subject to $x_1x_2 = 1$

A.4.1 Worked-out example

Inactive constraint

Let's start with an example where the constraint is inactive. Consider the same function $f(\mathbf{x})$ as before, this time with the constraint

$$-x_1x_2 + 1 \geqslant 0. \tag{A.18}$$

This is the negative of our previous constraint function. Equality is obtained for this function for the same values of x_1 and x_2 as before, but the gradient is reversed. Solving the Lagrangian $\mathcal{L} = \exp(-\mathbf{x}^\top\mathbf{x}) + \lambda(1 - x_1x_2)$ subject to the KKT conditions gives us:

$$\begin{cases} 2\exp(-\mathbf{x}^\top\mathbf{x})(-x_1) - \lambda x_2 = 0 & \text{(A)} \\ 2\exp(-\mathbf{x}^\top\mathbf{x})(-x_2) - \lambda x_1 = 0 & \text{(B)} \\ 1 - x_1x_2 \geqslant 0 & \text{(C)} \\ \lambda \geqslant 0 & \text{(D)} \\ \lambda(1 - x_1x_2) = 0 & \text{(E)} \end{cases} \tag{A.19}$$

From (A) and (B) we get again that $x_1 = \pm x_2$. From (E) we get that if λ is non-zero, $x_1 = x_2 = \pm 1$. However filling $x_1 = x_2$ in (A) results in $\lambda = -\exp(\mathbf{x}^\top\mathbf{x})$, which must be negative and therefore violates (D). Therefore λ must be zero, and we get from (A) and (B) that, for non-infinite values of x_1 and x_2 (as these would violate (C),) $x_1 = x_2 = 0$. This is illustrated in Figure A.5

Active constraint

Another example, where the constraint is active, would be the following:

$$f(\mathbf{x}) = \exp(-\mathbf{x}^\top\mathbf{x}) \tag{A.20}$$

$$g(\mathbf{x}) = x_1x_2 - 1 \geqslant 0 \tag{A.21}$$

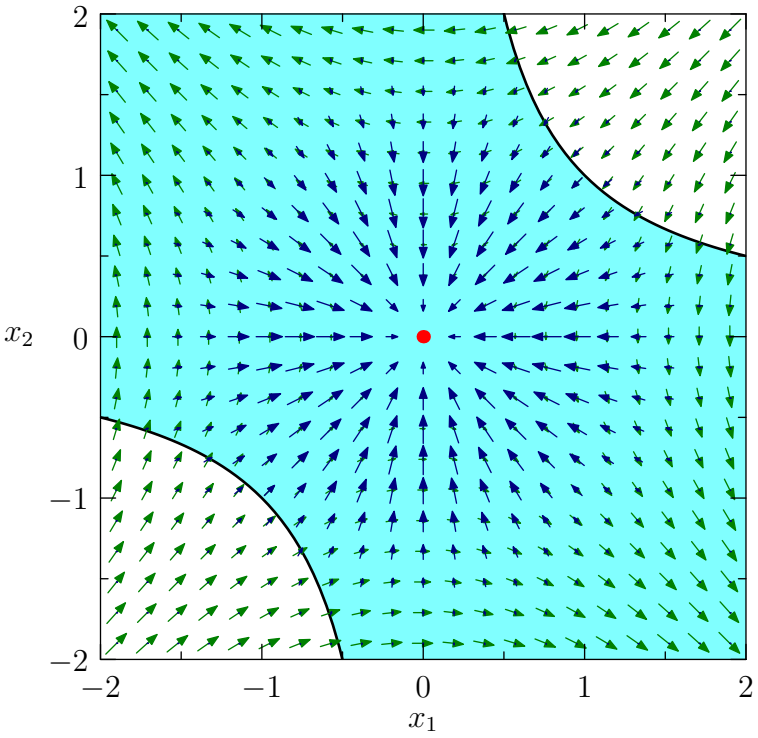


Figure A.5: Maximum of $\exp(\mathbf{x}^\top\mathbf{x})$ subject to $-x_1x_2+1 \geqslant 0$. In this illustration, the shaded area represents where the inequality constraint is fulfilled and the red dot indicates the constrained maximum of the function.

In this case, the conditions become:

$$\begin{cases} 2\exp(-\mathbf{x}^\top\mathbf{x})(-x_1) + \lambda x_2 = 0 & \text{(A)} \\ 2\exp(-\mathbf{x}^\top\mathbf{x})(-x_2) + \lambda x_1 = 0 & \text{(B)} \\ x_1x_2 - 1 \geqslant 0 & \text{(C)} \\ \lambda \geqslant 0 & \text{(D)} \\ \lambda(x_1x_2 - 1) = 0 & \text{(E)} \end{cases} \tag{A.22}$$

If we assume that the constraint is inactive, we get from (A) and (B) that $x_1 = x_2 = 0$, which violates C, or $x_1 = x_2 = \infty$, which violates (E). Therefore the constraint must be active, so that we fall in the same situation as exposed before for the equality constraints and obtain two constrained maxima: $(-1,-1)$ and $(1,1)$. The result is illustrated in Figure A.6.

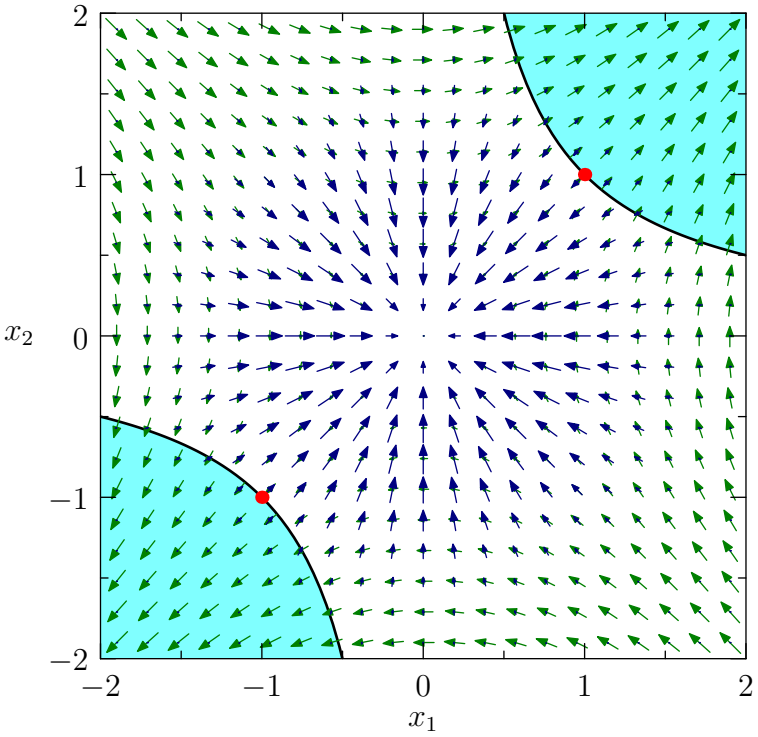


Figure A.6: Maximum of $\exp(\mathbf{x}^\top \mathbf{x})$ subject to $x_1 x_2 - 1 \geqslant 0$. In this illustration, the shaded area represents where the inequality constraint is fulfilled and the red dot indicates the maximum of the function.

Appendix B

Probability distributions

B.1 Bernoulli distribution

The Bernoulli distribution can be used to represent the probability of a single, binary event $x \in \{0, 1\}$. It has a single parameter, μ, which lies in the interval $[0, 1]$. The probability of an event being one is given by

$$p(x|\mu) = \mu^x (1-\mu)^{(1-x)} \tag{B.1}$$

$$\mathbb{E}[x] = \sum_{x \in \{0,1\}} x\, p(x) \tag{B.2}$$

$$= 1.\mu^1 + 0.(1-\mu)^0 \tag{B.3}$$

$$= \mu \tag{B.4}$$

B.2 Beta distribution

Blah. See Figure B.1

$$\text{Beta}(x; a, b) = \frac{\Gamma(a+b)}{\Gamma(a) + \Gamma(b)} x^{a-1}(1-x)^{b-1} \tag{B.5}$$

The posterior distribution of μ, after seeing $a-1$ positive examples and $b-1$ negative examples. It is is, therefore, the conjugate prior for the parameter of the Bernoulli distribution.

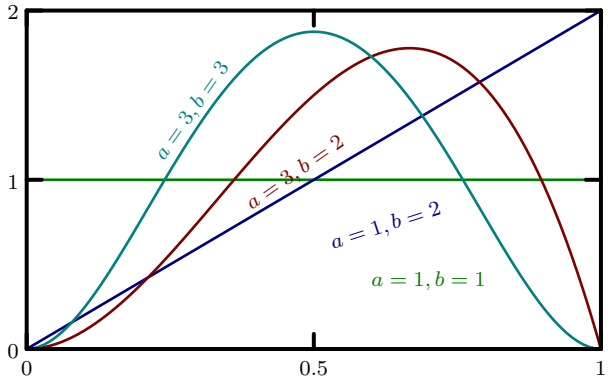


Figure B.1: Plot of the Beta distribution

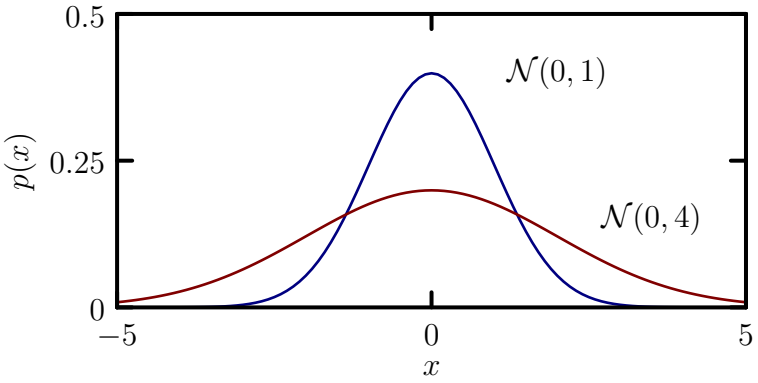
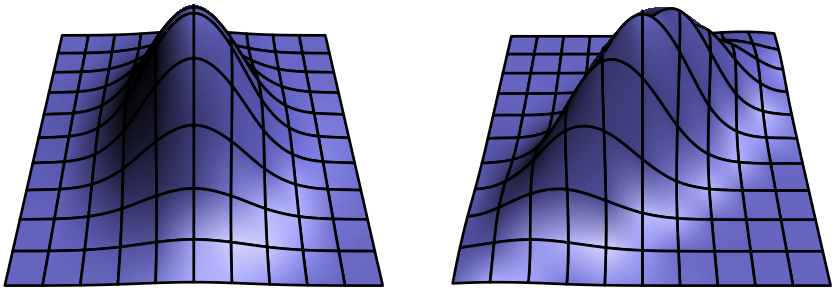


Figure B.2: Plot of the Gaussian distribution in one and two dimensions



$$\boldsymbol{\Sigma} = \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix} \qquad \boldsymbol{\Sigma} = \begin{bmatrix} 4 & 3 \\ 3 & 4 \end{bmatrix}$$

Figure B.3: Plot of 2-dimensional Gaussian distributions

B.3 Binomial distribution

B.4 Gaussian distribution

Appendix C

Dealing with products of many probabilities

One issue that is often encountered when implementing probabilistic models, is that the numbers we deal with become very small. If we have a thousand independent variables, their joint probability is given by the product of their individual probabilities. If each variable should have a probability $p(x_n) = 0.1$, their joint probability $p(x_1, \ldots, x_n) = 0.1^{1000}$ could not be represented as a double-precision floating point variable. Such collections of variables are two a penny,[1] especially when dealing with sequential data, and special care must then be taken to minimise numerical errors.

Two solutions are common: (1) rescale your probabilities periodically or (2) work with log-probabilities rather than probabilities. The first solution is often applied to sequential models such as HMM, but it requires extra bookkeeping work and is prone to errors. In this chapter, we expose how to work with the logarithm of probabilities instead.

C.1 Using log-probabilities

Probabilities (and probability densities) are, by definition constrained to lie in the interval $[0, 1]$. The natural logarithm of these quantities therefore lie in the interval $[-\infty, 0]$. This gives our computer a much larger dynamic range to work with, and drastically reduces the effect of numerical errors. When dealing with probabilities, three different operations are common: the product of two probabilities (to compute joint probabilities), the quotient of two probabilities (to compute conditional probabilities using Bayes' rule), and the sum of two probabilities (to marginalise out variables).

The product and quotient of probabilities is easily computed using log-probabilities, since $\log(p(x) \times p(y)) = \log p(x) + \log p(y)$ and $\log(p(x)/p(y)) = \log p(x) - \log p(y)$.

The sum of probabilities is a bit harder to deal with, but can be solved as follows. We want to compute $\log\big(p(x) + p(y)\big)$, when given $\log p(x)$ and $\log p(y)$. This can decompose this as follows:

$$\begin{aligned}
\log(p(x) + p(y)) &= \log(\exp \log p(x) + \exp \log p(y)) \\
&= \log(\exp \log p(x)(1 + \frac{\exp \log p(y)}{\exp \log p(x)})) \\
&= \log p(x) + \log(1 + \exp(\log p(y) - \log p(x))) \qquad \text{(C.1)}
\end{aligned}$$

Now, let us have a closer look at the function $\log(1 + \exp(x))$, depicted in Figure C.1: when x is a large negative number, it tends to zero, and when x is large it tends to x. In practice, using double-precision arithmetic, for $x \leqslant -44$, the difference between $\log(1 + e^x)$ and 0 is smaller than machine precision, and for $x \geqslant 44$ the difference between $\log(1 + e^x)$ and x is smaller than machine precision. In practice, the function only needs to be evaluated within that small range around zero.

[1] They occur often. Apparently, inhabitants of the US would use the expression "a dime a dozen" instead which, as McKay points out, does not leave much room for fluctuation in the exchange rate between pounds and dollars.

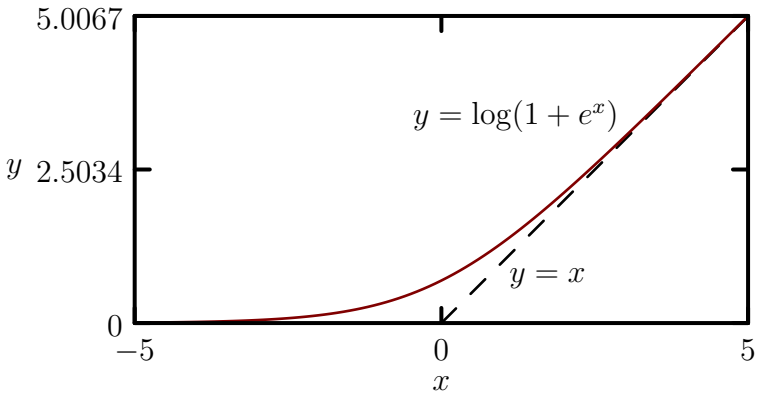


Figure C.1: Plot of the function $\log(1 + e^x)$

To summarise, when working with the logarithm of probabilities, we replace products with sums, quotients with subtractions, and sums with a special function which computes Eq. (C.1). This function, which I will call lse (for log-of-sum-of-exp), should really be part of any toolbox dealing with probabilities. For efficiency, and in order to deal with zero probabilities correctly ($\log 0 = -\infty$), which can lead to results being not-a-number when evaluating functions of two probabilities which are close to zero), it can be expedient to replace Eq. (C.1) with:

$$\text{lse}(\log p(x), \log p(y)) = \log(p(x) + p(y)) \tag{C.2}$$

$$= \begin{cases} \log p(x) & \text{if } \log p(x) > \log p(y) + 44 \\ \log p(y) & \text{if } \log p(y) > \log p(x) + 44 \\ \log(1 + e^{\log p(x) - \log p(y)}) & \text{otherwise,} \end{cases} \tag{C.3}$$

where the constant, 44 in this case, must be adapted to the precision of your machine.

When dealing with long summations, notice that $a + b + c = (a + b) + c$. We can therefore compute

$$\log(p(x) + p(y) + p(z)) = \text{lse}(\text{lse}(\log p(x), \log p(y)), \log p(z)) \tag{C.4}$$

This can then be iterated for summations of any length.

Appendix D

Jensen's inequality

Jensen's inequality states that, for variables λ_i for which $\sum_i \lambda_i = 1$, if $f(x)$ is a convex function, then:

$$\sum_i \lambda_i f(x_i) \geqslant f\left(\sum_i \lambda_i x_i\right) \tag{D.1}$$

If we further constrain the λ_i's to be non-negative, then we can interpret them as probabilities, and then the above equation can be rewritten as:

$$\mathbb{E}[f(x)] \geqslant f(\mathbb{E}[x]) \tag{D.2}$$

for any convex function $f(x)$. This is an important result, and we use it more than once in this course, but it is hard to get an intuitive feeling of. In the following, I try to provide you with just that intuition. The proofs are not important, they are really just provided to explain and accompany the figures — the figures are the important bit, first try to understand those.

Figure D.1 illustrates what a convex function is. A convex function is a function for which any chord, that is, any straight line between two points on the function, lies above the function.[1] We can express a value $\hat{x}$ lying between values a and b as $\hat{x} = a + \lambda'(b - a)$, with $0 \leqslant \lambda' \leqslant 1$, or, equivalently (and where we use $\lambda = 1 - \lambda'$ for convenience):

$$\hat{x} = \lambda a + (1 - \lambda)b \tag{D.3}$$

[1] Actually, that's a strictly convex function. With a convex function, the chord can also lie on the function in some segments; with a strictly convex function, the chord must lie strictly above the function.

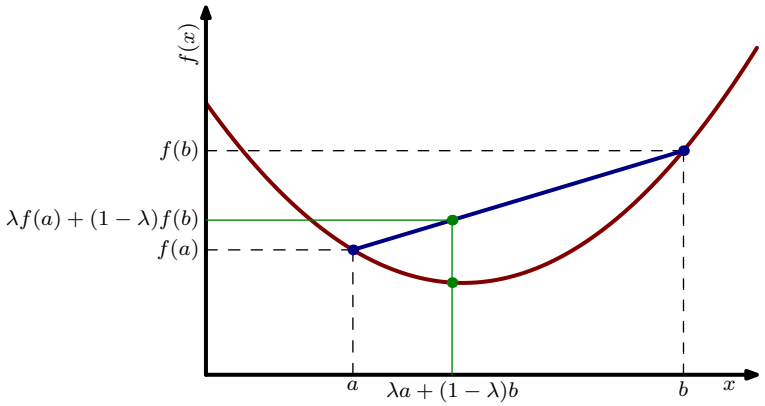


Figure D.1: A convex function (shown in red) and a chord between two random points on the function (shown in blue)

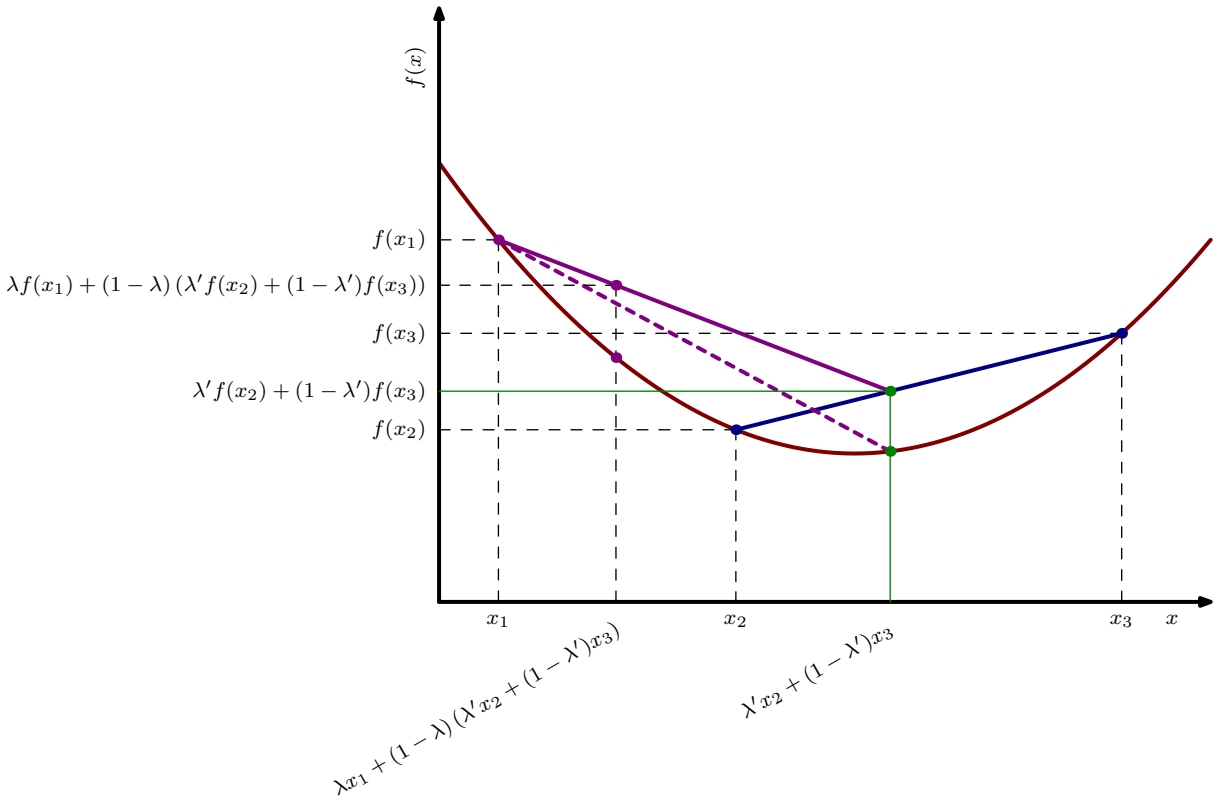


Figure D.2: Illustration of Jensen's inequality. A convex function (shown in red) lies below the linear interpolation between the function values of two random points x_2 and x_3 (shown in blue). The linear interpolation between a third point, x_1 and any point on the original interpolation between x_2 and x_3 (shown in magenta) must therefore lie higher than the linear interpolation between x_1 and the corresponding point on the function between x_2 and x_3 (shown in dashed magenta), and therefore also higher than the function itself.

A convex function is, therefore, a function for which

$$\lambda f(a) + (1-\lambda)f(b) \geqslant f(\lambda a + (1-\lambda)b) \tag{D.4}$$

If we have three variables λ_1, λ_2 and λ_3 such that $\lambda_1 + \lambda_2 + \lambda_3 = 1$, then, since we have that

$$\lambda_1 f(x_1) + \lambda_2 f(x_2) + \lambda_3 f(x_3) = \lambda_1 f(x_1) + (1-\lambda_1)\left[\frac{\lambda_2}{1-\lambda_1} f(x_2) + \frac{\lambda_3}{1-\lambda_1} f(x_3)\right] \tag{D.5}$$

and, because the values of λ_i sum up to one,

$$\frac{\lambda_2}{1-\lambda_1} + \frac{\lambda_3}{1-\lambda_1} = \frac{\lambda_2 + (1-\lambda_2-\lambda_1)}{1-\lambda_1} = 1 \tag{D.6}$$

then, from Eq. (D.4), we have that

$$\frac{\lambda_2}{1-\lambda_1} f(x_2) + \frac{\lambda_3}{1-\lambda_1} f(x_3) \geqslant f(\frac{\lambda_2}{1-\lambda_1} x_2 + \frac{\lambda_3}{1-\lambda_1} x_3) \tag{D.7}$$

and

$$\lambda_1 f(x_1) + (1-\lambda_1) f\left(\frac{\lambda_2}{1-\lambda_1} x_2 + \frac{\lambda_3}{1-\lambda_1} x_3\right) \geqslant f\left(\lambda_1 x_1 + (1-\lambda_1)\left[\frac{\lambda_2}{1-\lambda_1} x_2 + \frac{\lambda_3}{1-\lambda_1} x_3\right]\right) \tag{D.8}$$

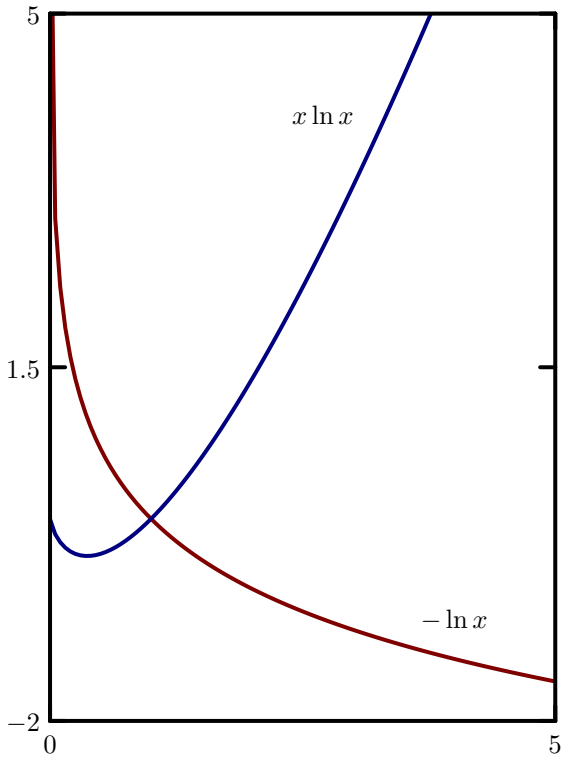


Figure D.3: Examples of useful convex functions

and therefore,

$$\lambda_1 f(x_1) + \lambda_2 f(x_2) + \lambda_3 f(x_3) \geqslant f\left(\lambda_1 x_1 + \lambda_2 x_2 + \lambda_3 x_3\right) \tag{D.9}$$

This reasoning can be applied recursively, so that we obtain Eq. (D.1) by induction. Examples of convex functions are shown in Figure D.3.

Bibliography

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 1 edition, October 2007. ISBN 0387310738.

H. Peter Clamann. Statistical analysis of motor unit firing patterns in a human skeletal muscle. *Biophysical Journal*, 9(10):1233–1251, October 1969. ISSN 0006-3495. doi: 10.1016/S0006-3495(69)86448-9.

Noel A.C. Cressie. *Statistics for Spatial Data*. Wiley Series in Probability and Statistics. Wiley-Interscience, rev sub edition, January 1993. ISBN 0471002550.

Carl H. Ek, Philip H. Torr, and Neil D. Lawrence. Gaussian process latent variable models for human pose estimation. In *Proceedings of the 2007 conference on Machine Learning for Multimodal Interfaces*, 2007.

Carl H. Ek, Jon Rihan, Philip H. Torr, and Neil D. Lawrence. Ambiguity modeling in latent spaces. In *Machine Learning for Multimodal Interaction*, volume 5237/2008 of *Lecture Notes in Computer Science*, pages 62—73. Springer, 2008.

D. C Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1):503528, 1989.

Radford M. Neal. Probabilistic inference using markov chain monte carlo methods. Technical Report CRG_TR_93-1, University of Toronto, Toronto, 1993.

A. Y Ng and M. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *Advances in NIPS*, 2002.

Judea Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, March 2000. ISBN 9780521773621.

K. B. Petersen and M. S. Pedersen. *The Matrix Cookbook*. Technical University of Denmark, 2006.

Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006. ISBN 0-262-18253-X.

Jack M. Wang, David J. Fleet, and Aaron Hertzmann. Gaussian process dynamical models for human motion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(2):283–298, 2008. ISSN 0162-8828. doi: http://doi.ieeecomputersociety.org/10.1109/TPAMI.2007.1167.