IAIN STYLES

# MACHINE LEARNING

# *Contents*

# *List of Figures*

# *Mathematical Preliminaries*

## *Linear Algebra*

Many problems in machine learning can be naturally expressed as systems of linear equations. In this section, we briefly describe how linear equations can be represented using the language of vectors and matrices.

Consider a set of $N$ numbers $(x_1, x_2, \ldots, x_N)$. These numbers could represent, for example, the coordinates of a point in 3D space; they could represent some measurement; they could even be the intensities of pixels in an image. We will require that they all have the same "unit" (distance, time, chemical concentraion, intensity etc). Such sets of numbers are extremely common in machine learning. Many machine learning methods will involve the construction of models that map one such set of numbers onto some other set of numbers, and the goal wil be to learn the model that performs this mapping. A common form of such a model is a *linear mapping* that constructs a second set of numbers $(y_1, y_2, \ldots, y_M)$ by adding up the $x$'s in some weighted combination, i.e:

$$\begin{aligned}
y_1 &= A_{11}x_1 + A_{12}x_2 + \cdots + A_{1N}x_N && (1)\\
y_2 &= A_{21}x_1 + A_{22}x_2 + \cdots + A_{2N}x_N && (2)\\
&\cdots && (3)\\
y_M &= A_{M1}x_1 + A_{M2}x_2 + \cdots + A_{MN}x_N && (4)\\
& && (5)
\end{aligned}$$

Note that the label on the $A'$s has two components: one corresponding to the label on the associated $y$ and one corresponding to the label on the associated $x$. This will be important.

It is clear that working with a model like this will quickly become very tiresome and tedious unless we can find a more efficient way to represent this problem. That language is linear algebra. In this problem, we have three mathematical "objects": the set of $x'$s, the set of $y'$s and the set of $a'$s. The $x$ and $y$ each represent something – a position in space, an image etc. The $A'$s define the relationsip between them. That is, given the $x$ and the $A'$s, one can calculate the $y$s.

A very convenient way of working with this sort of problem is to adopt the following notation. We consider the $x$ and $y$ to be *components* of a quantity called a vector, which is a convenient shorthand for the set. We write

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} \qquad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{pmatrix} \tag{6}$$

We will use lower-case, bold-face letters to represent vectors such as these. Note that we write the vector as a *column* rather than as a *row*.

We now need a similar representation for the $a$'s. The form of the equations above gives us a hint as to how we might right this down. Noting that the $a$'s have two labels, we will write them as a two-dimensional *matrix*

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1N} \\ A_{21} & A_{22} & \dots & A_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{M1} & A_{M2} & \dots & A_{MN} \end{pmatrix} \tag{7}$$

The labels (indices) on the $A$'s denote the row and column of the matrix they are in. We denote matrices using a bold-face upper-case letter.

This representation now allows us to write a compact shorthand for our linear mapping. We can simply write

$$\mathbf{y} = \mathbf{A}\mathbf{x} \tag{8}$$

We will describe $\mathbf{A}$ and $\mathbf{x}$ as being *multiplied*. Writing this out in full, we have

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1N} \\ A_{21} & A_{22} & \dots & A_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{M1} & A_{M2} & \dots & A_{MN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} \tag{9}$$

Our original equations define how the components of $\mathbf{A}$ and $\mathbf{x}$ are combined in matrix multiplication: each row of $\mathbf{y}$ is formed from the corresponding row of $\mathbf{A}$ multiplied component-wise with $\mathbf{x}$ and then summed. Notice that this requires that $\mathbf{A}$ has the same number of rows as $\mathbf{y}$ has components, and the same number of columns as $\mathbf{x}$ has components.

This defines a neat and tidy formalism through which we can reprent linear mappings. Notice that sets of simultaneous linear equations can be represented in this notation. For example, the equations

$$2a - b = 3 \tag{10}$$

$$a + b = 3 \tag{11}$$

can be equivalently written as

$$\begin{pmatrix} 2 & -1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \end{pmatrix} \tag{12}$$

## *Operations on matrices and vectors*

Matrices and vectors can be added together provided that they are of the same size (number of rows and columns). This is a simple operation: the result of an addition (or subtraction) is a matrix /vector of the same size, with the components added together. That is,

$$\begin{aligned} \mathbf{a} &= \mathbf{b}+\mathbf{c} \quad \rightarrow \quad a_i = b_i + c_i && (13) \\ \mathbf{A} &= \mathbf{B}+\mathbf{C} \quad \rightarrow \quad A_{ij} = B_{ij} + C_{ij} && (14) \\ & && (15) \end{aligned}$$

Similarly, multiplication by a scalar number multiplies each of the components:

$$\begin{aligned} \mathbf{a} &= \alpha\mathbf{b} \quad \rightarrow \quad a_i = \alpha b_i && (16) \\ \mathbf{A} &= \alpha\mathbf{B} \quad \rightarrow \quad A_{ij} = \alpha B_{ij} && (17) \\ & && (18) \end{aligned}$$

The *length* or *magnitude* of a vector can be calculated as the square-root of the sum of the squares of its components (courtesy of Pythagoras' Theorem for calculating the hypotenuse of a triangle):

$$|x| = \sqrt{x_1^2 + x_2^2 + \cdots + x_N^2} = \sqrt{\sum_{i=1}^{N} x_i} = \sqrt{\mathbf{x}^{\mathrm{T}}\mathbf{x}} \tag{19}$$

where $\mathbf{x}^{\mathrm{T}}$ is the *transpose* of $\mathbf{x}$, which is the vector written as a row rather than as a column. If one considers a vector to be a special case of a matrix with only one row/column, then it is easy to verify that this is entirely consistent with the rules of matrix/vector multiplication.

It is also possible to combine two matrices together. this is very similar to matrix-vector multiplication, except we have multiple columns:

$$\mathbf{A} = \mathbf{XY} \quad \rightarrow \quad A_{ik} = \sum_j X_{ij} Y_{jk} \tag{20}$$

For example:

$$\begin{pmatrix} 1 & 2 \\ 2 & 3 \\ 3 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 4 & 1 \\ 1 & 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1\times2+2\times1 & 1\times1+2\times3 & 1\times4+2\times2 & 1\times1+2\times1 \\ 2\times2+3\times1 & 2\times1+3\times3 & 2\times4+3\times2 & 2\times1+3\times1 \\ 3\times2+1\times1 & 3\times1+1\times3 & 3\times4+1\times2 & 3\times1+1\times1 \end{pmatrix} = \begin{pmatrix} 4 & 7 & 8 & 3 \\ 7 & 11 & 14 & 5 \\ 7 & 6 & 14 & 4 \end{pmatrix} \tag{21}$$

This can be used to form compositions of linear mappings: the matrix **A** represents the linear mapping **Y** followed by the linear mapping **X**. Note the order here: in the multiplication **XYx**, **X** is applied to **Yx**.

One special case of this is of particular interest. A special matrix that we will see quite frequently is the *identity* matrix

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix} \tag{22}$$

This matrix has the property that $\mathbf{Ix} = \mathbf{x}$ (and similarly $\mathbf{IA} = \mathbf{A}$). A special case of matrix compositions is the case where $\mathbf{XY} = \mathbf{I}$. That is, $\mathbf{XYx} = \mathbf{x}$. The linear transformation implemented by $\mathbf{X}$ has reversed the transformation implemented by $\mathbf{Y}$. We will refer to $\mathbf{X}$ as being the *inverse* of $\mathbf{Y}$ and will write this is $\mathbf{X} = \mathbf{Y}^{-1}$. As an example of where this can be useful, consider the linear equations we wrote down earlier:

$$\begin{pmatrix} 2 & -1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \end{pmatrix} \tag{23}$$

$$\mathbf{Ax} = \mathbf{y} \tag{24}$$

If we can find $\mathbf{A}^{-1}$, then we can do the calculation

$$\mathbf{Ax} = \mathbf{y} \quad \rightarrow \quad \mathbf{A}^{-1}\mathbf{Ax} = \mathbf{A}^{-1}\mathbf{y} \quad \rightarrow \quad \mathbf{Ix} = \mathbf{A}^{-1}\mathbf{y} \quad \rightarrow \quad \mathbf{x} = \mathbf{A}^{-1}\mathbf{y} \tag{25}$$

It is straightforward to verify that

$$\mathbf{A} = \begin{pmatrix} 2 & -1 \\ 1 & 1 \end{pmatrix} \longrightarrow \mathbf{A}^{-1} = \frac{1}{3} \begin{pmatrix} 1 & 1 \\ -1 & 2 \end{pmatrix}, \tag{26}$$

and this allows us to solve the equations to find that $a = 2$ and $b = 1$.

The inverse of a matrix can only be computed exactly for square matrices with certain properties. We will not go into the details of this, nor will we be concerned with the precise methods used to compute a matrix inverse. In fact, it is rare that we actually need to compute a matrix inverse explicitly: this is a computationally expensive operation, and there are alternative methods one case use. For the problem we have just considered, it is much more efficient (especially for very large systems of equations) to use a procedure such as Gaussian elimination to solve the problem directly without inverting $\mathbf{A}$.

Students who would like to develop their knowledge of linear algebra are advised to consider Gibert Strang's excellent online course which has been made available vi MIT Open Courseware and can be found at `https://ocw.mit.edu/courses/mathematics/18-06-linear-algebra-spring-2010/`.

## *A Very Brief Introduction to Differentiation*

A recurring theme in machine learning is the idea that we need to minimise some function that describes the cost or *loss* or error of a model with respect to some dataset. In many simple cases, we will do this using a process called *differentiation*.

The idea that we are working towards is that the ability of a model to match a set of data cannot be better than perfect, and therefore there exists a cost function that has an absolute lower bound of zero. Our model will not normally be able to reach this lower bound, but there will be some optimal value of its parameters that gets us as close as possible. If the optimal parameters are changed, even by a small amount, the cost of the model will increase, as shown in a very simple case in Figure 1, where the value $x = 5$ minimises the value of $y$.

The minimum point of a function that behaves in this way has a special property: at the point, it has a gradient, or slope, of zero (it is perfectly "flat") as indicated by the black line. The process of finding the minimum of the function then becomes one of finding the point where the gradient is zero.

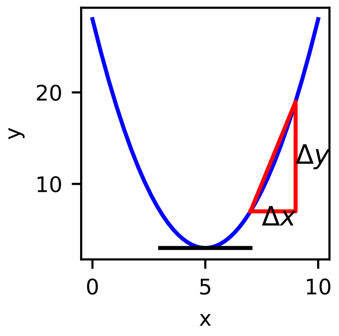

Figure 1: A schematic illustration of the process of computing the derivate (gradient) of a function.

The process of computing the gradient (also called the derivative) of an arbitrary function is known as "differentiation". This can be an intricate process and we will only sketch out the principles here. The basic idea is that we approximate the function as a straight line segment and compute the gradient of that straight line. We then make the straight line shorter and figure out how the derivative behaves as the length of the segment approaches zero. This will give the gradient at a point.

Consider the red triangle of width $\Delta x$ and height $\Delta y$ shown in Figure 1. The gradient of the hypotenuse of this triangle is $m = \Delta y / \Delta x$. At a general point x, the gradient of a function $f(x)$ will be

$$m \approx \frac{\Delta y}{\Delta x} = \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{27}$$

In order to improve this approximation, we will want to see how the gradient behaves as $\Delta x$ is reduced in size. Formally, we will write this as

$$m_x = \frac{\mathrm{d}y}{\mathrm{d}x} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} \tag{28}$$

Where $\lim \Delta x \to 0$ means "take $\Delta x$ to 0", $m_x$ is the gradient of $f(x)$as a function of $x$, and the notation $\frac{\mathrm{d}y}{\mathrm{d}x}$ is a formal way of writing the derivative which implies the process of taking the limit. Let us illlustrate with a couple of examples. First, one where the know the answer: $y = 3x + 2$, which we know to have $m = 3$.

$$m_x = \frac{\mathrm{d}y}{\mathrm{d}x} = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} = \frac{3(x+\Delta x) + 2 - 3x + 2)}{\Delta x} = \frac{3\Delta x}{\Delta x} = 3. \tag{29}$$

In fact, we didn't need to take a limit here because everything cancelled out. Let us try a different example: $y = x^2$:

$$\begin{aligned} m_x = \frac{\mathrm{d}y}{\mathrm{d}x} &= \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} && (30) \\ &= \lim_{\Delta x \to 0} \frac{(x+\Delta x)^2 - x^2)}{\Delta x} && (31) \\ &= \lim_{\Delta x \to 0} \frac{x^2 + 2x\Delta x + (\Delta x)^2 - x^2}{\Delta x} && (32) \\ &= \lim_{\Delta x \to 0} \frac{2x\Delta x + (\Delta x)^2}{\Delta x} && (33) \end{aligned}$$

Now, in the limit $\Delta x \to 0$, the term $(\Delta x)^2$ becomes vanishingly small. We are left with the result that $\frac{\mathrm{d}y}{\mathrm{d}x} = 2x$ for $y = x^2$.

A general result that can shown to be to true for polynomials is that for $y = x^n$, $\frac{dy}{dx} = nx^{n-1}$.

## *Random Variables*

A variable that is said to be random is one that is subject to some degree of nondeterminism. For example, the draw of a card from a deck, the role of a die, or the arrival time of a bus/train all have some element of randomness. In this section, we will briefly review some important properties of random variables and present some important results on how to work with them.

Let us consider first a simple case of rolling a die. There are six distinct outcomes, and in the case of an unbiased die, these all have an equal chance of occurring. We can write down the probability of each outcome very easily, provided that we follow some simple rules.

1. A probability of 1 for an outcomes means that outcome is certain. For examples, if all faces of the die had three dots, $P(3) = 1$.
2. One of the possible outcomes *must* occur, and no other outcome can occur, so the probabilities must add to 1.
3. No outcome can have a probability of greater than one (implies that an outcome occurs more frequently than every time!)
4. No outcome can have a negative probability.

The probability matrix for the role of a single die, following these rules, is shown in Table 1. This is an example of a *discrete* probability distribution.

| X | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| P(X) | $\frac{1}{6}$ | $\frac{1}{6}$ | $\frac{1}{6}$ | $\frac{1}{6}$ | $\frac{1}{6}$ | $\frac{1}{6}$ |

Table 1: The probability matrix of the outcomes of a single role of a die.

Now consider how this matrix would change if we had a biased die, for which the probability of rolling a 6 is twice that of rolling any of the other numbers. We may be tempted to change $P(X = 6)$ to $\frac{2}{6}$ in Table 1, but this would not be correct: it would mean that the sum of the probabilities of the possible outcomes would be $\frac{7}{6}$, which violates rule 2 above. We need to *normalise* the outcomes such that this rule is obeyed. We do this by rescaling all of the probabilities by the sum of the outcomes, that is

$$P(X = X_i) \rightarrow \frac{P(X = X_i)}{\sum_i P(X = X_i)} \tag{34}$$

In the case of our biased die, the relative probabilities of the size outcomes are (in order): $\{1, 1, 1, 1, 1, 2\}$ and so $\sum_i P(X = X_i) = 7$, and the probabilities becomes $\left\{\frac{1}{7}, \frac{1}{7}, \frac{1}{7}, \frac{1}{7}, \frac{1}{7}, \frac{2}{7}\right\}$.

Now consider the case of rolling *two* die. Each die has six possible outcomes, and so there are 36 possible outcomes of rolling two dice. Again, following our rules, these are shown in Table 2.

| $X_2$ \ $X_1$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ |
| 2 | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ |
| 3 | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ |
| 4 | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ |
| 5 | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ |
| 6 | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ | $\frac{1}{36}$ |

Table 2: The joint distribution of the outcomes of rolling two dice.

This is known as the *joint* probability of $X_1$ and $X_2$. Because $X_1$ and $X_2$ are *independent* random variables, the joint distribution of the outcomes is formed from the product of the probabilities of the independent variables, that is, $P(X_1, X_2) = P(X_1)P(X_2)$ or in general:

$$P(X_1, X_2, \ldots, X_N) = \prod_{i=1}^{N} P(X_i) \tag{35}$$

From the joint probability, we can compute the probability of any possible outcome. For example, $P(X_1 + X_2)$ is obtained summing the relevant cells in Table 2:

| $X_1 + X_2$ | 2 | 3 | 4 | 5 | 6 | 7 | 6 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $P(X_1 + X_2)$ | $\frac{1}{36}$ | $\frac{2}{36}$ | $\frac{3}{36}$ | $\frac{4}{36}$ | $\frac{5}{36}$ | $\frac{6}{36}$ | $\frac{5}{36}$ | $\frac{4}{36}$ | $\frac{3}{36}$ | $\frac{2}{36}$ | $\frac{1}{36}$ |

Table 3: The probability of the sum of two rolls of a die.

For our biased die, we can form a similar table (Table 4). This is a little more interesting because it has some heterogeneity in it, but the two variables are still independent.

| $X_2$ \ $X_1$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{2}{49}$ |
| 2 | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{2}{49}$ |
| 3 | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{2}{49}$ |
| 4 | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{2}{49}$ |
| 5 | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{1}{49}$ | $\frac{2}{49}$ |
| 6 | $\frac{2}{49}$ | $\frac{2}{49}$ | $\frac{2}{49}$ | $\frac{2}{49}$ | $\frac{2}{49}$ | $\frac{4}{49}$ |

Table 4: The joint distribution of the outcomes of two rolls of a biased die.

Let us consider a different example with more illustrative power. The variables are again independent, but we can ask richer questions. Our chosen example is a diagnostic medical test, in which the outcome of the test depends on the whether a patient has a disease. The test has the following properties:

- If the patient has the disease, the probability of a positive test is 0.9 (a *true positive*).
- If the patient does not have the disease, the probability of a positive test is 0.05 (*false positive*).

These are *conditional* probabilities, which we denote $P(X|Y)$: the value of $X$ (the test) depends on the value of $Y$ (the disease state). We use $X$ to denote the test, and $Y$ to denote the didease state, and noting that both can be either true ($T$) or false ($F$) we have

$$\begin{aligned} P(X = T|Y = T) &= 0.9 && (36) \\ P(X = T|Y = F) &= 0.05 && (37) \end{aligned}$$

Given these, we can also deduce that because a subject with the disease must give rise to either a true or false test, then

$$\begin{aligned} P(X = F|Y = T) &= 0.1 && (38) \\ P(X = F|Y = F) &= 0.95 && (39) \end{aligned}$$

We also know that the disease is quite rare, with only 0.1 of the population suffering from it, so $P(Y = T) = 0.1$ and $P(Y = F) = 0.9$. From this information we can form the joint distribution by noting that (for example) $P(X = T, Y = T)$ is formed by weighting the conditional probability $P(X = T|Y = T)$ by the probability that $Y = T$. That is, $P(X = T, Y = T) = P(X = T|Y = T)P(Y = T)$. The full joint probability distribution is shown in Table 5

| $Y$ \ $X$ | T | F |
|---|---|---|
| T | 0.09 | 0.01 |
| F | 0.045 | 0.855 |

Table 5: The joint distribution of the outcome of a medical test $X$ and the prevalence of the disease $Y$.

In building this table we have made explicit use of two fundamental rules of probability: the **sum rule**

$$P(X) = \sum_{\{Y\}} P(X, Y) \tag{40}$$

(where $\{Y\}$ is the set of all possible values $Y$ can take); and the product rule

$$P(X,Y) = P(X|Y)P(Y). \tag{41}$$

We can ask some interesting questions of this data. First, we verify what we already know. Using the sum rule, we add across the rows to compute the probabilities of disease/not disease: $P(Y = T) = 0.09 + 0.01 = 0.1$ and $P(Y = F) = 0.045 + 0.855 = 0.9$. Good, this is what we started with so we have not made a mistake. In a similar vein we can compute the probability of positive/negative tests: $P(X = T) = 0.09 + 0.045 = 0.135$ and $P(X = F) = 0.01 + 0.855 = 0.865$. But we can ask even more interesting and pertinent questions, for example: if the test is positive, what is the probability that the patient has the disease (i.e. $P(Y = T|X = T)$)? On the face of it, our test is very good: it has a high false positive and low false positive rate, but is this borne out in practice?

To answer this question, we make an observation about conditional probabilities. We used the relation $P(X,Y) = P(X|Y)P(Y)$, but since $P(X,Y) = P(Y,X$ we also have $P(X,Y) = P(Y|X)P(X)$. It follows that $P(X|Y)P(Y) = P(Y|X)P(X)$, and that

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \tag{42}$$

This is known as *Bayes' Rule*, and it is a very powerful tool which which to reason about probabilities. A loosely intuitive explanation of Bayes' rule is that is our knowledge of $Y$ following (*posterior* to) a measurement of $X$, $P(Y|X)$, depends on the *likelihood* $P(X|Y)$ of measuring $X$ given $Y$, given *prior* knowledge of $X$, $P(X)$. $P(X)$ is a normalising factor.

Let us formulate our question in these terms. We want to compute $P(Y = T|X = T)$ which, from Bayes' rule, is $P(Y = T|X = T) = P(X = T|Y = T)P(Y = T)/P(X = T)$. We know all of these numbers: $P(X = T|Y = T) = 0.9$, $P(Y = T) = 0.1$, and $P(X = T) = 0.135$. We therefore have that $P(Y = T|X = T) = 0.9 \times 0.1/0.135 = 0.67$: only 2 out of 3 positive tests is from a patient who has the disease. Notice that we could also have derived this directly from the joint distribution: $P(Y = T|X = T) = P(X = T, Y = T)/P(X = T)$.

Despite the fact that the test is very good, the outcomes are surprisingly unreliable. This is a consequence of the rarity of the disease: although the false positive rate is very low (5%), the proportion of patients who could give rise to this (90%) is very large. This demonstrates the importance of taking into account prior information when modelling an inference or learning problem. Inference via Bayesian means will be an important tool in our studies of machine learning.

Our discussions to the point have been restricted to variables that take discrete values, but everything was have discussed can be transferred quite straightforwardly to the case of continuous variables, subject to a few technical modifications. Whilst for discrete variables we have worked with tables of probabilities, for continuous variables we will work with **probability density functions** (PDF) which describes how the possible values of the variable are distributed. These are entirely analogous to the tables of probabilities that we have considered in the discrete case with one important difference: in the discrete case, we model the probability that the random variable takes a certain value; in the continuous case we cannot do this, because there are infinitly many values and by definition the probability that the variable will take a precise value is zero. However, the probability that the variable will lie within some range of value is finite, and is given by the area under the PDF in the range of interest.

All PDFs obey some important properties that are almost the same as discrete probability distributions, but with some important differences. A PDF $P(x)$ must have the following properties:

1. $P(x) \geq 0$ for all values of $x$
2. The area under $P(x)$ must be equal to one.
3. The $P(a \leq x \leq b)$ is given by the area under the PDF in that range of $x$ values

The main differences are that because continuous variable can take infinitely many values, by definition the probability that the variable will take a precise value is zero and it is only meaningful to talk about ranges of values (point 3 above). A second difference is that the PDF can take values of greater than one provided that the total area under the PDF is exactly equal to one.

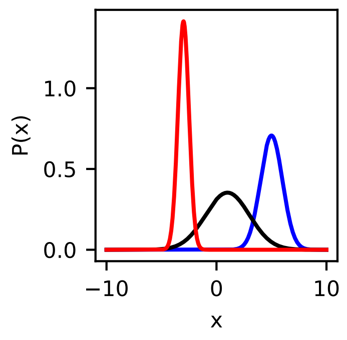

Figure 2: Three examples of normal distributions. Red: $\mu = -3, \sigma = 0.5$; Blue: $\mu = 5, \sigma = 1$; Black: $\mu = 1, \sigma = 2$.

Let us consider an example to illustrate these ideas. The *Gaussian*, or *normal* distribution describes many common physical and statistical processes, especially those involving large numbers of independent random variables. In particular, the average values of random variables drawn from independent distributions can be shown to be normally distributed. Physical measurements, which are frequently the result of a large number of independent random processes are very often normally distributed. The normal distribution is also commonly used as a surrogate distribution for variables where the true distribution is unknown. For a single random variable, the normal distribution is given by

$$p(x|\mu, \sigma) = \mathcal{N}(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{43}$$

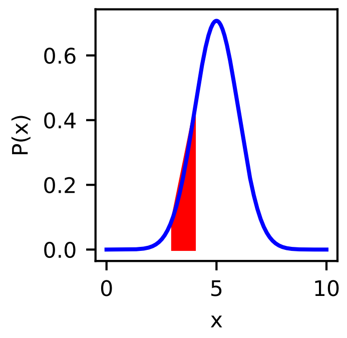

Figure 3: A example of a probability density function, in this case a normal distribution (Gaussian) $\mathcal{N}(x|5,1)$, ie with mean $\mu = 5$ and standard deviation $\sigma = 1$. The area of the red shaded region is the probability that $x$ has a value between 3 and 4.

Notice that we have written this as a conditional probability: it is conditional on the parameters $\mu$ and $\sigma$ which control the centre and width of the distribution respectively. Three examples of Gaussian PDFs for different combinations of $\mu$ and $\sigma$ are shown in Figure 2. Notice that the PDF can take values of greater than one provided the total area underneath is exactly one.

As with all PDFs, the normal distribution describes the probability that a value drawn from it will lie in a certain range. Concretely, the area under any portion of this curve bounded by two values of $x$ is the probability that a values drawn from that distribution will like between those two values. This is illustrated in Figure 3, in which the red shaded area represents $P(3 \leq x \leq 4)$.

The normal distribution can also be generalised to the multivariate case to represent a joint distribution. For two independent variables $x$ and $y$, it has the form

$$\mathcal{N}(x|\mu_x, \mu_y, \sigma_x, \sigma_y) = \frac{1}{2\pi(\sigma_x \sigma_y)} \exp\left( - \left( \frac{(x-\mu_x)^2 +}{2\sigma_x^2} + \frac{(y-\mu_y)^2 +}{2\sigma_y^2} \right) \right) \tag{44}$$

and this is shown in Figure 4. In higher dimensions, the general form for dependent variables is

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^N |\boldsymbol{\Sigma}|}} \exp\left( - \frac{(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{x}-\boldsymbol{\mu})}{2} \right) \tag{45}$$

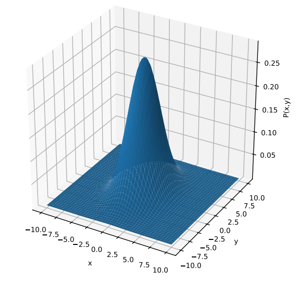

Figure 4: A example of a 2d normal PDF, with $\mu_x = \mu + y = 0$, $\sigma_x = 2$ and $\sigma_y = 3$.

where $\boldsymbol{\mu}$ is a vector of the mean values of each variable, $\boldsymbol{\Sigma}$ is the *covariance* matrix and $|\boldsymbol{\Sigma}|$ is its *determinant*. We will study this in more detail when the need arises.

Performing detailed calculations with PDFs generally requires an extensive knowledge of integral calculus to compute the desired areas under the PDF. This is beyond the scope of what we can cover in-depth during this module. For the moment, the key is to recognise the following points:

- Integration can be very loosely defined as a technical process for computing the area under a curve.
- Integration acheives this by summing a series of small strips under the curve, in the limit that those strips become vanishingly small.
- The process of integration is represented by the notation

  $$I(a,b) = \int_a^b f(x)dx \tag{46}$$

  which is a statement that we are computing the area under the curve $f(x)$ between $x = a$ and $x = b$.

You need to be able to recognise integration when you see it, and understand what is being achieved by integration. You will not be expected to calculate any integrals. It is worth noting that integration and differentiation are intimately related: they are essentially inverse processes. However there are some caveats to this. For example, differentiation irreversibly loses information about constant terms because they do not affect the gradient of a curve, whereas constant terms do affect the area under the curve, so some care is needed. Issues such as this will be explained when we meet them.

# Regression

## Introduction

One of the most easily stated problems in machine learning is the regression problem. In its absolute simplest form, regression is about fitting lines to data, and there are many ways in which this can be done (Figure 5). More generally, it is about finding the relationship between one or more *dependent* variables and one (or more) *independent* variables. The desired outcomes of a regression are typically to (a) be able to predict values of the dependent variables given values of the independent variables; (b) derive estimates of the parameters that define an underlying mathematical model.

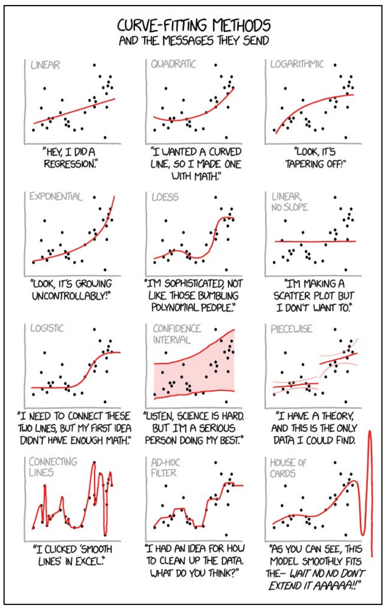
CURVE-FITTING METHODS
AND THE MESSAGES THEY SEND
LINEAR
"HEY, I DID A REGRESSION."
QUADRATIC
"I WANTED A CURVED LINE, SO I MADE ONE WITH MATH."
LOGARITHMIC
"LOOK, IT'S TAPERING OFF!"
EXPONENTIAL
"LOOK, IT'S GROWING UNCONTROLLABLY!"
LOESS
"I'M SOPHISTICATED, NOT LIKE THOSE BUMBLING POLYNOMIAL PEOPLE."
LINEAR, NO SLOPE
"I'M MAKING A SCATTER PLOT BUT I DON'T WANT TO."
LOGISTIC
"I NEED TO CONNECT THESE TWO LINES, BUT MY FIRST IDEA DIDN'T HAVE ENOUGH MATH."
CONFIDENCE INTERVAL
"LISTEN, SCIENCE IS HARD. BUT I'M A SERIOUS PERSON DOING MY BEST."
PIECEWISE
"I HAVE A THEORY, AND THIS IS THE ONLY DATA I COULD FIND.
CONNECTING LINES
"I CLICKED 'SMOOTH LINES' IN EXCEL."
AD-HOC FILTER
"I HAD AN IDEA FOR HOW TO CLEAN UP THE DATA. WHAT DO YOU THINK?"
HOUSE OF CARDS
"AS YOU CAN SEE, THIS MODEL SMOOTHLY FITS THE– WAIT NO NO DON'T EXTEND IT AAAAAA!!"

Figure 5: Some approaches to curve fitting. `https://t.co/X76aDTJXI9`

Consider the simple dataset shown in Figure 6 in which we have an independent variable $x$ upon which $y$ is dependent. In this simple example there are two questions we can ask:

1. What is the value of $y$ at $x = 2.5$ (or 3.2, or 1.9 etc)?

2. Given that it seems reasonably obvious that $y$ has a linear dependence on $x$, what are the parameters (intercept, gradient) of the underlying linear model?

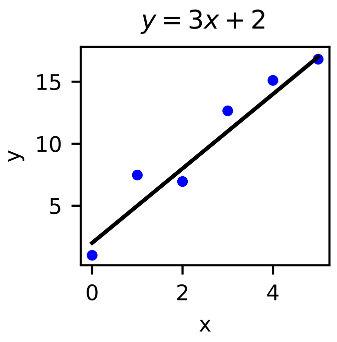

Figure 6: A simple example of a dataset with independent variable $x$ and dependent variable $y$.

How to answer these questions will be the focus of this part of the course. Regression is an important topic with a wide range of applications, but it can also be used to introduce a range of important topics of wider generality. We will signpost these along the way.

## Linear regression

In the first instance, let us consider a class of problems that are *linear*. We will define what we mean by this shortly. In the first instance, we will restrict ourselves to the simple case of data that has one independent variable $x$ and one dependent variable $y$. A dataset is then comprised of a set of $N$ pairs of data points

$$\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\} = \{(x_i, y_i)\}_{i=1}^{N} \tag{47}$$

We wish to model the relationship between $x$ and $y$ as a mathematical function $f(\mathbf{w}, x)$ such that $y_i \approx f(\mathbf{w}, x_i)$ with unknown parameters $\mathbf{w}$.

One simple way to approach regression tasks is to model regression as an optimisation problem in which the objective is to find the value of $\mathbf{w}$, which we denote $\mathbf{w}^*$ that minimises some "loss", or objective function $\mathcal{L}(\mathbf{w})$.

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}) \tag{48}$$

The intuition behind this is that $\mathcal{L}(\mathbf{w})$ should be designed to capture the difference between the data and the predictions of the model, and the optimisation will seek to minimise that difference. One very common choice for $\mathcal{L}(\mathbf{w})$ is the *least-squares error*. Given our dataset $\mathcal{D}$ and a modelling function $f(\mathbf{w}, x)$, we construct, for each datapoint in $\mathcal{D}$ a *residual error* defined as

$$r_i(\mathbf{w}) = y_i - f(\mathbf{w}, x_i). \tag{49}$$

This is illustrated in Figure 7.

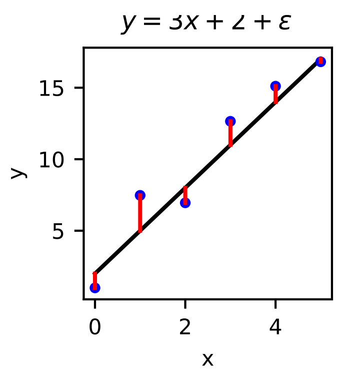

Figure 7: The residuals (shown in red) are a measure of the goodness-of-fit of a function (black line) to a set of data (blue points).

The least squares error (LSE) loss function is then defined in terms of the residuals as

$$\mathcal{L}_{\text{LSE}}(\mathbf{w}) = \sum_{i=1}^{n} r_i^2 = \mathbf{r}^{\mathrm{T}}\mathbf{r} \tag{50}$$

It is important to note here that $\mathcal{L}_{\text{LSE}}$ has no upper bound, but it does have a finite lower bound because it is a strictly positive quantity. It is therefore possible to minimise this quantity and, following Equation (48), our goal is to find $\mathbf{w}$ that minimises the loss:

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \mathcal{L}_{\text{LSE}}(\mathbf{w}) \tag{51}$$

Optimisation problems can be extremely difficult and we will, for now, restrict ourselves to a specific case which can be analysed using some relatively straightforward mathematics: models which are *linear*. What we mean by this is not the familar "$y = mx + c$" example (althought this *is* a linear problem) that you may have seen before, but a more general class of models that are *linear in their unknown parameters*. These will turn out to be surprisingly powerful. Linear models take the form

$$f(\mathbf{w}, x) = w_1\phi_1(x) + \cdots + w_M\phi_M(x) = \sum_{i=1}^{M} w_i\phi_i(x). \tag{52}$$

Our function is a *linear combination* of a set of *basis functions* $\{\phi_i(x)\}_{i=1}^{M}$ weighted by the free parameters $\{w_i\}_{i=1}^{M}$. Note that the basis functions have no free parameters and depend solely on the independent variable. The basis functions are a representation of the problem and should be chosen carefully to be sufficiently expressive to capture the features in the data. A very common choice of basis is the polynomials $\{x^i\}_{i=1}^{M}$ which we will use in the examples that follow.

For a finite set of data points $\mathcal{D}$, we can rewrite Equation (52) in matrix form by defining matrix $\mathbf{\Phi}$ with components $\Phi_{ij} = \phi_j(x_i)$ in terms of which

$$\mathbf{f}(\mathbf{w}) = \mathbf{\Phi}\mathbf{w} \tag{53}$$

where the dependency on $x$ is now absorbed into the components of $\mathbf{f}$. It is important to note the order of the indices in the definition of $\Phi_{ij}$: each row (indexed by $i$) corresponds to a single data point, whilst each column corresponds to a basis function. As an example for a simple quadratic model $f(\mathbf{w}, x) = w_1 + w_2 x + w_3 x^2$ with basis functions $\{x^0, x^1, x^2\} = \{1, x, x^2\}$, we have

$$\mathbf{\Phi} = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ & \vdots & \\ 1 & x_N & x_N^2 \end{pmatrix} \tag{54}$$

Having restricted ourselves to linear models, we can begin to solve our optimisation problem (Equation (48)). The residuals (Equation (49)) can now be written as

$$\mathbf{r} = \mathbf{y} - \mathbf{\Phi}\mathbf{w}, \tag{55}$$

and the loss function (Equation (51)) becomes

$$\mathcal{L}_{\text{LSE}}(\mathbf{w}) = (\mathbf{y} - \mathbf{\Phi}\mathbf{w})^{\text{T}} (\mathbf{y} - \mathbf{\Phi}\mathbf{w}) \tag{56}$$

We now take advantage of our observation that $\mathcal{L}_{\text{LSE}}$ has no upper bound but does have a lower bound to solve Equation (48). To minimise $\mathcal{L}_{\text{LSE}}$ we find the point at which its gradient with respect to its free parameters is zero. Since $\mathcal{L}_{\text{LSE}}$ has no maximum, this point must be the minimum. We therefore differentiate $\mathcal{L}_{\text{LSE}}$ with respect to $\mathbf{w}$ and set to zero. We get

$$\frac{\partial \mathcal{L}_{\text{LSE}}(\mathbf{w})}{\partial \mathbf{w}} = 2\mathbf{\Phi}^{\text{T}} (\mathbf{y} - \mathbf{\Phi}\mathbf{w}). \tag{57}$$

To understand how we obtain this result let us break the calculation down step-by-step. Noting that $\mathcal{L}_{\text{LSE}}(bw) = \mathbf{r}^{\text{T}}\mathbf{r}$, we first compute the derivative of $\mathbf{r}$ with respect to the components of $\mathbf{w}$. We first note that

$$r_i = y_i - \sum_j \Phi_{ij} w_j \tag{58}$$

Then, picking a particular component of $\mathbf{w}$, say, $w_k$, to differentiate with respect to, we find that

$$\frac{\partial r_i}{\partial w_k} = -\Phi_{ik}. \tag{59}$$

Now, we note $\mathcal{L}_{\text{LSE}} = \sum_i r_i^2$ and so

$$\frac{\mathcal{L}_{\text{LSE}}}{\partial r_l} = 2r_l \tag{60}$$

Then, we apply the chain rule of differentiation:

$$\frac{\partial \mathcal{L}_{\text{LSE}}}{\partial w_k} = \sum_l \frac{\mathcal{L}_{\text{LSE}}}{\partial r_l} \times \frac{\partial r_l}{\partial w_k} \tag{61}$$

$$= \sum_l 2 r_l \Phi_{lk} \tag{62}$$

Now we note that since **r** is a *column* vector, we have to do some rearrangements to rewrite this in matrix notation. This means we have to make sure that i) **r** is the last term in the equation, and ii) that the matrix **Φ** is in the correct orientation for the multiplication. Writing the result as a vector $\frac{\partial \mathcal{L}_{\text{LSE}}}{\partial \mathbf{w}}$ with components $\frac{\partial \mathcal{L}_{\text{LSE}}}{\partial w_k}$, we have:

$$\frac{\partial \mathcal{L}_{\text{LSE}}}{\partial w_k} = \sum_l -2 r_l \Phi_{lk} = -2 \sum_l \Phi^{\text{T}}_{kl} r_l \quad \rightarrow \quad \frac{\partial \mathcal{L}_{\text{LSE}}}{\partial \mathbf{w}} = -2\mathbf{\Phi}^{\text{T}}\mathbf{r} = -2\mathbf{\Phi}^{\text{T}}\left(\mathbf{y} - \mathbf{\Phi}\mathbf{w}\right). \tag{63}$$

Finally, we set the result to zero to obtain

$$\mathbf{\Phi}^{\text{T}}\mathbf{y} - \mathbf{\Phi}^{\text{T}}\mathbf{\Phi}\mathbf{w}^* = 0 \tag{64}$$

This result is known as the **normal equation** and is a set of simultaneous linear equations that we can solve for $\mathbf{w}^*$. A naïve way to do this is to evaluate $\mathbf{w}^* = (\mathbf{\Phi}^{\text{T}}\mathbf{\Phi})^{-1}\mathbf{\Phi}^{\text{T}}\mathbf{y}$, but numerical inversion of matrices can be troublesome, especially if the matrix is large (in this case, large $M$) and this is best avoided. It is therefore usual to solve the normal equation directly (eg using Gaussian elimination).

What we have developed here is a set of mathematical procedures by which the parameters of some model can be *learned from data*. This is the very core of what machine learning is about. Although we have constrained the form of the model, it is from the data that we learn what its precise form is. Let us see how it works in practice. We will initially consider a very simple example by attempting to model a simple sinusoidal function using a polynomial basis. That is, we seek to find coefficients **w** such that

$$\sin(2\pi x) = \sum_{i=0}^{M} w_i x^i. \tag{65}$$

Should we expect to be able to do this? Yes – it is a well known mathematical result that $\sin(ax) = ax - \frac{a^3x^3}{3!} + \frac{a^5x^5}{5!} - \frac{a^7x^7}{7!} + \cdots$, and the coefficients of this for $a = 2\pi$ are given in Table 6. We will perform two experiments. First, we will attempt to fit a polynomial to $y = \sin(2\pi x)$. Then, we will investigate the effect of a little additive noise by fitting a polynomial to $y = \sin(2\pi x) + \epsilon$.

In Figure 8, we see the results of different orders $M$ of fit to ten points sampled uniformly in the range $x \in [0, 5]$. We see that when the degree of fit reaches $M = 3$, the fitted curve is a very good approximation to the underlying function, and adding further terms makes only a small difference. This makes sense: we expect that $w_4 = 1/4! = 1/24$ and so the contribution of the higher order terms decreases rapidly. It is instructive to plot the root-mean-square error, $R = \sqrt{\frac{1}{N}\sum_i r_i^2}$, which is shown in Figure 9. From this curve, we see that the error converges rapidly towards zero, with little change after $M = 7$, which visually looks to be a good model of the data. Notice the plateaus in the RMS plot which reflect the even terms with a coefficient of zero in the Taylor series.

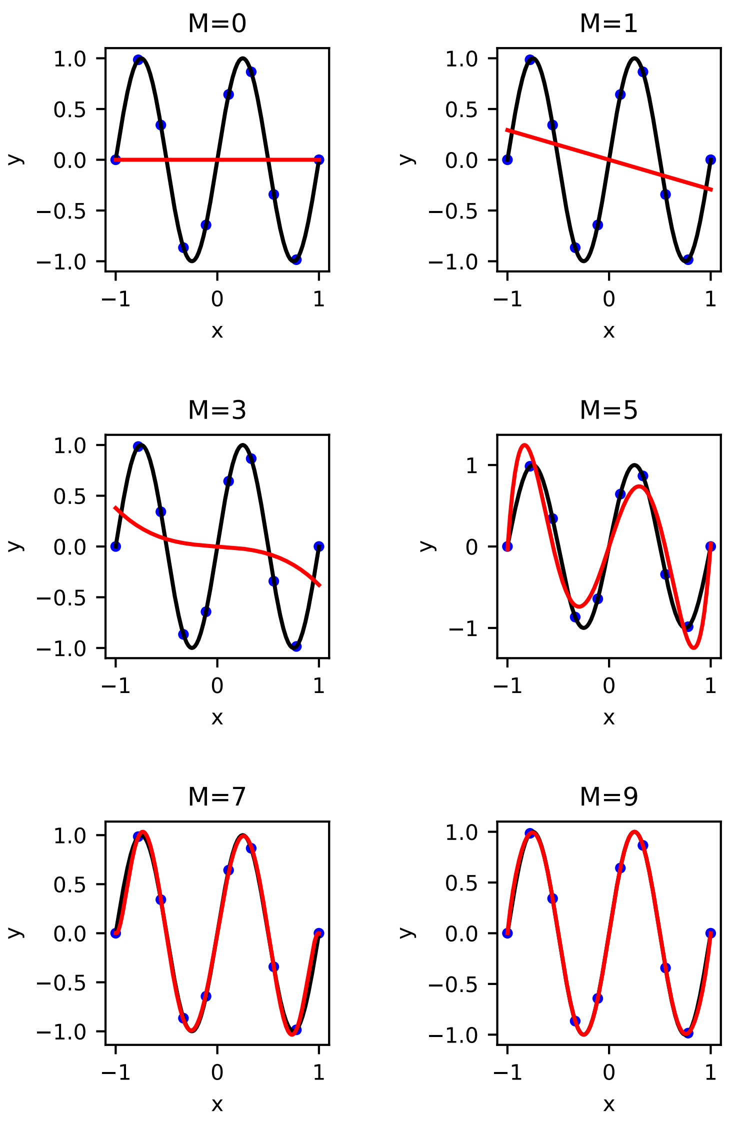

Figure 8: Fitting $y = \sin(2\pi x)$ with polynomial fits of increasing degree $M$. The blue line represents the true function; the black points are the sampled points $(x_i, y_i)$; the red line is the best-fit polynomial of that order.

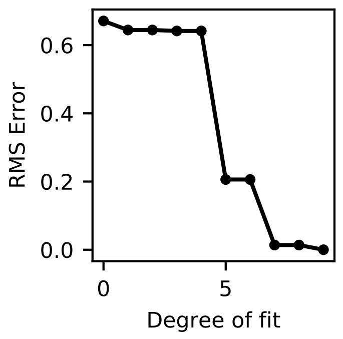

Figure 9: RMS Error of polynomial fits of different degree to $y = \sin(2\pi x)$.

It is also instructive to study the coefficients that we obtain from the fit to make sure they are consistent with the Taylor series. These are shown in Table 6. We should not expect these to match our expectations exactly: we are looking only the range $x \in [-1, 1]$ in which low-order polynomials can model the data very well. We notice that the correspondence between the terms and their true values gradually improves as we add high order terms in to the model until for $M = 9$, the difference is very small for the first few terms.

Let us now consider the effect of adding some noise to the sam-

| M | $w_0$ | $w_1$ | $w_2$ | $w_3$ | $w_4$ | $w_5$ | $w_6$ | $w_7$ | $w_8$ | $w_9$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.00 | | | | | | | | | |
| 1 | 0.00 | -0.29 | | | | | | | | |
| 2 | 0.00 | -0.29 | -0.00 | | | | | | | |
| 3 | 0.00 | -0.07 | -0.00 | -0.31 | | | | | | |
| 4 | 0.00 | -0.07 | 0.00 | -0.31 | -0.00 | | | | | |
| 5 | 0.00 | 3.85 | -0.00 | -16.51 | 0.00 | 12.69 | | | | |
| 6 | 0.00 | 3.85 | -0.00 | -16.51 | 0.00 | 12.69 | -0.00 | | | |
| 7 | -0.00 | 6.00 | 0.00 | -35.84 | -0.00 | 54.04 | 0.00 | -24.20 | | |
| 8 | -0.00 | 6.00 | 0.00 | -35.84 | -0.00 | 54.04 | 0.00 | -24.20 | -0.00 | |
| 9 | -0.00 | 6.28 | 0.00 | -41.12 | -0.00 | 78.61 | 0.00 | -63.77 | -0.00 | 20.00 |
| True | 0 | 6.28 | 0 | -41.34 | 0 | 81.61 | 0 | -76.7 | 0 | 42.1 |

Table 6: Coefficients of polynomial fits of different degree to $y = \sin(2\pi x)$.

pled data. This is much more realistic in most situations where the data has been obtained through some measurement process. We generate ten uniformly spaced data points in $x \in [0, 5]$ using $y = \sin(2\pi x) + \epsilon$ where $\epsilon \sim \mathcal{N}(0, 0.25)$. In Figure 10 we show different degrees of polynomial fit to this data.

For low order fits, the fitted curves look very similar to the noise-free case and a quintic fit gives a reasonable result. However, at $M = 9$ something odd happens. The fit appears to get much worse with large divergences from the ground truth. Yet the RMS error, shown in Figure 11, shows the error continuing to decrease. What is going on here?

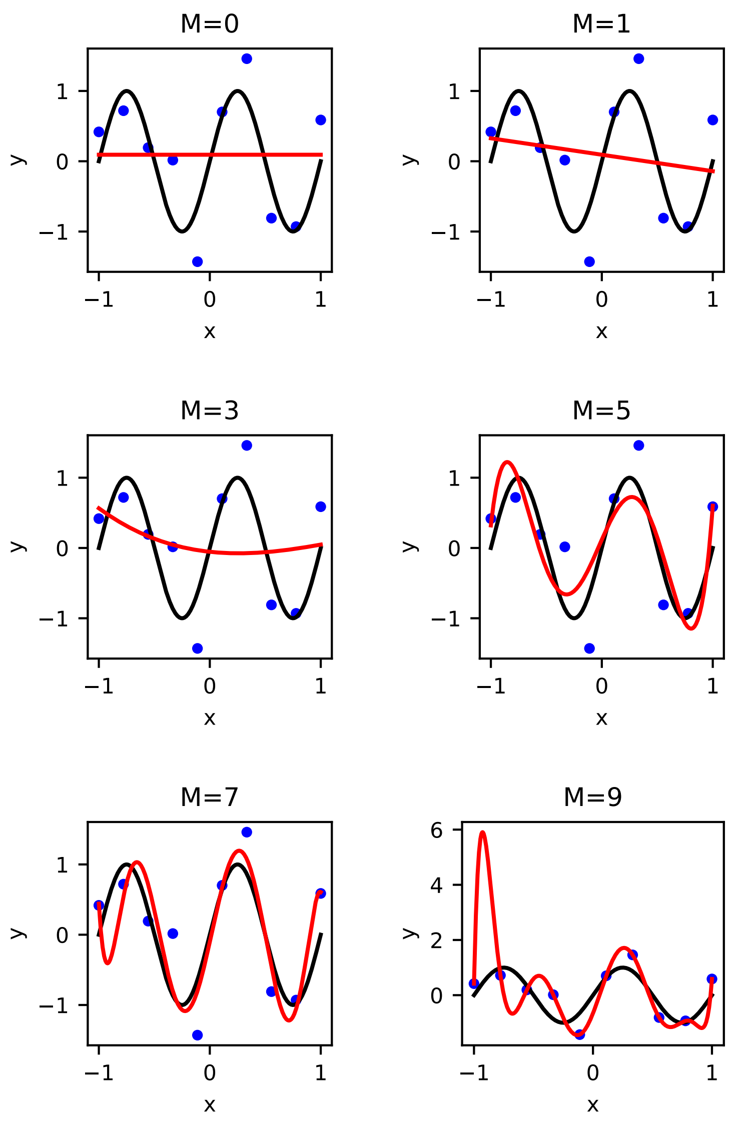

Figure 10: Fitting $y = \sin(2\pi x)$ with polynomial fits of increasing degree $M$. The blue line represents the true function; the black points are the sampled points $(x_i, y_i)$; the red line is the best-fit polynomial of that order.

To understand this, we should think about what we are trying to achieve in quite simple terms. We have $N$ data points from which we are trying to derive the values of $M$ free parameters. In the case $M < N$ we have fewer free parameters than we have data points, and the solution that minimises the least-squares loss function find the free parameters that minimise the total Euclidean distance of the fitted line from the data points, but *does not necessarily pass through any of them*, as we see for small values of $M$. The case $M = 9$, however, has ten unknowns, and ten free parameters. It is therefore possible (assuming the basis is suitably expressive) to choose these parameters such the function passes exactly through all of the data points. In the noise-free case, we saw that this didn't matter – the data points all lie on the curve $y = \sin(2\pi x)$, but when we add a little noise to the data, we introduce high-frequency fluctuations into the data that cannot be represented using low-order fits, but can be matched exactly by high-order fits. At $M = 9$ we have ten free parameters and so we can construct a curve that exactly passes through all of the data points. The price of this "perfect fit" is that the curve deviates wildly from the ground truth between the sampled points. In order to exactly fit the high-frequency noise components, higher order terms tend to have large amplitudes (Table 7) so that the high order polynomial terms can generate the high-frequency fluctuations needed, but cancel each other out at the data points so that the fit is exact. This leads to the large deviations we see between the sampled points. This is know as *overfitting* and is a very common problem when working with noisy datasets and complex models. A model that overfits its training data tends to be able to generalise to unseen data. For examples, consider what value of $y$ our $M = 9$ model would predict for $x = -0.5$.

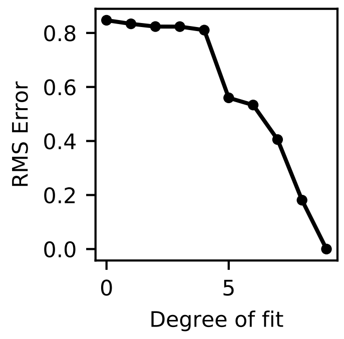

Figure 11: RMS Error of polynomial fits of different degree to $y = \sin(2\pi x)$.

| M | $w_0$ | $w_1$ | $w_2$ | $w_3$ | $w_4$ | $w_5$ | $w_6$ | $w_7$ | $w_8$ | $w_9$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | -0.66 | 10.98 | 25.62 | -117.80 | -143.29 | 405.10 | 246.74 | -561.32 | -127.01 | 263.129 |

Table 7: Coefficients of a high-order polynomial fit to noisy data show characteristic large values of high-order coefficients.

Before we investigate technical ways to overcome overfitting, it is worth noting that one very simple way to stop this is to acquire more data! In Figure 12 we show a polynomial fit of degree $M = 9$ to a set of $N = 30$ data points. The fluctuations that were apparent with ten data points have vanished and the fit is smooth. This highlights a very important issue that motivates what we will discuss in the next section. In both cases, the data points were produced by the same underlying process. In both cases, the model we fitted was of the same complexity and expressiveness. Yet we obtained very different outcomes. It seems rather unsatisfactory that simply changing the density of the data sampling should affect the result of the learning process. We next consider some ways in which this can be controlled.

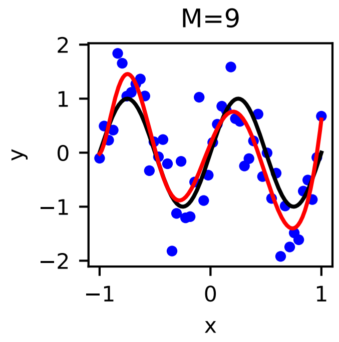

Figure 12: Polynomial fit of 30 data points with $M = 9$ polynomial.

## *Regularisation*

Our approach to learning a function from data has been based around trying to minimise the loss, or error, with which the model is able to describe the data. As we have seen, it is inevitable that a model of sufficient complexity will always be able to perfectly match the data points supplied. This seems like it should be highly desirable, but the consequence is that the model does not generalise to data that was not in the training set used to learn the unknown parameters: the model does not learn the underlying trend in noisy data, but rather uses high-order terms in the basis set to fit both the trend of the data and the noise.

A popular way of preventing models from overfitting is to incorporate *prior* information about the nature of the solution in a process known as *regularisation*. This can be achieved by adding a term to the loss function that penalises solutions that do not match our prior belief of what solutions should look like. In general, the loss function for regularised problems is written

$$\mathcal{L}(\mathbf{w}) = \mathcal{L}_{\text{err}}(\mathbf{w}) + \lambda R(\mathbf{w}) \tag{66}$$

where $\mathcal{L}_{\text{err}}$ is a measure of the mismatch between the model prediction and the data (such as the least-squares error), and $R(\mathbf{w})$ is a function of the parameter vector that imposes some penalty on solutions that are undesireable by increasing the value of the loss function for these solutions.

Let us think about how this might be used to prevent overfitting. We saw in Table 7 that one of the characteristics of overfitted solutions is the presence of very large coefficients that enable higher order terms to fit the noise in the data. We might therefore propose that somehow preventing the size of the coefficients from getting too large would prevent this. One way in which we might acheive this is by imposing a penalty when the magnitude of the parameter vector gets large, ie. we penalise large values of

$$R(\mathbf{w}) = \sum_i w_i^2 = \mathbf{w}^{\mathrm{T}}\mathbf{w}. \tag{67}$$

With this choice, the loss function, with least-squares error, is written

$$\mathcal{L}(\mathbf{w}) = (\mathbf{y} - \mathbf{\Phi}\mathbf{w})^{\mathrm{T}} (\mathbf{y} - \mathbf{\Phi}\mathbf{w}) + \lambda \mathbf{w}^{\mathrm{T}}\mathbf{w} \tag{68}$$

To minimise this requires that both the error term and the magnitude of the weights are simultaneously minimised, with the free parameter $\lambda$ controlling the relative extent to which each of these terms contributes to the overall loss. Small values of $\lambda$ mean that reducing the error is preferable to reducing the weight magnitude, which large values of $\lambda$ favour reducing the weights over reducing the error.

We can proceed with the analysis as we did previously: we differentiate with respect to the weights and set this to zero to find the minimum, noting that the new term is bounded from below by zero, and has no upper bound.

$$\frac{\partial \mathcal{L}_{\mathrm{LSE}}(\mathbf{w})}{\partial \mathbf{w}} = -2\mathbf{\Phi}^{\mathrm{T}}\left(\mathbf{y} - \mathbf{\Phi}\mathbf{w}\right) + 2\lambda\mathbf{w}, \tag{69}$$

which we set to zero to obtain

$$\mathbf{\Phi}^{\mathrm{T}}\mathbf{y} - \left(\mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi} - \lambda\mathbf{I}\right)\mathbf{w}^{*} = 0 \tag{70}$$

where we have employed the identity matrix $\mathbf{I}$ to enable the expression to be factorised in this form. This method is referred to in the literature by a number of different names:

- *Ridge regression*.
- *$L_2$ regularisation*, because $\sum_i w_i^2$ is the $L_2$ norm of $\mathbf{w}$, written as $||\mathbf{w}||_2^2$.
- *Weight decay*, because it pushes weights towards zero.
- *Tikhonov regularisation*, of which it is a special case. Tikhonov regularisation uses $R(\mathbf{p}) = ||\mathbf{\Gamma}\mathbf{w}||$. Here, we have $\mathbf{\Gamma} = \lambda\mathbf{I}$.

Let us investigate how this works in practice. In Figure 13 we show the effect of different values of $\lambda$ on the learning of the model. The fluctuations in the model curve are "smoothed" out by the regularisation term. This comes at the cost of a higher error (Figure 14), because the fit no longer exactly matches the data, but it does model the trend in the data more effectively.

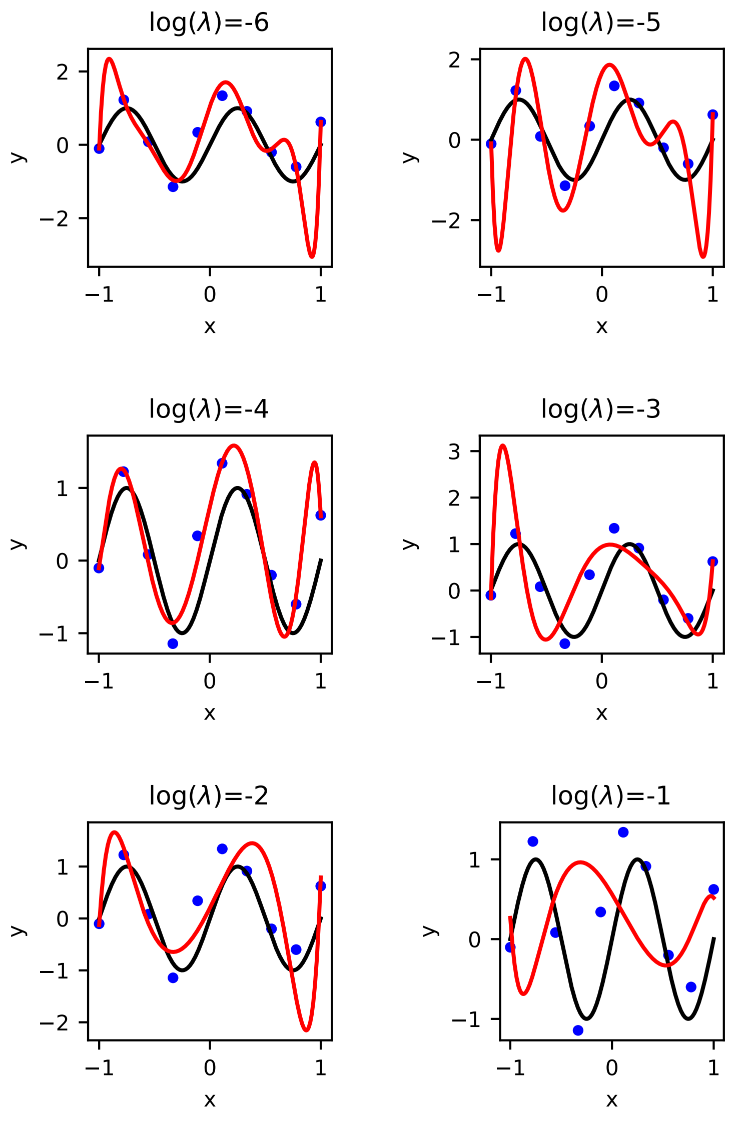

Figure 13: Fitting $y = \sin(2\pi x)$ with polynomial fits of degree $M = 9$ with $L_2$ regularisation using different regularisation coefficients $\lambda$. The blue line represents the true function; the black points are the sampled points $(x_i, y_i)$; the red line is the best-fit polynomial of that order.

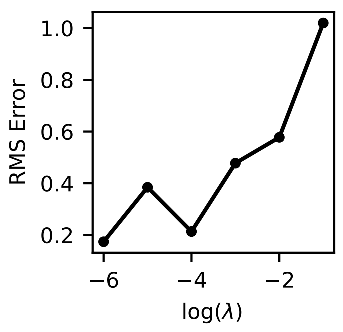

Figure 14: RMS Training Error of polynomial fit with $M = 9$ to $y = \sin(2\pi x)$ for different degrees of regularisation.

## *Selecting and Evaluating Models*

So far, we have seen how, given a set of data and a model (in our case, a linear sum of polynomials), we can compute the model coefficients such that the model matches the data. We have also looked at how to use regularisation to control the model's parameters. However, we not looked at how to formally compare and evaluate different models.

When we perform a regression task there are typically two cases of particular interest:

1. The underlying model is known and we wish to find its parameters.

2. The underlying model is unknown and we wish to learn to make predictions on unseen data.

An example of the first situation is the trajectory of a moving object that is known to be moving according to the well-known equation $s = s_0 + ut + \frac{1}{2}at^2$. If we acquire a dataset $\mathcal{D} = \{(t_i, s_i)\}_{i=1}^N$ of measurements of position $s$ at time $t$ and then fit a quadratic model $y(t) = w_0 + w_1 t + w_2 t^2$ then we can deduce the initial position $s_0 = w_0$, the initial velocity $u = w_1$, and the acceleration $a = w_2$. In this case we use our prior knowledge of the problem to select the appropriate model. Notice that in this case we do not risk overfitting because the model is chosen to model the underlying trend and does not have the capacity to model noise in the measurements.

The second situation is much more interesting and difficult. When we have no first-principles way of determining what the model should be, we have to turn to empirical means, by which we mean "experiments".

In the small experiments we have done so far, we have treated our dataset as a single object and looked to see how well out model can describe it. We have also argued, on intuitive grounds, that when the model is too complex, it can overfit the data. We will now investigate this claim. There are two situation in which we will be interested:

- Cases where we have a large quantity of data
- Cases where we have a small quantity of data

### *The Large Data Regime: Train–Validate–Test Split*

Consider a situation in which we have a large dataset $\mathcal{D}$ that is far larger than is needed to estimate the underlying model. The basic idea behind a *holdout* scheme is that we use some of the data to train the model, some of the data to evaluate whether that model is any good, and some of the data to test what we think is the best model. Let us define a partitioning of the data as follows:

- A training set $\mathcal{T}$, randomly sampled from $\mathcal{D}$.

- An validation set $\mathcal{V}$, randomly sampled from $\mathcal{D} - \mathcal{T}$.
- A test, or evaluation set $\mathcal{E} = \mathcal{D} - \mathcal{T} - \mathcal{V}$.

Our starting position is that we have a set of models $\{\mathcal{M}_i\}_{k=1}^{K}$ that we wish to evaluate (using a loss function $\mathcal{L}$) to determine which has the most predictive ability. We need to distinguish between two distinct aspects of our models: their learnable parameters $\mathbf{w}$, and their *hyperparameters*, which are those features of the model that we do not learn directly from the data: the choice of basis, the regularisation scheme etc. The basic idea is then that we use the training set to learn the model parameters, and then we use the validation set to select the choice of hyperparameters that best allows the model learned on $\mathcal{T}$ to generalise to $\mathcal{V}$, in other words, for *model selection*. Finally, we perform an evaluation of the best model on $\mathcal{E}$ to assess how well the model performs on unseen data, which is the ultimate test of its ability to generalise, and allows us to guard against overfitting of the hyperparameters to the validation set. This process is described by Algorithm 1.

```
Data: Set of models $\{\mathcal{M}_i\}_i$
Data: Dataset $\mathcal{D}$ split into training ($\mathcal{T}$), validation ($\mathcal{V}$), and
      test/evaluation ($\mathcal{E}$) sets.
Result: Identification of model $\mathcal{M}*$ with best predictive power.
for each model $\mathcal{M}_i$ do
    Train $\mathcal{M}_i$ on $\mathcal{T}$;
    Compute model loss $\mathcal{L}_{\mathcal{T}}$ on training set $\mathcal{T}$;
    Compute model loss $\mathcal{L}_{\mathcal{V}}$ on evaluation set $\mathcal{V}$;
end
Select model $\mathcal{M}*$ with best overall performance on training and
validation sets.;
Compute loss on test set $\mathcal{E}$ to determine final model
performance;
```

**Algorithm 1**: Model Selection and Evalulation using a Train–Evaluate–Test Procedure.

There are some important features of this scheme that are worth some consideration. Firstly, the dataset $\mathcal{D}$ must be large enough to enable the split to be performed whilst ensuring that the training set $\mathcal{T}$ is large enough to enable the model to learn its structure. Secondly, the dataset partitioning must be carefully performed to make sure that each of the sub-groups ($\mathcal{T}$, $\mathcal{V}$, $\mathcal{E}$) is representative of the full dataset. For example, in our toy problem of data generated by $y = \sin(2\pi x) + \epsilon$, the partioning needs to be done such that each subset contains sample from across the data's domain $x$. This is particularly important to bear in mind when working with data that has a natural order to it. Similarly, we would also need to be careful to make sure that the full range of $y$ values is represented and the data subsets are not just sampling (for example) the positive values of $y$. This is very difficult to do when working with small datasets and so an alternative schemes is necessary.

Let us implement this on our example. We generate twenty data points and split them into a training set $\mathcal{T}$ of ten points, a validation set $\mathcal{V}$ of five points, and a test set $\mathcal{E}$ of 5 points. Although these are quite small sets, for this simple example, this is sufficient.

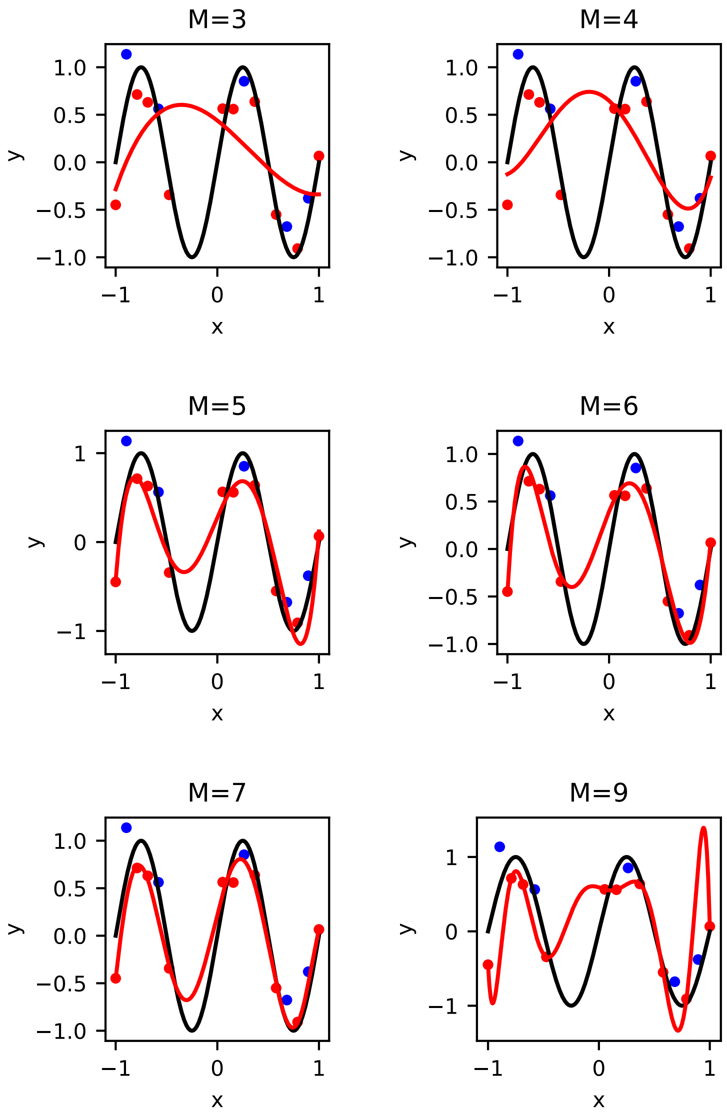

Figure 15: Evaluation of polynomial regression to $y = \sin(2\pi x)$ using a train-validate-test split of (10, 5, 5) samples. Training data is shown as red dots, validation data as blue dots. The true trend is shown as a solid black line. The polynomial fit is shown as a red line. Bottom-right: the training (black) and testing (blue) RMS errors.

Visually, we see from the fits, that, as we saw before, the low-order polynomials are not able to explain the data very well. At degree $M = 5$ we see that this changes visually. With reference to the error shown in Figure 16, we see that the training error decreases significant at this point, and the validation error also shows a significant decrease. It is interesting to see that the validation error tracks the training error quite closely for low-degree polynomials. This is to be expected of a model where the training and validation datasets are representative of the full dataset. If the validation error does not track the training error, this is an indication that at least one of the two sets is not representative of the full dataset.

At very high order fits, (remembering that we are training on ten points), we see that whilst the training error continues to improve, the validation error suddenly gets dramatically worse. This is the characteristic sign of overfitting: the model is able to fit the training data very well, but has learned both the trend in the data and the noise distribution of those datapoints: it has effectively memorised the training data and is unable to generalise its results to the validation set. On this data, fits of order $M = 5$ or $M = 6$ seem like they ought to be optimal, and a useful maxim to apply here is *Occam's Razor* which roughly states that one should choose the simplest hypothesis (model) that is supported by the data, i.e we should choose $M = 5$. On the test set, this gives an RMS error of 0.68 which is comparable with the errors on the validation set and suggests that we have learned the underlying trend in the data reasonably well.

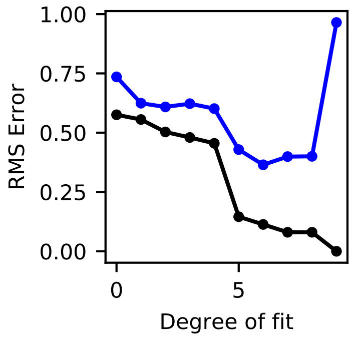

Figure 16: Training and validation errors as a function of polynomial degree for the data shown in Figure 15.

For the data we have considered here, a full training-validation-testing split seems to work, and is consistent with our intuitions. However, this is a very small dataset and an alternative technique – cross-validation – would be more appropriate.

## *The Small Data Regime: Cross-Validation*

If we only have a small number of data points (ten, in our toy problem), it is clear that a three-way split of the training data is problematic. Any significant reduction in the number of data points used to train the model will inevitably lead to difficulties: loss of the trend in the data, overfitting, loss of generalisation etc. In this circumstance, a more data-efficient method is necessary: *cross-validation*.

```
Data: Set of models $\{\mathcal{M}_i\}_i$
Data: Dataset $\mathcal{D}$ split into cross-validation ($\mathcal{V}$), and
      test/evaluation ($\mathcal{E}$) sets.
Data: Number of folds, $K$
Result: Identification of model $\mathcal{M}*$ with best predictive power.
Divide $\mathcal{C}$ into $K$ folds $\{c_k\}_{k=1}^K$ such that $\mathcal{C} = \bigcup_{k=1}^K c_k$;
for each model $\mathcal{M}_i$ do
    for $k = 1 \to K$ do
        Train $\mathcal{M}_i$ on training set $\mathcal{C} - c_k$;
        Compute model loss $\mathcal{L}_\mathcal{T}$ on training set $\mathcal{C} - c_k$;
        Compute model loss $\mathcal{L}_\mathcal{V}$ on evaluation fold $c_k$;
    end
end
Select model $\mathcal{M}*$ with best overall performance on training and
validation sets;
Compute loss on test set $\mathcal{E}$ to determine final model
performance;
```
**Algorithm 2**: Model Selection and Evaluation using Cross-Validation.

In a cross-validation scheme, the dataset is typically divided into a test/evaluation set $\mathcal{E}$, and a cross-validation set $\mathcal{C}$. The test set serves the same role as it did before: a fraction of the data is held out in order to evaluate the best model on unseen data. The cross-validation set is used somewhat differently. It is used to both *train* and *validate* the model and therefore to both learn the model parameters and the hyperparameters; that is, both training and validation data is drawn from $\mathcal{C}$. The way that this is done is by splitting $\mathcal{C}$ into $K$ *folds*. $K-1$ of the folds are used to train the data, and then the remaining fold is used as the validation set. This process is repeated $K$ times with each fold being used in turn as the validation set. The best performing model over the $K$ repeats, typically judged on the average performance over all of the split, is selected as the final model which can then be evaluated against the test set. Common approaches to cross validation are 10-fold cross-validation (train on 90%, validate on 10%), and hold-one-out, which is equivalent to $N$-fold cross validation, for $N$ data points (train on $N-1$, validate on 1). This is a common strategy when the data is very small indeed.

## *Bayesian View of Regression*

So far, we have adopted quite an informal approach to regression: we wrote down an error function (least-squares) that made some sense from an intuitive viewpoint. We then look at how to control the solution using regularisation, also relying on informal intuitive arguments. Although these arguments seems to make logical sense, we have no formal basis for claiming that they are a correct and valid way to do this. Studying the problem from a Bayesian perspective will give us the formal rigour that we need in order to justify the choices we have made.

Our starting point will be to construct a model of the underlying data-generating process. We assume that each data point is the result of some process that has a deterministic component, and some associated sampling uncertainty.

$$y = \mathbf{f}(x, \mathbf{w}) + \epsilon \tag{71}$$

where $\epsilon \sim \mathcal{N}(0, \sigma^2)$ in which $\sigma$ is a measure of the uncertainty in the sampling. When the value of $y$ is sampled for some value of $x$, it will be drawn from a normal distribution with mean $f(x, \mathbf{w})$ and variance $\sigma^2$. Under this model, we can write the distribution of $y$ as

$$p(y|x, \mathbf{w}, \sigma^2) = \mathcal{N}(y|f(x, \mathbf{w}), \sigma^2) \tag{72}$$

that is, it is normally distributed with mean $f(x, \mathbf{w})$ and variance $\sigma^2$.

Now consider that we have a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$ which we will write as $(\mathbf{x}, \mathbf{y})$. We assume that the datapoints are sampled independently from distributions with the same variance $\sigma^2$, allowing the joint probability distributions over all data points to be written as the product of the distributions for each point:

$$p(\mathbf{y}|\mathbf{x}, \mathbf{w}, \sigma^2) = \prod_{i=1}^N \mathcal{N}(y_i|f(x_i, \mathbf{w}), \sigma^2) \tag{73}$$

This is known as the *likelihood* of $y$.

Let us now ask a question of the likelihood: what value of $\mathbf{w}$ maximises it? This is a meaningful question because the likelihood is, by construction, a proper probability distribution and is therefore bounded below by zero and above only by the requirement that it be normalised.

First, we substitute in the full form of the normal distribution $\mathcal{N}(x|\mu, \sigma^2) = (2\pi\sigma^2)^{-\frac{1}{2}} \exp(-(x-\mu)^2/(2\sigma^2))$

$$p(\mathbf{y}|\mathbf{x}, \mathbf{w}, \sigma^2) = (2\pi\sigma^2)^{-\frac{N}{2}} \prod_{i=1}^N \exp(-(y_i - f(x_i, \mathbf{w}))^2/(2\sigma^2)) \tag{74}$$

We now take the logarithm of this to get rid of the exponential terms, and use the identity $\ln \prod_i a_i = \sum_i \ln a_i$ which gives us

$$\ln p(\mathbf{y}|\mathbf{x}, \mathbf{w}, \sigma^2) = -\frac{N}{2} \ln 2\pi\sigma^2 - \frac{1}{2\sigma^2} \sum_{i=1}^N (y_i - f(x_i, \mathbf{w}))^2 \tag{75}$$

The second term on the right-hand side of this equation is the familiar least-squares error term. Noting the minus-sign, we have therefore shown that maximising the log-likelihood is equivalent to minimising the least-squares error.

This rather elegant result paves the way to more sophisticated analysis of regression problems. When we considered how to address the overfitting problem through regularisation, we did this in a rather informal way based on using our intuition to add prior information. The probabilistic view of regression permits us to add prior information in a more formal way using Bayes' rule

$$p(a|b) = p(b|a)p(a)/p(b) \tag{76}$$

where $p(a|b)$ is the posterior distribution of $a$ given $b$, $p(b|a)$ is the likelihood of $b$ given $a$ and $p(a)$ is the prior distribution of $a$. Let us consider the simple and tractable case of a normal distribution of the parameters with zero mean:

$$\begin{aligned} p(\mathbf{w}|\lambda) &\propto \prod_{i=1}^{M} \exp(-\lambda w_i^2) && (77)\\ &= \exp(-\lambda \mathbf{w}^{\mathrm{T}}\mathbf{w}) && (78)\\ & && (79) \end{aligned}$$

Using Bayes Theorem we have

$$p(\mathbf{w}|\mathbf{x},\mathbf{y},\sigma^2,\lambda) \propto p(\mathbf{y}|\mathbf{x},\mathbf{w},\sigma^2) \times p(\mathbf{w}|\lambda) \tag{80}$$

and noting that $\times \mapsto +$ under a logarithm, we find that this is maximised by the minimum of

$$\sum_{i=1}^{N} (y_i - f(x_i,\mathbf{w}))^2 + \lambda \mathbf{w}^{\mathrm{T}}\mathbf{w}. \tag{81}$$

That is, $L_2$ regularisation is equivalent to specifying a *Gaussian prior* with zero mean and variance $\sigma^2 = 1/2\lambda$ on the magnitudes of the weights.

## *Bias-Variance Decomposition*

The Bayesian treatment of regression leads to a central result that applies across the spectrum of machine learning methods. Let us ask a very simple question (using a simplified notation for convenience):

Given data $y$ generated by an underlying function $h(x) + \epsilon$, and an estimated model $f(x)$, what is the expected value (i.e. the average) of the least-squares loss, which must be a random variable because the data from which the loss is constructed is a random variable. Writing $f(x) = f$ and $h(x) = h$ for simplicity, the expected loss is given by

$$\begin{aligned} \mathbb{E}[\mathcal{L}] &= \mathbb{E}[(y-f)^2] && (82)\\ &= \mathbb{E}[y^2] + \mathbb{E}[f^2] - 2\mathbb{E}[yf] && (83) \end{aligned}$$

We will rewrite this using the definition of the variance of a random variable:

$$\text{var}[X] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2, \tag{84}$$

and a property of two independent variables

$$\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y] \tag{85}$$

The expected loss then becomes

$$\mathbb{E}[\mathcal{L}] = \text{var}[y] + (\mathbb{E}[y])^2 + \text{var}[f] + (\mathbb{E}[f])^2 - 2\mathbb{E}[y]\mathbb{E}[f] \tag{86}$$

Now we can apply what we know about the problem. Since $y = h(x) + \epsilon$, then under the assumption that $\mathbb{E}[\epsilon] = 0$ and $\text{var}[\epsilon] = \sigma^2$, then $\mathbb{E}[y] = h$ and $\text{var}[y] = \sigma^2$. The expected loss becomes

$$\begin{aligned} \mathbb{E}[\mathcal{L}] &= \sigma^2 + h^2 + \text{var}[f] + (\mathbb{E}[f])^2 - 2h\mathbb{E}[f] && (87) \\ &= \sigma^2 + \text{var}[f] + h^2 + (\mathbb{E}[f])^2 - 2h\mathbb{E}[f] && (88) \\ &= \sigma^2 + \underbrace{\text{var}[f]}_{\text{variance}} + \underbrace{(h - \mathbb{E}[f])^2}_{\text{bias}} && (89) \end{aligned}$$

How can we interpret this result? First, we notice that the data is no longer explicitly present in this expression, only a measure of its variance $\sigma^2$. All dependency on the *specific sample*, $y$, of the data has been absorbed into the other terms. In particular, the variance of $f$ is a consequence of the variance in the data: if the data has no noise, we will always learn the same model, but different samples will lead to different models. Therefore, the term var $f$ represents the degree to which the estimated model is sensitive to the choice of data.

The third term in the expression, $(h(x) - \mathbb{E}[f(x)])^2$, is the square-difference between the true underlying model $h$ and the "average" estimated model $f$. This term represents the ability of the estimated model to accurately represent the true model: it is the *bias* of the estimate. For examples, fitting a linear function $f(x) = mx * c$ to $h(x) = \sin(2\pi x)$ has a high bias, because it cannot accurately match the true underlying function.

Minimising the expected loss is achieved by choosing a model that simultaneously minimises the dependence on the data sampling, and the ability of the model to represent the data. These two factors nearly always conflict with each other. A very complex model will have a low bias, because it can represent the data very accurately, but a high variance, because a different sampling of the data will give a completely different model. A very simple model will conversely have a low variance but a high bias. The need to simultaneously minimise both terms, and the trade-offs that are necessary is at the core of modern learning theory.

## *Multivariate Regression*

We have so far considered problems consisting of a single independent variable $x$ and a single dependency variable $\mathbf{y}$. The arguments we have developed can be easily extended to allow us to deal with problems with multiple dependent and independent variables. First, let us consider the case of multiple independent variables. Here, we will have to be a bit careful with our notation because things can easily become complicated. We will denote our independent variables as $x_i$. The value of the variable at a particular data points will be denoted $x_{i,j}$ (for the $j$'th data point.). In these terms, our dataset is written as

$$\mathcal{D} = \{(x_{1,i}, x_{2,i}, \ldots, x_{K,i}, y_i)\}_{i=1}^{N} \tag{90}$$

where we have $K$ independent variables. Performing a regression over this dataset can be done in exactly the same way as we did for the case of a single independent variable – we choose a basis, and them solve the normal equations – but with one important difference: our basis matrix $\mathbf{\Phi}$ now needs to contain terms from all variables. To illustrate this, consider the simplest possible case of two independent variables, and a model that contains a constant term and linear terms in the two variables:

$$y = w_0 + w_1 x_1 + w_2 x_2 \tag{91}$$

which has basis matrix

$$\mathbf{\Phi} = \begin{pmatrix} 1 & x_{1,1} & x_{2,1} \\ 1 & x_{1,2} & x_{2,2} \\ \vdots & \vdots & \vdots \\ 1 & x_{1,N} & x_{2,N} \end{pmatrix} \tag{92}$$

A more complex quadratic model

$$y = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_2^2 + w_5 x_1 x_2 \tag{93}$$

has basis matrix

$$\mathbf{\Phi} = \begin{pmatrix} 1 & x_{1,1} & x_{2,1} & x_{1,1}^2 & x_{2,1}^2 & x_{1,1}x_{2,1} \\ 1 & x_{1,2} & x_{2,2} & x_{1,2}^2 & x_{2,2}^2 & x_{1,2}x_{2,2} \\ \vdots & \vdots & \vdots & & & \\ 1 & x_{1,N} & x_{2,N} & x_{1,N}^2 & x_{2,N}^2 & x_{1,N}x_{2,N} \end{pmatrix} \tag{94}$$

It is straightforward to generalise this to other models.

Now let us consider the case of multiple ($L$) dependent variables $y_i$ (using the same notation as we did for the dependent variables). There are two ways in which we could extend our analysis. The first is to treat each dependent variable separately and to solve multiple regression problems independently. Assuming that we use the same choice of basis $\mathbf{\Phi}$ for each dependent variable we have

$$\begin{aligned}
y_1(x_1,\dots,x_K) &= \sum_j w_{1,j}\phi_j(x_1,\dots,x_K) &\rightarrow\quad \mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi}\mathbf{w}_1 &= \mathbf{\Phi}^{\mathrm{T}}\mathbf{y}_1 \quad (95)\\
y_2(x_1,\dots,x_K) &= \sum_j w_{2,j}\phi_j(x_1,\dots,x_K) &\rightarrow\quad \mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi}\mathbf{w}_2 &= \mathbf{\Phi}^{\mathrm{T}}\mathbf{y}_2 \quad (96)\\
&\vdots & \vdots & \quad (97)\\
y_L(x_1,\dots,x_K) &= \sum_j w_{L,j}\phi_j(x_1,\dots,x_K) &\rightarrow\quad \mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi}\mathbf{w}_L &= \mathbf{\Phi}^{\mathrm{T}}\mathbf{y}_L \quad (98)\\
& & & \quad (99)
\end{aligned}$$

where $\mathbf{y}_j = \left[y_{j,1}, y_{j,2}, \dots, y_{j,N}\right]^{\mathrm{T}}$ and $\mathbf{w}_i = [w_{i,1}, w_{i,2}, \dots w_{i,M}]$.

Alternatively, we could perform a "joint" optimisation over all of the dependent variables in a single step. Combining the equations for each dependent variable we arrive at the following relationship

$$\begin{aligned}
\begin{pmatrix} y_{1,1} & y_{2,1} & \dots & y_{K,1} \\ y_{1,2} & y_{2,2} & \dots & y_{K,2} \\ \vdots & \vdots & \ddots & \vdots \\ y_{1,N} & y_{2,N} & \dots & y_{K,N} \end{pmatrix} &= \begin{pmatrix} \phi_1(x_1) & \phi_2(x_1) & \dots & \phi_M(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \dots & \phi_M(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(x_N) & \phi_2(x_N) & \dots & \phi_M(x_N) \end{pmatrix} \begin{pmatrix} w_{1,1} & w_{2,1} & \dots & w_{K,1} \\ w_{1,2} & w_{2,2} & \dots & w_{K,2} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,M} & w_{2,M} & \dots & w_{K,M} \end{pmatrix}\\
\rightarrow \mathbf{Y} &= \mathbf{\Phi}\mathbf{W}
\end{aligned}$$

where each column of $\mathbf{Y}$ corresponds to a dependent variable, and the corresponding column of $\mathbf{W}$ holds the weights for that variable. In these terms, the (unregularised) optimisation can be performed jointly over all variables by solving the normal equations

$$\mathbf{\Phi}^{\mathrm{T}}\mathbf{\Phi}\mathbf{W} = \mathbf{\Phi}^{\mathrm{T}}\mathbf{Y}. \qquad (100)$$

This simultaneously minimises the least-squared loss over all dependent variables. Of course, it also requires that all dependent variables can be modelled in terms of the same basis functions. If this is not the case then the problem should be solved as a series of independent regression tasks.

## *Non-linear Regression*

*This section is provided for information only and is not examinable content.*

The analysis to date is restricted only to models that ca be expressed in the form given in Eq. (52), but there may be many cases where this is not possible (eg. $y(x) = a \sin bx$, where we need to find $a$ and $b$). We will still aim to minimise the sum of the squares of the residuals, but we cannot use the linearity of the fitting function to help us formulate the problem in a tractable way that permits a single-step solution. Our approach is to break the problem down into a series of linear approximations, each of which *reduces* the loss rather than absolutely minimising it.

We must formulate a process that instead of solving Eq. (64) to find the value of $\mathbf{w}$ that minimises the error, finds a set of parameters that simply reduces the error. We will do this iteratively until no further reduction in error is possible.

We will proceed by examining the behaviour of the error when we change $\mathbf{w}$ by a small amount. This we do by approximating the function as a straight line over a small region using *Taylor's Theorem*:

$$\begin{aligned} f(x_i, \mathbf{w} + \mathbf{\Delta w}) &\approx f(x_i, \mathbf{w}) + \sum_{j=1}^{m} \frac{\partial f(x_i)}{\partial w_j} \Delta w_j && (101) \\ &= f(x_i, \mathbf{w}) + \sum_{j=1}^{m} J_{ij} \delta w_j && (102) \end{aligned}$$

where matrix $\mathbf{J}$ has components $J_{ij} = \frac{\partial f(x_i)}{\partial w_j}$ and the residual after the change is

$$\mathbf{r}(\mathbf{w} + \mathbf{\Delta w}) = \mathbf{y} - \mathbf{f}(\mathbf{w}) - \mathbf{J}\mathbf{\Delta w} \qquad (103)$$

where $f_i(\mathbf{w}) = f(x_i, \mathbf{w})$. The error is therefore

$$e(\mathbf{w} + \mathbf{\Delta w}) = \left(\mathbf{y} - \mathbf{f}(\mathbf{w}) - \mathbf{J}\mathbf{\Delta w}\right)^{\mathrm{T}} \left(\mathbf{y} - \mathbf{f}(\mathbf{w}) - \mathbf{J}\mathbf{\Delta w}\right). \qquad (104)$$

By minimising this quantity, we find the *change* in $\mathbf{w}$ that gives us the largest reduction in the error. We differentiate as before and set to zero:

$$\left(\mathbf{J}^{\mathrm{T}}\mathbf{J}\right) \mathbf{\Delta w}^* = \mathbf{J}^{\mathrm{T}} \left(\mathbf{y} - \mathbf{f}(\mathbf{w})\right) \qquad (105)$$

which we know how to solve.

It is common practice to make a small modification to this:

$$\left(\mathbf{J}^{\mathrm{T}}\mathbf{J} + \lambda\, \mathrm{diag}(\mathbf{J}^{\mathrm{T}}\mathbf{J})\right) \mathbf{\Delta w}^* = \mathbf{J}^{\mathrm{T}} \left(\mathbf{y} - \mathbf{f}(\mathbf{w})\right) \qquad (106)$$

This is a regularisation that penalises large updates to the parameter vector along directions with steep gradients, encouraging movement along shallow gradients and thus a faster convergence. The amount of increase is controlled by the $\lambda$ which is modified iteratively to give the best results. The full algorithm, named after its inventors Levenberg and Marquadt is described in Algorithm 3. This method is widely used for nonlinear regression problems, but should be used cautiously because for complex models, there is the possibility that the loss function will have multiple local minima, and because this is a gradient-based method, it is prone to getting stuck in these local minima. In this circumstance, other optimisation approaches that are not based on gradient descent should be used, for examples, evolutionary/genetic algorithms.

## *Reading*

Sections 1.1 and 3.1, 3.2 of Bishop, Pattern Recognition and Machine Learning.

```
Data: Data $\mathbf{x}, \mathbf{y}$; Initial parameter guess $\mathbf{w}_0$; Function $f(x, \mathbf{w})$;
      Jacobian $Jac(x, \mathbf{w})$; initial values of $\lambda, \nu$.
Result: $\mathbf{w}, e(\mathbf{w})$, parameters that minimmise the least squares
        error, the error.
% Evaluate the error at the initial parameter value
$\mathbf{w} \leftarrow \mathbf{w}_0$;
$e \leftarrow (\mathbf{y} - f(x, \mathbf{w})^{\mathrm{T}}(\mathbf{y} - f(x, \mathbf{w})$;
% Iterate until convergence
$Converged \leftarrow False$;
while $Converged == False$ do
    % Calculate the Jacobian
    $\mathbf{J} \leftarrow \frac{\partial \mathbf{f}}{\mathbf{w}}$;
    % Calculate update for different values of $\lambda$ until
      we improve the error
    $LambdaSet \leftarrow false$;
    while $LambdaSet{==}false$ do
        % Calculate $\Delta\mathbf{w}$ for $\lambda = \lambda_0$ and $\lambda = \lambda_0/\nu$
        $\Delta_1 = \left(\mathbf{J}^{\mathrm{T}}\mathbf{J} + \lambda\, \mathrm{diag}(\mathbf{J}^{\mathrm{T}}\mathbf{J})\right) \setminus \mathbf{J}^{\mathrm{T}}\, (\mathbf{y} - \mathbf{f}(x, \mathbf{w}))$;
        $\Delta_2 = \left(\mathbf{J}^{\mathrm{T}}\mathbf{J} + (\lambda/\nu)\, \mathrm{diag}(\mathbf{J}^{\mathrm{T}}\mathbf{J})\right) \setminus \mathbf{J}^{\mathrm{T}}\, (\mathbf{y} - \mathbf{f}(x, \mathbf{w}))$;
        % Calculate the error at the new parameter values
        $e_1 \leftarrow (\mathbf{y} - f(x, \mathbf{w} + \Delta_1)^{\mathrm{T}}(\mathbf{y} - f(x, \mathbf{w})$;
        $e_2 \leftarrow (\mathbf{y} - f(x, \mathbf{w} + \Delta_2)^{\mathrm{T}}(\mathbf{y} - f(x, \mathbf{w})$;
        if $e_1 < e$ then
            % Set new values of $\mathbf{w}$ and $e$, storing old value
              as $e'$
            $\mathbf{w} \leftarrow \mathbf{w} + \Delta_1$;
            $e' \leftarrow e$;
            $e \leftarrow e_1$;
            % Current $\lambda$ is OK
            $LambdaSet \leftarrow true$;
        else if $e_2 < e$ then
            % Set new values of $\mathbf{w}$ and $e$, storing old value
              as $e'$
            $\mathbf{w} \leftarrow \mathbf{w} + \Delta_2$;
            $e' \leftarrow e$;
            $e \leftarrow e_2$;
            $\lambda \leftarrow \lambda/\nu$;
            % Current $\lambda$ is OK
            $LambdaSet \leftarrow true$;
        else
            $\lambda \leftarrow \lambda \times \nu$;

    end
    if $e'/e < TerminationCriterion$ then
        $Converged \leftarrow true$
    end
end
```

**Algorithm 3**: The Levenberg-Marquadt Algorithm.

# *Classification*

## *Introduction*

In regression, we are interested in predicting the values of a *continuous* variable. In classification problems, we are interested in predicting *categorical* variables. Common examples of classification problems include:

- Determining what type of object is present in an image.
- Sorting documents into different types.
- Determining whether a set of diagnostic tests implies that a patient has a disease.

Classification is, like regression, a *supervised* learning technique. A classifier is trained on some set of training data, and then used to classify unseen examples.

## *The MNIST Dataset*

A classic example of a classification problem, is the MNIST set of handwritten digits. This dataset consists of a training set of 70,000 images of handwritten numerical digits (0–9), divided into a training set of 60,000 images and a test set of 10,000 images. Each image has a class label that indicates which digit it contains. There is also a test set of 10,000 examples. The images in both sets have been rescaled so that they are the same size: $28 \times 28$ pixels

MNIST is one of the most frequently used datasets for classification tasks because the data is well curated and requires little or no preprocessing before it can be used. It is based on an original dataset originally developed by the National Institute of Standards and Technology (NIST) in the USA and has been modified to make it more suitable for machine learning applications: the original dataset had substantial bias, with the training set taken from employees of the American Census Bureau employees, and the testing set from American high school students. Modified NIST – MNIST – rebalances this dataset. Full details of how this was done can be found at the database's website, `http://yann.lecun.com/exdb/mnist/`. A few examples from the MNIST training set are shown in Figure 17.

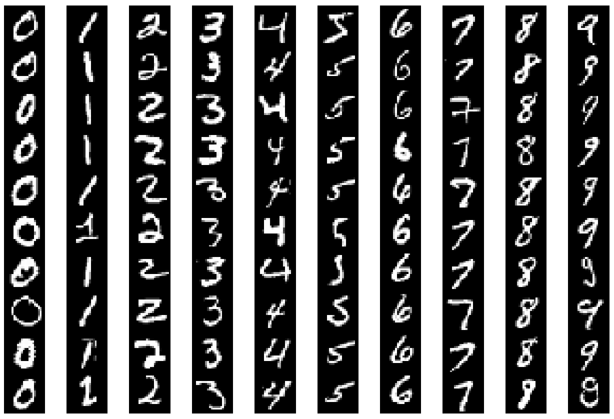

Figure 17: Ten samples from each category of the MNIST training dataset.

We will use the MNIST dataset as a vehicle through which we will study a range of algorithms for classification. All of the methods that we will study will require data that is "vectorial" in nature; that is, we need a way in which our image data can be represented in vector form. The simplest way to do this is to "unroll" the images by rastering across them as shown in Figure 18. One hundred real examples of vectorised MNIST images from each category are shown in Figure 19.

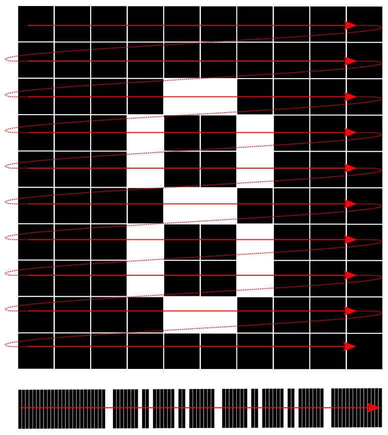

Figure 18: (Top) raster scanning an image to form (Bottom) a vectorised representation.

## *k-nearest-neighbours Classification*

Let us begin to explore the classification problem by considering one of the conceptually simplest ideas of all: we will try to classify MNIST digits according to how similar they are to examples for which we already know the category. This is a very crude form of learning in that all we do is memorise the training set and see how similar new samples are to those that we already know the label for. The simplest possible example of this is nearest-neighbour classification, where for each sample with an unknown category, we find it's nearest-neighbour in the training data. The $k$-nearest-neighbours algorithm is given in Algorithm 4.

The is shown graphically in Figure 20. The $k$-nn method has a major advantage over other methods: its training time is zero – there is no training phase, there is just the training dataset. The disadvantage, though, is that the time taken to make predictions is proportional to the size of both the training and testing datasets: every element of the test set has to be compared to every element of the training set. This could be a major inconvenience if either is very large.

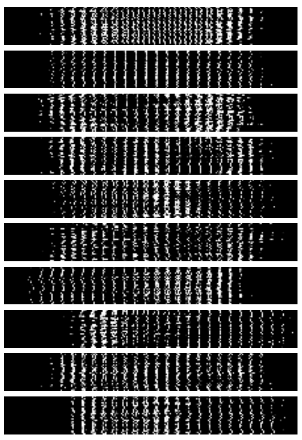

Figure 19: One hundred vectorised samples from each MNIST category. Each row of each sub-image corresponds to a single vectorised sample. Top–bottom: 0–9.

```
Data: A set of labelled training data
Data: A set of unlabelled test data
Data: Integer $k$
Result: For each item in the test set, returns the most common
        label of that items $k$ nearest neighbours in the training
        set.
for each item $x$ in test set do
    for each item $y$ in training set do
        Compute similarity $d(x, y)$
    end
    Find the $k$ most similar items to $x$ in the training set.
    Compute the most common label
end
```

**Algorithm 4**: *k*-nearest neighbours classification.

One might regard it as entirely unreasonable to think that such a simple method could succeed at the task of classifying MNIST digits. There are several objections:

- Vectorising the images loses much of their spatial information. The visual cues that humans rely on are lost (for example closed vs open loops in 5 vs 6), and cannot be recovered from the vector representation alone.
- There is substantial variability between characters (eg the 1's in Figure 17), suggesting that simple pixel-by-pixel comparison will be highly unreliable.

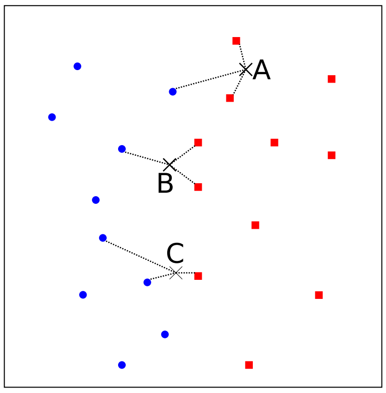

Figure 20: A illustration of the $k$-nearest-neighbours algorithms for $k = 3$. Two classes of labelled training data are shown: blue circles and red squares. The unknown test points, marked as black crosses (A, B, C) are marked along with their 3 nearest neighbours. The majority class of the three neighbours is assigned to the test point: A: red; B: red; C: blue. Notice that the value of $k$ affects this assignments: the nearest neighbour of point C is red.

Nevertheless, it will be instructive for us to see how we this works in practice. For our first experiments, we will choose 10000 random examples from the training dataset, and 1000 random examples from the testing dataset. We will use the Euclidean distance as our measure of similarity, where smaller values mean more similar. For images vectors $\mathbf{x}$ and $\mathbf{y}$, this is given by

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{((\mathbf{x} - \mathbf{y})^{\mathrm{T}}(\mathbf{x} - \mathbf{y}))} = \sqrt{\sum_i (x_i - y_i)^2}. \tag{107}$$

where $i$ indexes the components of the vector (pixels of the image).

In the first instance, we try a very simple $k = 1$ nearest-neighbour classification. The results of this are shown in the *confusion matrix* shown in Figure 21.

We may be justifiably surprised at how well this has done! Some character classes (1) are predicted with 100% accuracy. The overall accuracy (average of the diagonal elements) is 67%. Some characters appear harder than others: 5 is hard to distibnguish from 8 (but not the other way, curiously); 4 is hard to distinguish from 9. Both of these confusions make a lot of sense: they are things we may have difficulty with in cases of poor handwriting.

| T \ P | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 83 | 1 | 1 | 0 | 0 | 0 | 5 | 0 | 10 | 0 |
| 1 | 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 11 | 53 | 2 | 1 | 0 | 3 | 4 | 25 | 0 |
| 3 | 0 | 11 | 2 | 48 | 0 | 1 | 4 | 3 | 28 | 3 |
| 4 | 2 | 9 | 0 | 0 | 42 | 0 | 2 | 3 | 16 | 26 |
| 5 | 2 | 7 | 0 | 4 | 0 | 36 | 2 | 0 | 43 | 6 |
| 6 | 3 | 6 | 0 | 0 | 0 | 1 | 80 | 0 | 10 | 0 |
| 7 | 0 | 11 | 0 | 1 | 0 | 0 | 1 | 75 | 4 | 8 |
| 8 | 2 | 13 | 0 | 6 | 1 | 3 | 3 | 4 | 65 | 3 |
| 9 | 0 | 5 | 1 | 1 | 4 | 0 | 0 | 4 | 2 | 83 |

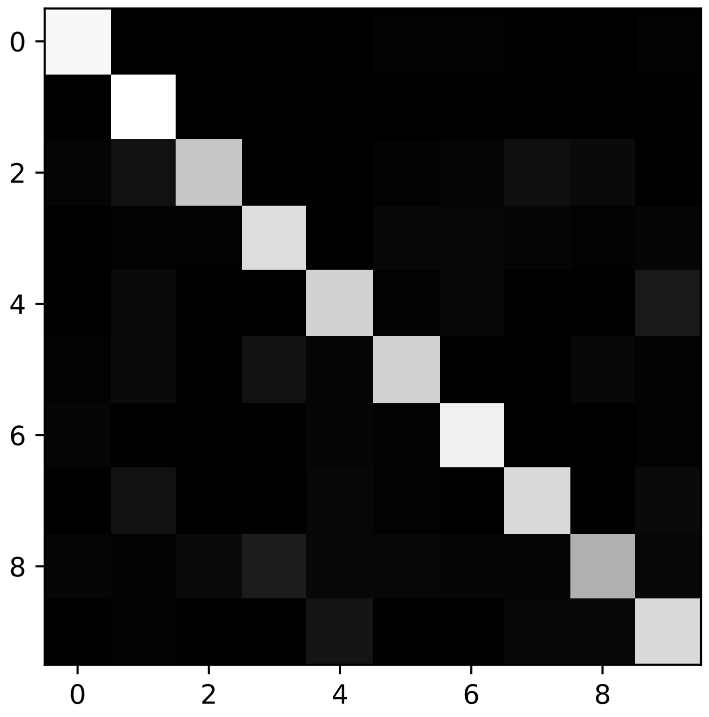

Figure 21: MNIST classification results (Target T vs Prediction P) with *k*NN for $k = 1$.

Can we do better? Let's try some more neighbours. In Figures 22, 23 and 24 we show the confusion matrices for $k = 3$, $k = 5$, and $k = 7$.

| P T | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 95 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 3 | 0 |
| 1 | 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 4 | 14 | 68 | 0 | 0 | 0 | 1 | 2 | 11 | 0 |
| 3 | 2 | 13 | 4 | 64 | 0 | 1 | 3 | 2 | 8 | 3 |
| 4 | 2 | 13 | 1 | 0 | 51 | 0 | 4 | 2 | 3 | 24 |
| 5 | 5 | 13 | 0 | 10 | 1 | 39 | 2 | 0 | 24 | 6 |
| 6 | 2 | 7 | 0 | 0 | 1 | 1 | 88 | 0 | 1 | 0 |
| 7 | 0 | 18 | 2 | 1 | 1 | 1 | 0 | 68 | 3 | 6 |
| 8 | 3 | 18 | 0 | 3 | 1 | 3 | 3 | 4 | 65 | 0 |
| 9 | 1 | 7 | 0 | 1 | 1 | 0 | 0 | 2 | 2 | 86 |

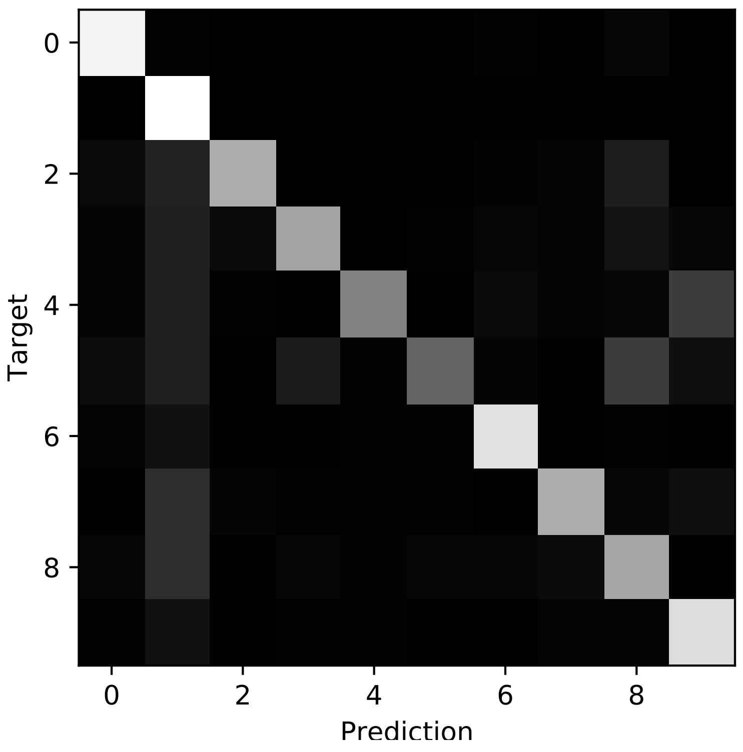

Figure 22: MNIST classification results (Target T vs Prediction P) with $k$NN for $k = 3$.

The overall accuracy in these cases is 72%, 74%, and 75%. The consensus voting approach used in $k$nn appears to bring significant gains when compare to the simple nearest neighbour approach. We do seem to be reaching a point of diminishing returns though, and the confusion matrices show us that whilst 0, 1, 6, and 9 can be identified very accurately indeed with this approach, 3, 4, and 5 are proving much more difficult. Is there anything we can do to improve this?

Let us try something that at first seems to make no sense whatsoever. We will take each of our image vectors, which have $28 \times$

| T \ P | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 97 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 3 | 17 | 69 | 1 | 0 | 0 | 2 | 3 | 5 | 0 |
| 3 | 1 | 19 | 1 | 60 | 0 | 0 | 6 | 3 | 7 | 3 |
| 4 | 2 | 12 | 1 | 0 | 50 | 0 | 5 | 1 | 4 | 25 |
| 5 | 5 | 9 | 0 | 5 | 2 | 51 | 2 | 0 | 19 | 7 |
| 6 | 2 | 7 | 0 | 0 | 1 | 1 | 89 | 0 | 0 | 0 |
| 7 | 0 | 18 | 0 | 0 | 1 | 1 | 0 | 73 | 2 | 5 |
| 8 | 3 | 18 | 1 | 3 | 0 | 1 | 4 | 5 | 65 | 0 |
| 9 | 1 | 9 | 0 | 0 | 0 | 0 | 0 | 1 | 4 | 85 |

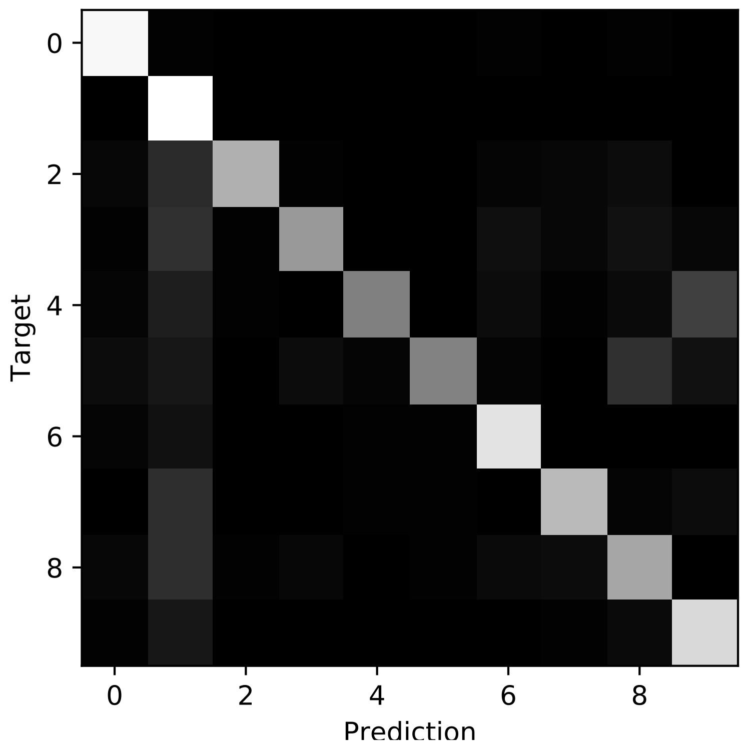

Figure 23: MNIST classification results (Target T vs Prediction P) with *k*NN for $k = 5$.

| T \ P | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 95 | 1 | 0 | 0 | 0 | 0 | 3 | 0 | 1 | 0 |
| 1 | 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 17 | 70 | 0 | 0 | 0 | 2 | 4 | 6 | 0 |
| 3 | 1 | 20 | 0 | 61 | 0 | 1 | 6 | 2 | 5 | 4 |
| 4 | 3 | 9 | 0 | 0 | 55 | 0 | 4 | 1 | 2 | 26 |
| 5 | 5 | 9 | 1 | 5 | 1 | 51 | 3 | 0 | 17 | 8 |
| 6 | 2 | 7 | 0 | 0 | 0 | 1 | 90 | 0 | 0 | 0 |
| 7 | 1 | 17 | 1 | 0 | 1 | 0 | 0 | 75 | 1 | 4 |
| 8 | 3 | 16 | 1 | 2 | 0 | 1 | 4 | 5 | 66 | 2 |
| 9 | 1 | 7 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 88 |

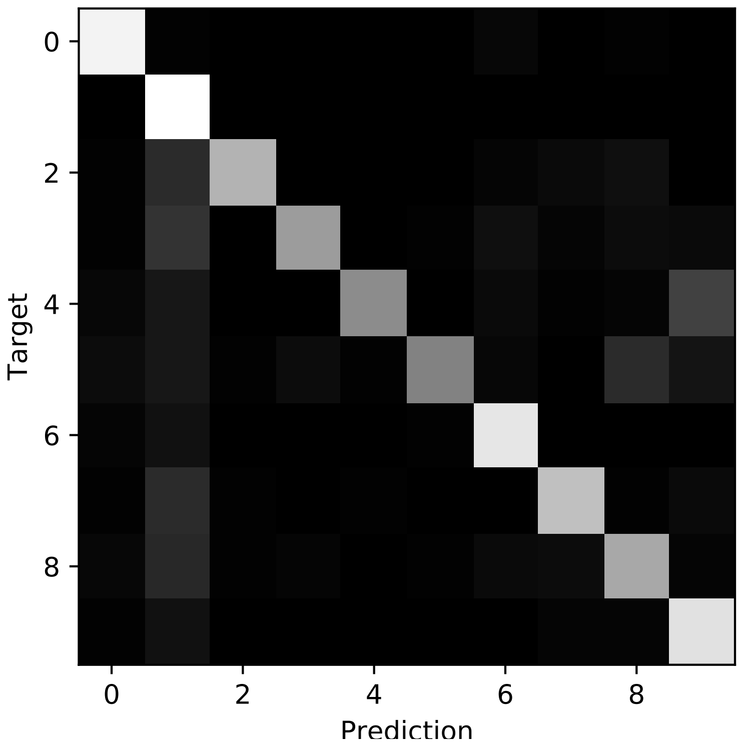

Figure 24: MNIST classification results (Target T vs Prediction P) with *k*NN for $k = 7$.

28 = 784 elements, and we will take its scalar (dot) product with each of 40 *random* vectors; vectors where the components are drawn independently from a normal distribution $\mathcal{N}(0,1)$ with mean 0 and variance 1. For image vectors $\mathbf{v}_i$, and random vectors $\mathbf{r}_i$ (both assumed to be column vectors) we compute new vectors

$$\begin{pmatrix} \mathbf{z}_1 & \mathbf{z}_2 & \dots & \mathbf{z}_N \end{pmatrix} = \begin{pmatrix} \mathbf{r}_1^{\mathrm{T}} \\ \mathbf{r}_2^{\mathrm{T}} \\ \dots \\ \mathbf{r}_M^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \dots & \mathbf{v}_N \end{pmatrix}. \tag{108}$$

The $\mathbf{z}_i$ are said to be *random projections* of the original image vectors $\mathbf{v}$. There is one of these per sample in the dataset, but each now has only 40 components, instead of the original 768. Superficially this seems to be a very strange thing to do indeed: why would we even consider this? That is a question to which we will return in due course. For the moment though, let us just try it. We form new training and test sets by randomly project every element of both sets onto the same 40 random vector, and we apply *k*nn, with $k = 7$ to the random projections. The results of this are shown in Figure 25.

Something quite dramatic has happened. The overall accuracy has increased from 75% to 87%! The troublesome characters 3, 4 and 5 are now classified with $> 80\%$ accuracy. The random projection must have done something quite dramatic to our data. We will next try to understand what has happened, and why. It turns out that computing distances between high-dimensional vectors is not as innocuous as it might seem, and there is a hidden danger.

## *The Curse of Dimensionality*

The underlying reason that reducing the dimensionality of the data using a random projection was useful is that the properties of high dimensional vector spaces are strange and counter intuitive. For a simple example of how our intuition breaks down, consider the following example.

A *hypercube* is the *n*-dimensional analogue of a square in 2d, and cube in 3d. In each dimension, the cube has a side of length $2r$ so that the centre of each of its faces is a distance $r$ from the centre of the hypercube. Consider now the *hypersphere* that the hypercube encloses. The hypersphere, which is defined as the set of points a distance $r$ from some central point, intersects with the hypercube only at the centres of the faces. This is shown in 3d is Figure 26

| T \ P | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 98 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 3 | 4 | 79 | 1 | 0 | 1 | 4 | 3 | 5 | 0 |
| 3 | 0 | 4 | 2 | 84 | 0 | 1 | 1 | 3 | 2 | 3 |
| 4 | 0 | 1 | 0 | 0 | 85 | 0 | 1 | 2 | 2 | 9 |
| 5 | 0 | 2 | 0 | 3 | 1 | 86 | 4 | 2 | 1 | 1 |
| 6 | 1 | 0 | 0 | 0 | 1 | 6 | 91 | 1 | 0 | 0 |
| 7 | 0 | 3 | 1 | 1 | 1 | 0 | 0 | 91 | 0 | 3 |
| 8 | 1 | 0 | 5 | 13 | 3 | 4 | 1 | 2 | 70 | 1 |
| 9 | 0 | 1 | 0 | 0 | 6 | 0 | 0 | 2 | 2 | 89 |

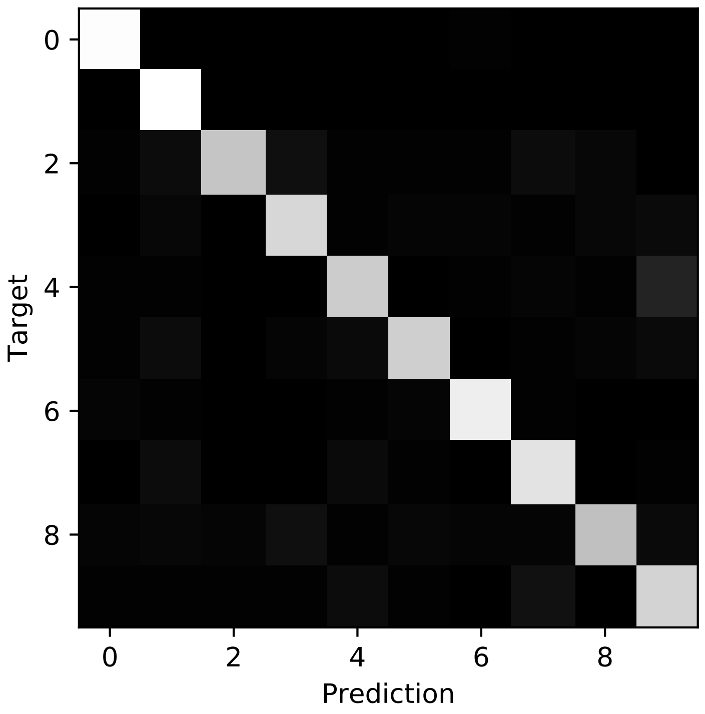

Figure 25: MNIST classification results (Target T vs Prediction P) with $k$NN for $k = 7$ using 40 random projections of the data.

By definition, all of the points on the sphere are a distance $r$ from the centre. The faces of the cube are also $r$ from the centre. How far away from the centre are the corners? The answer is trivial to compute, but has surprising implications. In 2d, the corner of a square is $\sqrt{r^2 + r^2} = r\sqrt{2}$ from the centre. In 3d, the corner of a cube is $\sqrt{r^2 + r^2 + r^2} = r\sqrt{3}$. The pattern is obvious and follows directly from Pythagoras' theorem. In $n$-dimensions, the corner of a hypercube (of side length $2r$) is $\sqrt{\sum_{i=1}^{n} r^2} = r\sqrt{n}$. We plot this relationship in Figure 27. Remembering that the surface of the hypersphere is always $r$ from the origin, we see that the distance to the corner is many times the sphere radius in high dimensions. In other words, although the sphere touches the cube in the centre of all of its faces, it does not get near to the corners!

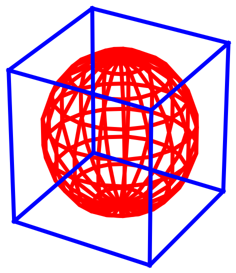
Figure 26: A sphere in 3d and its enclosing hypercube

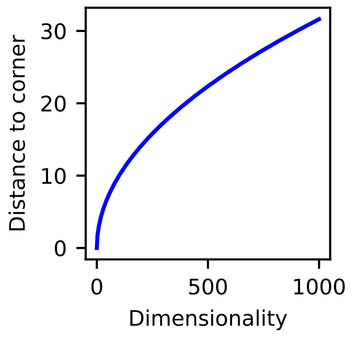

Figure 27: The distance from the centre to the corners of a hypercube of side $r = 1$ increases as the square-root of the dimensionality.

Let us look at this in a different way and consider the fraction of the the volume of the hypercube that is occupied by its enclosed hypersphere. In 2d, a square of side $2r$ has area $4r^2$. The enclosed circle has area $\pi r^2$, and so the circle occupies a fraction $\pi/4 \approx 0.785$ of the square. In 3d, the cube has volume $8r^3$; the sphere has volume $4\pi r^3/3$, and the ratio is $4\pi/24 = 0.52$. This is already a substantial decrease! In $n$-dimensions, the volume of the hypercube is $(2r)^n$, and the volume of the hypersphere can be shown to be given by $\frac{\pi^{\frac{n}{2}}}{\Gamma\left(\frac{n}{2}+1\right)} R^n$, where $\Gamma(x)$ is the Gamma function, an extension of the factorial function that is beyond the scope of this module to study in detail. Some information can (naturally!) be found at `https://en.wikipedia.org/wiki/Gamma_function`. Using these results, we plot the sphere/cube ratio in Figure 28.

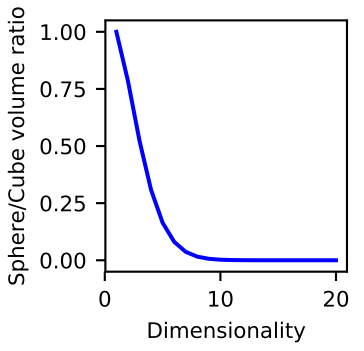

Figure 28: The ratio of the volume of a hypersphere to its enclosing hypercube as a function of dimensionality.

It is remarkable how quickly the ratio drops to zero: in 10d, nearly all of the volume in the cube is outside of the sphere. When combined with our previous result on the distance from the centre of the cube to its corners, we may conclude that nearly all of the volume of a hypercube is near to the corners, and that almost no volume is near the centre. This can, if you so wish, be verified numerically by constructing a grid of evenly spaced points in high dimensions and seeing how many are within distance $r$ of the centre: very few.

To conclude this discussion, consider two hyperspheres, of radius $r$ and $r - \delta$ respectively (with $\delta \ll r$). The volume of the larger sphere is $\alpha r^n$, where $\alpha$ is a constant that depends on the dimensionality $n$; and the volume of the smaller sphere is $\alpha(r - \delta)^n$. The volume of the "shell" that is inside the larger sphere but outside the smaller sphere is therefore $\alpha\left(r^n - (r - \delta)^n\right)$. As a proportion of the larger sphere, the shell occupies a volume

$$\frac{V_{\text{shell}}}{V_{\text{sphere}}} = \frac{\alpha\left(r^n - (r-\delta)^n\right)}{\alpha r^n} \tag{109}$$

$$= 1 - r^{-n}(r-\delta)^n \tag{110}$$

$$= 1 - \left(r^{-1}(r-\delta)\right)^n \tag{111}$$

$$= 1 - \left(1 - \frac{\delta}{r}\right)^n \tag{112}$$

Taking the limit as the dimension tends to infinity gives

$$\lim_{n\to\infty} \frac{V_{\text{shell}}}{V_{\text{sphere}}} = \lim_{n\to\infty} 1 - \left(1 - \frac{\delta}{r}\right)^n \tag{113}$$

$$= 1 \tag{114}$$

because $1 - \frac{\delta}{r} < 1$. Thus most of the volume of the hypersphere is concentrated in a thin shell around its edge and most of the volume is at its edge.

Although these results are intrinsically interesting, it is not clear that they are immediately relevant to our problem of doing distance-based classification of MNIST digits, so let us perform a simple numerical experiment. For a range of dimensionalities, we generate $10^6$ uniformly randomly distributed data points and compute the distances between all pairs of points. We then find the maximum and minimum distances between points and compute the ratio of the range of distances as compare to the minimum distance, $(d_{\text{max}} - d_{\text{min}})/d_{\text{min}}$. The resulting graphs are shown in Figure 29.

The figure provides an empirical verification of a well-known result:

$$\lim_{n\to\infty} \mathbb{E}\left(\frac{d_{\text{max}} - d_{\text{min}}}{d_{\text{min}}}\right) \to 0 \tag{115}$$

which is a statement that in high dimensions, the difference between minimum and maximum distances between points tends towards zero, and hence all distances become "similar". This result has profound implications for any algorithm that requires computation and comparison of pairwise distances in high dimensions.

To what extent is this relevant in MNIST? To test this, we compute the pairwise distances between 1000 points from the test set and 1000 points from the training set and plot a histogram of their distribution (Figure 30). This shows that there are no "small" distances in the data, a direct result of each pairwise distance being the sum of the squares of each of 784 components, and the distribution of distances is roughly normal, with a mean/median of $\approx 2300$ and a standard deviation of $\approx 300$. This means that 68% of pairwise distances lie between 2000 and 2600, and 95% between 1700 and 2900. The range of distance is indeed compressed, but perhaps not as much as we might have expected. Why is this?

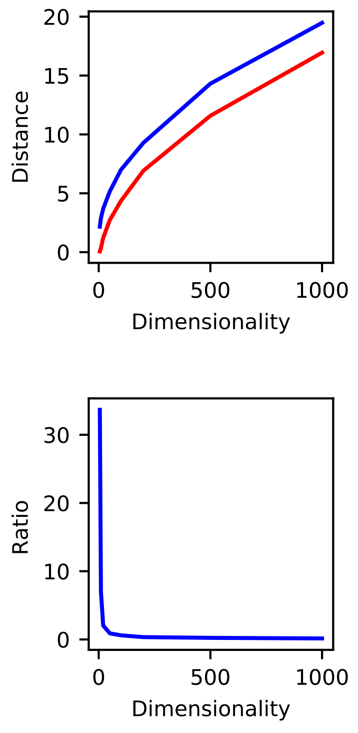

Figure 29: *Top*: Minimum (red) and maximum (blue) pairwise distances in a set of $10^6$ uniformly randomly distributed data points as a function of dimensionality. *Bottom*: The ratio $(d_{\text{max}} - d_{\text{min}})/d_{\text{min}}$.

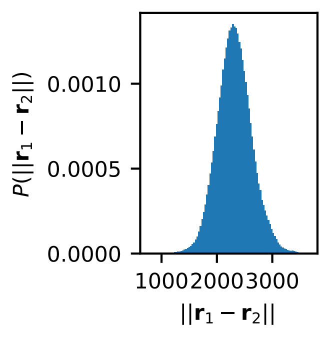

Figure 30: Distribution of pairwise distances in pixel space between 1000 examples from the MNIST test set, and

One problem with the analysis we have done so far is that we have considered points that are distributed throughout a hyper-volume. "Real" data does not behave in this way; it tends to lie on some low-dimensional subspace. This is shown in Figure 31, which shows the mean and standard deviation of 1000 MNIST images, together with a mask that shows which pixels vary, and which do not. 175 of the 784 pixels in the image, which correspond to dimensions in the underlying vector space, have zero variance. In this case they also have zero mean, but that is less important. This means that we could, trivially, reduce the dimensionality of the data to 609, by just retaining those pixels which vary and therefore contain useful information.

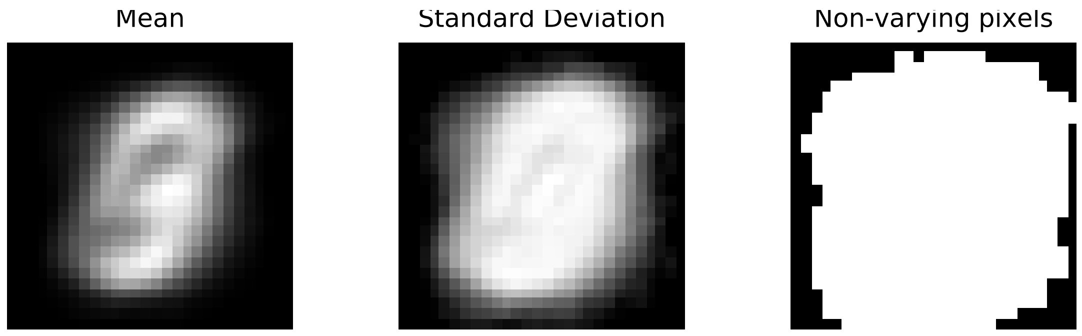

Figure 31: Image statistics from 1000 MNIST images. *Left*: The pixel-mean. *Centre*: the pixel standard deviation. *Right*: Black pixels are those with zero variance.

In Figure 32 we show the cumulative sum of the pixel variances, and see that nearly all of the variance is from 500 pixels, and around 75% of it is from only half of the pixels, suggesting that further reductions in dimensionality may be possible. In fact, by looking at each pixel in isolation, we are only scratching the surface of what is possible in terms of reducing the dimension of the data in a meaningful way. One easy-to-picture reason why we might be able to reduce the dimensionality further is that correlations between pixels mean that the value of one pixel can be predicted from the value of another, but more complex situations are possible. For instance, imagine that you have a picture or a poster that is rolled up. When unrolled, points on the poster can be described quite adequately in a 2d coordinate system. However, when the poster is rolled up, it becomes a three-dimensional object that requires a 3d coordinate system. If we could find a transformation that could "unroll" the poster, then we could work with it in 2d coordinates. This argument can be extended to higher dimensions, and the task of *dimensionality reduction* is to find the transformations that allows to represent data using their *intrinsic dimensionality*: the number of meaningful degree of freedom.

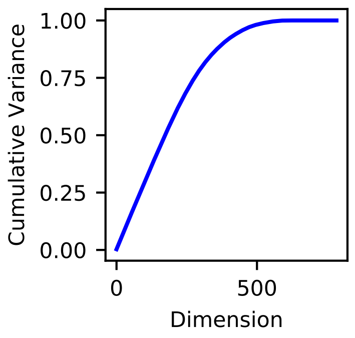

Figure 32: Cumulative variance as a function of dimensionality. The pixels were ordered by variance and the cumulative sum was calculated. This shows that around 75% of the variance in the data is from fewer than half of the pixels.

To understand what this means in terms of MNIST, let us consider what the underlying degrees of freedom are.

- We have ten digit class.
- Members of each class may vary by width and height (although

this is somewhat mitigated by the alignment process.)

- Each character class will have some shape variation – for example, whether one's have a serif and/or a base (eg 1 vs 1).

An order-of-magnitude estimate suggests that there may be a few ten's of intrinsics degree of freedom in the data. The question now is, how do we find a transformation that extracts them?

## *Dimensionality Reduction*

We will limit our discussion to *linear* methods for dimensionality reduction; that is, methods that apply a single global linear transformation (rotation, scale, shear etc) to the data. To understand what this might do, imagine a sheet of paper embedded in 3d. If we can rotate and shift the sheet so that it aligns with the 3d coordinate system $(x, y, z)$, then we can discard the coordinate that aligns with the thinnest dimension of the sheet of paper (its thickness), and just retain those coordinates that describe the main variations in the sheet: we can reduce the problem from 3d to 2d. We will be trying to do the same here: to shift and rotate the data to find a coordinate system that describes the main variation whilst preserving the internal structure of the data. Our hope is that by reducing the dimensionality of the data in a way that preserves its internal structure, we can mitigate some of the issues that we see in high dimensional spaces.

There are many ways in which the dimensionality of a dataset can be reduced; we will focus on the random*random projection*. This is an attractive method because it is computationally very simple and low-cost, but there is some elegant theory behind it that provides much insight into the problems that high dimensionality causes.

## *Random Projections*

We have already seen random projections in action, and have seen, experimentally, how much benefit they can bring to a high-dimensional learning problem. The central idea is exceptionally simple. Given an original dataset containing $N$ samples in $M$ dimensional space, arranged as an $M \times N$ matrix $\mathbf{X}$ (one sample per column):

- Generate $K$ random vectors with $M$ components randomly sampled from $\mathcal{N}(0, 1)$. Arrange these as a matrix $\mathbf{R}$ of size $M \times K$, with one vector per column.

- Normalise the columns of $\mathbf{R}$ so each has unit length.

- Compute $\mathbf{X}' = \mathbf{R}^T\mathbf{X}$

- The columns of $\mathbf{X}'$ contain the samples projected (Figure 33) into the lower dimensional space define by the $K$ random vectors.

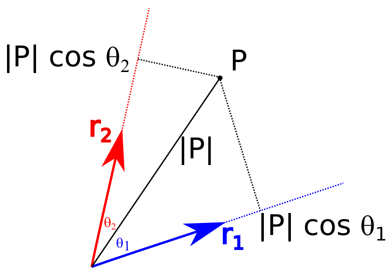

Figure 33: Projection of a point $\mathbf{P}$ onto two unit random vectors $\mathbf{r}_1$ and $\mathbf{r}_2$. Noting that $\mathbf{r}_1$ and $\mathbf{r}_2$ are unit vectors, the projection is given by the dot product $mathbf{P} \cdot \mathbf{r} = |P| \cos\theta$.

We saw, in $k$-nn classification, that this works. But why does it work? The key idea is that whilst distances in high-dimensional spaces are of limited use, *if* we could could map the the points into some low-dimensional space (where distances are more useful) *in a way that preserves the local structure of the data*, then we can render distances useful again.

Is it possible to find such a map? It turns out that random projections, subject to a few constraints, can be be shown to generate just such a mapping. The key to this is a mathematical result known as the *Johnson-Lindenstrauss lemma*[1]. This is a statement that points in a high-dimensional space can be mapped onto a low-dimensional space in such a way that relative distances between points are preserved, whilst the absolute distances are reduced. The lemma states that:

[1] Johnson, William B., and Joram Lindenstrauss. "Extensions of Lipschitz mappings into a Hilbert space." Contemporary mathematics 26.189-206 (1984): 1.

Given a set $X$ of $N$ data points in $\mathbb{R}^M$ ($M$-dimensional real space), there is a linear map $f : \mathbb{R}^M \mapsto \mathbb{R}^K$ that maps $X$ onto a *random* lower-dimensional space $\mathbb{R}^K$. The map obeys

$$(1-\varepsilon)\|x_1 - x_2\|^2 \leq \|f(x_1) - f(x_2)\|^2 \leq (1+\varepsilon)\|x_1 - x_2\|^2 \qquad (116)$$

for all $x_1, x_2 \in X$ and for $0 < \varepsilon < 1$ and $K > 8\ln(N)/\varepsilon^2$. It has also been proven [2] that this holds when the function $f$ is a random, orthonormal linear transformation. A proof of this result is far beyond the scope of this course. Let us try to unpick its implications.

[2] Frankl, Peter, and Hiroshi Maehara. "The Johnson-Lindenstrauss lemma and the sphericity of some graphs." Journal of Combinatorial Theory, Series B 44.3 (1988): 355-362.

Firstly, we need to be clear of the central result here: Johnson-Lindenstrauss and its extension by Frankl and Maehara together state that a "random, orthonormal linear transformation" can be used to project a set of points onto a lower-dimensional subspace whilst preserving the distances between them to within some bounds controlled by $\epsilon$: distances between points are scaled by a factor between $1-\varepsilon$ and $1+\varepsilon$. Noting the condition that $K > 8\ln(N)/\varepsilon^2$, this means that smaller values of $\varepsilon$ (better distance preservation) requires higher values of $K$.

One now comes across yet another remarkable property of high-dimensional spaces. The requirement that the the random vectors be orthormal – $\mathbf{r}_i \cdot \mathbf{r}_j = 1$ if i=j else 0 – seems like it could be rather onerous. There are well-established methods for doing this (such as Gramm-Schmidt orthogonalisation), but they carry a computational cost. Fortunately, high dimensionality saves us and it turns out that an explicit normalisation step is often not needed, because in very high-dimensional spaces, the probability that two random vectors will be orthogonal tends towards 1! We demonstrate this numerically in Figure 34, noting that because $\mathbf{x} \cdot \mathbf{y} = |x||y|\cos\theta$, $\mathbf{x} \cdot \mathbf{y} = 0$ means that the $\cos\theta = 0$ corresponds to an angle of $90°$ which means that $\mathbf{x}$ and $\mathbf{y}$ are orthogonal. These results demonstrate that as the dimensionality increases, the probability that two random vectors are are *not* orthogonal tends to zero. This is again a consequence of the dimensionality, and it can be shown that the number of sets of nearly orthonal vectors is exponential in the dimensionality[3]. If the dimensionality of our problem is high enough, random vectors are self-orthogonalising!

[3] Kainen, Paul C.; Kůrková, Věra (1993), "Quasiorthogonal dimension of Euclidean spaces", Applied Mathematics Letters, 6 (3): 7âĂŞ10, doi:10.1016/0893-9659(93)90023-G, MR

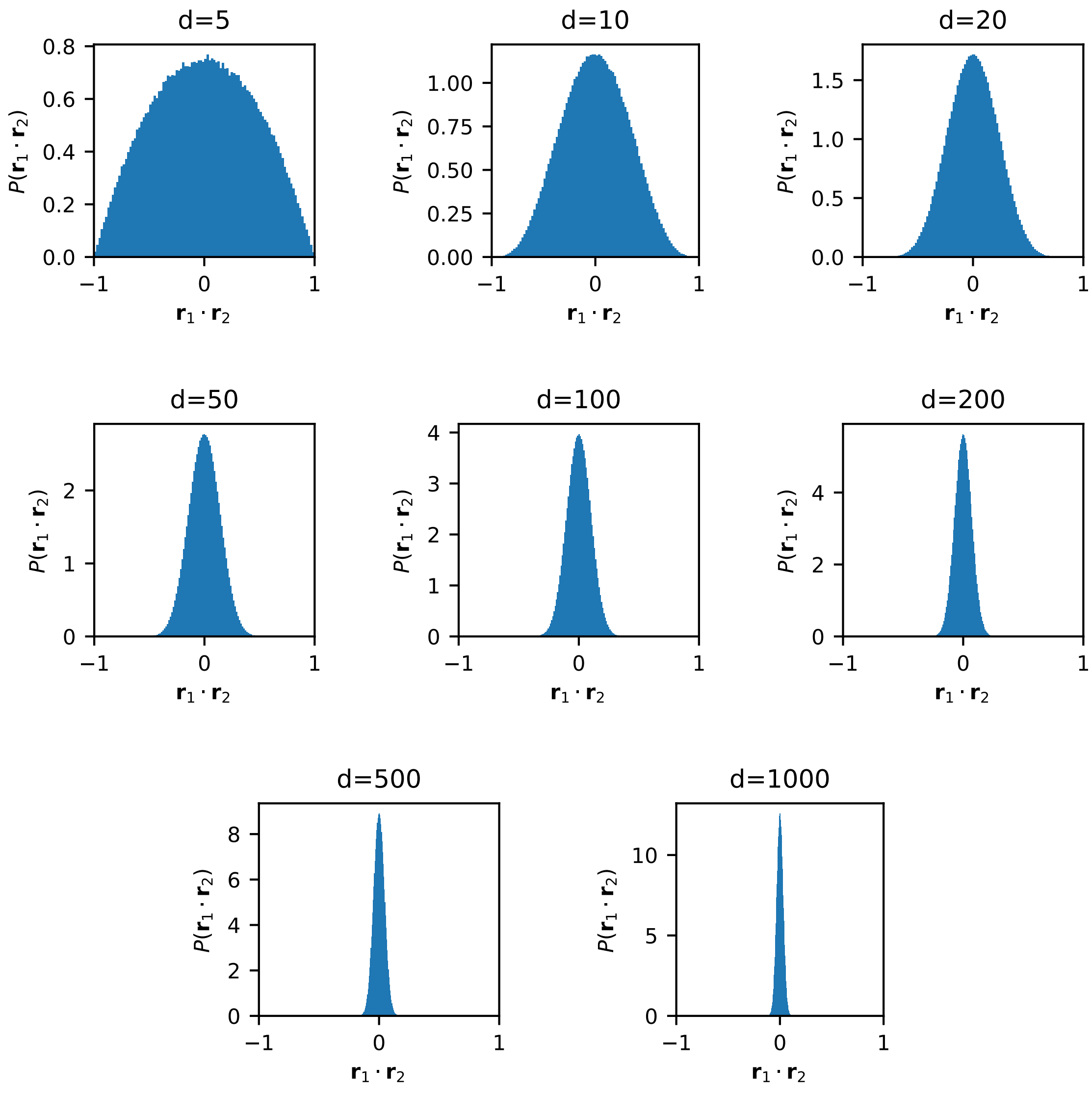

Figure 34: $P(\mathbf{r}_i \cdot \mathbf{r}_j | i \neq j)$ from 100,000 normalised random vectors in different dimensional spaces.

Finally, let us investigate the effect that random projection has on the pairwise distances. We repeat the calculation used to generate Figure 30, but this time with the data projected on to forty random vectors, and distances computed in the lower-dimensional space. The results are shown in Figure 35. The mean/median of the distribution has been reduced to $\approx 580$ from $\approx 2300$, and the standard deviation to $\approx 100$ from $\approx 300$. This does not seems entirely consistent with the Johnson-Lindenstrauss lemma, which stated that distances would be preserved. Why is this?

Although the main implications of Johnson-Lindenstrauss seem clear, there is a sublety. One would expect dimensionality reduction to *reduce* the distances between points, not to preserve them, but J-L seems to say the opposite. However, it is perhaps more helpful to rewrite the theorem as

$$1 - \varepsilon \leq \frac{\|f(x_1) - f(x_2)\|^2}{\|x_1 - x_2\|} \leq 1 + \varepsilon \tag{117}$$

which allows us to see that J-L is in fact a theorem about the *relative* distances between points: it states that it is the relative distances that are preserved, whilst the absolute distances become reduced due to dimensional scaling. This is what makes it such a powerful result for machine learning: we can use random projections to reduce the dimensionality and overcome the "curse" at a very low computational cost, whilst preserving the structure (relative distances) within the data. This is why we see such a dramatic performance improvement in $k$-nn classification on MNIST digits.

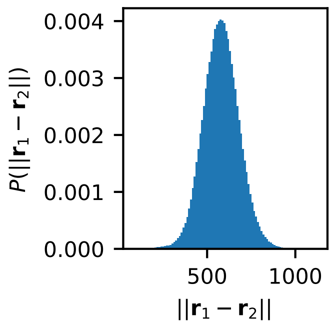

Figure 35: Distribution of pairwise distances following projection onto forty random vectors in pixel space between 1000 examples from the MNIST test set, and 1000 examples from the training set.

## *Linear and Quadratic Discriminant Analysis*

$k$-nearest-neighbours is a classification technique that is *disrcrimative*: it is able to discriminate between different classes of data on the basis of their nearest neighbours in the training set. We will now look at a different approach to classification based on a *generative* model. The basic idea is that we build probabilistic models of the data classes and will use these to derive explicit boundaries between the classes (Figure 36). Thus, training learns where the boundaries lie; inference determines where new data lies in relation to the boundaries. We will develop the mathematics for a linear binary classifier for which the decision boundary will be a linear combination of variables that best discriminates between pre-defined groups of points.

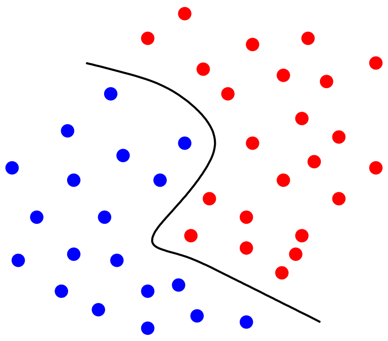

Figure 36: An example of an explicit classification boundary between two classes.

Our approach will make use of Bayes' theorem and will hence require us to define our prior knowledge and the likelihood. First we define our **prior knowledge** of the data: we define prior probabilities

$$P(\mathbf{x} \in \Pi_i) = \pi_i \tag{118}$$

that express the probability that a randomly selected observation $\mathbf{x}$ is in class $\Pi_i$. This requires us to understand something of the prior distribution of the classes. For MNIST, the prior is uniform and the same for all classes because the data is balanced. This may, of course, not always be the case.

We now define the likelihood of an observation, given that it is coming from a specified class, as the conditional probability

$$P(\mathbf{x}|\Pi_i) = f_i(\mathbf{x}) \tag{119}$$

A simple application of Bayes' theorem allows us to calculate the *posterior* probability $P(\Pi_i|\mathbf{x})$, that is, the probability that a given point $\mathbf{x}$ belongs to class $\Pi_i$. For a two-class model we have

$$P(\Pi_i|\mathbf{x}) = \frac{P(\mathbf{x}|\Pi_i)P(\Pi_i)}{P(\mathbf{x})} \tag{120}$$

$$= \frac{f_i(\mathbf{x})\pi_i}{f_1(\mathbf{x})\pi_1 + f_2(\mathbf{x})\pi_2} \tag{121}$$

So, if we know the prior distributions of the groups, and the distribution of points within each of the groups, then we can compute the probability of a specific point being in each of the groups, and assign the point to the class to which it belongs with the highest probability. That is,

$$\frac{P(\Pi_1|\mathbf{x})}{P(\Pi_2|\mathbf{x})} > 1 \mapsto \mathbf{x} \in \Pi_1 \tag{122}$$

otherwise $\mathbf{x} \in \Pi_2$. Re-expressing in terms of $f$ and $\pi$ we find that $\mathbf{x}$ belongs to $\Pi_1$ if

$$\frac{f_1(\mathbf{x})}{f_2(\mathbf{x})} > \frac{\pi_2}{\pi_1} \tag{123}$$

otherwise to $\Pi_2$. If the ratios are equal, then randomly (and with equal probability) assign the point to either class.

In order to implement this, we either need to know or to assume something about the properties of our classes. We will assume the following for a simple two-class binary classifier:

- Each data point belongs to exactly one of exactly two distinct and identifiable groups, $\Pi_1$ and $\Pi_2$
- The two groups are normally distributed with different means $\bar{\mathbf{x}}_1$ and $\bar{\mathbf{x}}_2$ but identical *covariances* $\mathbf{\Sigma}$.

Covariance is a measure of the extent to which two variables change together. It is defined for two variables $X$ and $Y$ as $\Sigma_{X,Y} = \mathbb{E}\left[(X-\bar{X})(Y-\bar{Y})\right]$ (where $\bar{X}$ is the mean value of $X$). and is related to the correlation $\rho_{x,y}$ by $\rho_{x,y} = \Sigma_{x,y}/\sigma_x\sigma_y$. As such, positive covariance implies that two variables increase/decrease together, whilst a negative covariance implies that as one increases, the other tends to decrease. For a multivariate problem, we can compute a *covariance matrix* $\mathbf{\Sigma}$ with components

$$\Sigma_{ij} = \frac{1}{N-1}\sum_{n=1}^{N}\left(x_i^{(n)} - \bar{x}_i\right)\left(x_j^{(n)} - \bar{x}_j\right) \tag{124}$$

which describe the covariance between the variables $x_i$ and $x_j$, measured over $N$ samples. It should be noted that when we compute the mean and covariance from data, they are known as the *sample mean* and *sample covariance* and are not necessariy the same as the true mean and covariance. This is especially problematic when we only have a small number of data points. **Caution: care is needed when computing the sample covariance from data.** This is because the class means are different so you cannot compute the mean of the whole dataset. You **must** align the means of the classes first.

There is a simple way to compute $\mathbf{\Sigma}$ if we organise our dataset in a sensible way. First, we construct the matrix $\mathbf{X}$ with components $X_{ij} = x_i^{(j)}$; that is, each *row* of $\mathbf{X}$ corresponds to a variable, and each *column* corresponds to a sample. Then, compute the mean of every row (variable) and tile the resulting column vector side-by-side $N$ times such that every column is identical and every row contains the mean value of the corresponding variable in every column. We will call the new matrix $\bar{\mathbf{X}}$. Then, we form the covariance matrix as

$$\mathbf{\Sigma} = \frac{1}{N-1}\left(\mathbf{X}-\bar{\mathbf{X}}\right)\left(\mathbf{X}-\bar{\mathbf{X}}\right)^{\mathrm{T}} = \tilde{\mathbf{X}}\tilde{\mathbf{X}}^{\mathrm{T}}. \tag{125}$$

Given the covariance matrix, we can then build a probabilistic model of the class-conditional likelihood, which is a multivariate distribution where the variables are *not* independent. This can be modelled, for classes $n = \{1, 2\}$, as:

$$P(\mathbf{x}|\Pi_n) = f_n(\mathbf{x}) = \frac{1}{(2\pi)^{r/2}\left|\mathbf{\Sigma}\right|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\bar{\mathbf{x}}_n)^{\mathrm{T}}\mathbf{\Sigma}^{-1}(\mathbf{x}-\bar{\mathbf{x}}_n)\right) \tag{126}$$

We have made two strong assumptions in writing down this distribution. First, we assume that each class obeys a normal distribution. Secondly, we assume that the covariance of each class is the same. There are no guarantees that this will hold for any particular example. The method can be modified to permit different covariance structure for each class and we will do this shortly.

Notice that if the variables are independent, the covariance matrix is diagonal and contains only the variances of each variable. This allows the distribution to be factorised into product form.

These distributions model the distributions of the points in our two classes, and permits us to formulate the classification rule. We will now derive a rule that maximally separates the two groups. First, consider the logarithm of the ratio of the two probability distributions:

$$\ln\frac{f_1(\mathbf{x})}{f_2(\mathbf{x})} = \left(\bar{\mathbf{x}}_1-\bar{\mathbf{x}}_2\right)^{\mathrm{T}}\mathbf{\Sigma}^{-1}\mathbf{x} - \frac{1}{2}\left(\bar{\mathbf{x}}_1-\bar{\mathbf{x}}_2\right)^{\mathrm{T}}\mathbf{\Sigma}^{-1}\left(\bar{\mathbf{x}}_1+\bar{\mathbf{x}}_2\right) \tag{127}$$

The second term on the RHS of this equation is independent of $\mathbf{x}$ and we rewrite

$$\left(\bar{\mathbf{x}}_1-\bar{\mathbf{x}}_2\right)^{\mathrm{T}}\mathbf{\Sigma}^{-1}\left(\bar{\mathbf{x}}_1+\bar{\mathbf{x}}_2\right) = \bar{\mathbf{x}}_1^{\mathrm{T}}\mathbf{\Sigma}^{-1}\bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_2^{\mathrm{T}}\mathbf{\Sigma}^{-1}\bar{\mathbf{x}}_2 \tag{128}$$

We note that modifying Eq. (127) slightly we have

$$L(\mathbf{x}) = \log_e\frac{f_1(\mathbf{x})\pi_1}{f_2(\mathbf{x})\pi_2} = \mathbf{m}^{\mathrm{T}}\mathbf{x} + c, \tag{129}$$

which is linear in $\mathbf{x}$ and has "gradient"

$$\mathbf{m} = \mathbf{\Sigma}^{-1}\left(\bar{\mathbf{x}}_1-\bar{\mathbf{x}}_2\right) \tag{130}$$

and "intercept"

$$c = -\frac{1}{2}\left(\bar{\mathbf{x}}_1^{\mathrm{T}}\mathbf{\Sigma}^{-1}\bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_2^{\mathrm{T}}\mathbf{\Sigma}^{-1}\bar{\mathbf{x}}_2\right) + \log_e\frac{\pi_1}{\pi_2} \tag{131}$$

Finally, we observe, by reference to Eq. (123) that

$$\text{if } L(\mathbf{x}) > 0 \text{ assign } \mathbf{x} \text{ to } \Pi_1 \tag{132}$$

else assign $\mathbf{x}$ to $\Pi_2$. The "straight line" defined by Eq. (130) defines a boundary between the classes.

The particular form of LDA we have introduced here is *Gaussian LDA*, and the the function $\mathbf{m}^T\mathbf{x}$ is referred to as *Fisher's linear discriminant function*.

The application of LDA is straightforward: the class means and covariance are computed from the training data, and this permits equations (130) and (131) to be evaluated.

LDA makes two strong assumptions about our data. First, we have imposed a Gaussian distribution on the data without any justification for doing so. This will hold for some problems, but is not guaranteed to hold in the general case. Caution is also necessary when working with small sample numbers because the estimates of the parameters of the distribution (mean, covariance) will be subject to significant uncertainty. Secondly, we have assumed that the classes differ only by their means, and that they have the same covariance structure. This is a very strong assumption that is very unlikely to be true in most situations, although this may not stop LDA from being an effective classifier. One advantage that it has is that it allows the covariance to be computed from the dataset as a whole, which can produce a more robust estimate. One can, of course, argue about whether a more robust estimate of the wrong quantity is the correct thing to do.

Fortunately, it is not too difficult to extend the analysis to the case where the covariances of the classes are not equal. A significant quantity of algebra is required and we do not reproduce them here[4] If the covariances of the two groups are given by $\mathbf{\Sigma}_1$ and $\mathbf{\Sigma}_2$ then the discriminant function (Eq. (129)) becomes

[4] See Chapter 8 of Izenman, Multivariate Statistical Analysis, Springer (New York) 2013

$$Q(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x} + \mathbf{b}^T\mathbf{x} + c \tag{133}$$

with

$$\begin{aligned}
\mathbf{A} &= -\frac{1}{2}\left(\mathbf{\Sigma}_1^{-1} - \mathbf{\Sigma}_2^{-1}\right) && (134)\\
\mathbf{b} &= \mathbf{\Sigma}_1^{-1}\bar{\mathbf{x}}_1 - \mathbf{\Sigma}_2^{-1}\bar{\mathbf{x}}_2 && (135)\\
c &= -\frac{1}{2}\left(\log_e\frac{|\mathbf{\Sigma}_1|}{|\mathbf{\Sigma}_2|} + \bar{\mathbf{x}}_1^T\mathbf{\Sigma}_1^{-1}\bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_2^T\mathbf{\Sigma}_2^{-1}\bar{\mathbf{x}}_2\right) - \log_e\frac{\pi_1}{\pi_2} && (136)
\end{aligned}$$

with the classification rule being

$$\text{if}\, Q(\mathbf{x}) > 0\, \text{assign}\ \mathbf{x}\ \text{to}\ \Pi_1 \tag{137}$$

else assign $\mathbf{x}$ to $\Pi_2$. This is known as the *Quadratic discriminant*.

So far, our analysis has been restricted to binary classification. Many problems involve more than two classes so an extension to multiple classes is useful, although the algebra is significantly more complex. The general argument is that we compute the pairwise relative probabilities as before and form the discriminant

$$L_{ij}(\mathbf{x}) = \log_e\left(\frac{P(\Pi_i|\mathbf{x})}{P(\Pi_j|\mathbf{x}}\right) = \log_e\left(\frac{f_i(\mathbf{x})\pi_i}{f_j(\mathbf{x})\pi_j}\right) \tag{138}$$

and then $\mathbf{x}$ is assigned to $\Pi_i$ if $L_{IJ} > 0$ for all $j \neq i$. From this it follows from the earlier argument that the discriminant function is

$$L_{ij}(\mathbf{x}) = \mathbf{m}_{ij}^{\mathrm{T}}\mathbf{x} + c_{ij} \tag{139}$$

with

$$\mathbf{m}_{ij} = \left(\bar{\mathbf{x}}_i - \bar{\mathbf{x}}_j\right)^{\mathrm{T}} \mathbf{\Sigma}^{-1} \text{ and} \tag{140}$$

$$c_{ij} = -\frac{1}{2}\left(\bar{\mathbf{x}}_i^{\mathrm{T}}\mathbf{\Sigma}^{-1}\bar{\mathbf{x}}_i - \bar{\mathbf{x}}_j^{\mathrm{T}}\mathbf{\Sigma}^{-1}\bar{\mathbf{x}}_j\right) + \log_e \frac{\pi_i}{\pi_j}. \tag{141}$$

One key point to note about this is we are drawing boundaries in space and it is not obvious that these are necessarily consistent with each other. The obviously problematic situation is shown in Figure 37, where in the shaded gray region in the middle, $P1 < P3$, $P3 < P2$, and $P3 < P1$ and we cannot resolve the class (1, 2, 3) in that region. However, a moment's thought explains why this situation is impossible. The class boundaries have been constructed from the lines $P1 = P2$, $P1 = P3$, and $P2 = P3$ and therefore they *must* meet at a single point where $P1 = P2 = P3$. The division of space is self-consistent *by design*.

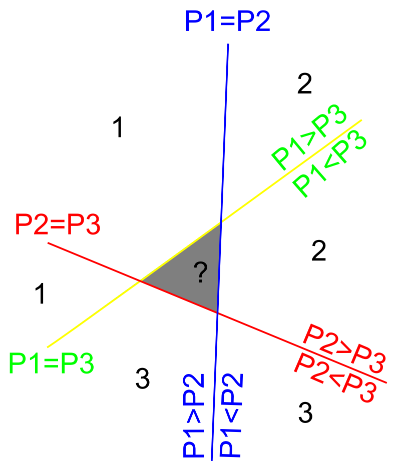

Figure 37: Decision boundaries in multiclass LDA. The shaded gray region in the middle is impossible by construction.

## *LDA on MNIST*

Let us see how LDA performs on MNIST digit classification. This is a ten-class problem and we will therefore need to use multiclass LDA. LDA requires that we model the distribution of data within each class as a Gaussian, and that the covariance of each class distribution is the same. We will not attempt to determine whether this is true for this problem, because this is, in general, a very difficult thing to do. Instead, we assume that this is the case and see how well we can do. Note that this is extremely common practice, although nor endorsed. As we will see, it can work very well even if the assumptions are not met.

Our model for the distributions of the data in each of the classes is the multivarite normal distribution (Equation (126)). First, we note that the prefactor is the same in all cases if we assume the same covariance for each class, so we can ignore this because we will be taking ratios of the class distributions (Equation (138)). Secondly, we need to compute the mean of each class $\{\bar{\mathbf{x}}_i\}_{i=0}^{9}$. This is a straighforward calculation that we do by taking the mean of each pixel for the images in that class. Thirdly, we need to compute the covariance that we assume to be the same for each class. With this, we have to be careful. It is tempting to simply compute the covariance of the whole dataset, but this is not the correct thing to do: the covariance of the dataset is not the same as the class covariance because the classes are in different locations in the image space. We have to align the means of the classes before we compute the covariance over the whole dataset, otherwise the covariance matrix will be modelling inter-class rather than intra-class variations in the data. In practice, this means that we substract the class mean from every point in the dataset, and then compute the covariance, before moving the points back to their original position.

A further issue arises in the MNIST problem because of the non-varying pixels in the data. These lead to entire rows/columns of the covariance matrix being all-zero and are therefore not linearly independent. This means that the necessary matrix inversion cannot be performed. We therefore have to remove these from the data before computing the covariance.

Finally, we compute the class priors $\{\pi_i\}_{i=0}^{9}$ by computing the proportion of the training set that belongs to each class. This gives us all of the information we need.

We train the model by computing the class means, the covariance, and the class priors, from the training data. From this, we can compute the class separation boundaries. In practice, though, it is not necessary to compute these explicitly. Instead, we can classify a new data point $\mathbf{x}$ by simply computing $\arg\max_i f_i(\mathbf{x})\pi_i$.

The results of applying this to the MNIST test set as shown in Figure 38. The overall classification accuracy is 83%. This might be regarded as impressive given tha naïvety of the approach we have taken here. There is no guarantee that the data obeys LDAs assumptions and we have sought no assurances that LDA will perform well on this data. Furthermore, by comparison with $k$-nn, LDA is computationally very efficient. Whilst there is no training in $k$-nn, inference has time complexity $\mathcal{O}(N^2)$, where $N$ is the number of samples (pairwise distance have to be computed between all pairs of points). In LDA, training and inference both have time complexity $\mathcal{O}(NM)$ (where $M$ is the number of classes) – a huge saving in this case where $M \ll N$.

## *Generative Properties of LDA*

The basis of LDA is that we "fit" a normal distribution to each class of data and use the lines of equality between the classes to define the boundaries between them for classification purposes. Because we have learned the distribution of the data in the form of the class conditional distribution, we are able to resample from the distributions and *generate new samples*. This can potentially be useful in situations where data is limited, but is increasingly being used as an end in itself to generate realistic synthetic examples. The state of the art in this area uses generative adversarial networks (GANs), a technique in which a generator and discriminator (real/not real) are trained against each (adversarially) such the the generator tried to fool the discriminator that its samples are real, and the discriminator tries to guess whether the sample it sees is real or not. By co-training the discriminator and generator, both can simulataneously improve which leads to vastly improved quality of generated samples. The best current approach to this is "BigGAN" from Google Brain, described at `https://arxiv.org/abs/1809.11096`.

Our model is nowhere near as sophisticated as this: we have

| T \ P | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.96 | 0.00 | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 | 0.00 | 0.01 | 0.00 |
| 1 | 0.00 | 0.96 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.02 | 0.00 |
| 2 | 0.09 | 0.03 | 0.73 | 0.03 | 0.02 | 0.00 | 0.03 | 0.01 | 0.05 | 0.00 |
| 3 | 0.07 | 0.00 | 0.02 | 0.82 | 0.00 | 0.02 | 0.00 | 0.01 | 0.03 | 0.01 |
| 4 | 0.03 | 0.01 | 0.01 | 0.00 | 0.88 | 0.00 | 0.01 | 0.00 | 0.01 | 0.05 |
| 5 | 0.05 | 0.01 | 0.00 | 0.04 | 0.01 | 0.79 | 0.02 | 0.01 | 0.04 | 0.02 |
| 6 | 0.12 | 0.01 | 0.01 | 0.00 | 0.02 | 0.03 | 0.79 | 0.00 | 0.02 | 0.00 |
| 7 | 0.09 | 0.03 | 0.01 | 0.01 | 0.02 | 0.00 | 0.00 | 0.77 | 0.00 | 0.07 |
| 8 | 0.04 | 0.03 | 0.01 | 0.03 | 0.02 | 0.04 | 0.01 | 0.00 | 0.80 | 0.03 |
| 9 | 0.04 | 0.01 | 0.00 | 0.01 | 0.06 | 0.00 | 0.00 | 0.04 | 0.01 | 0.83 |

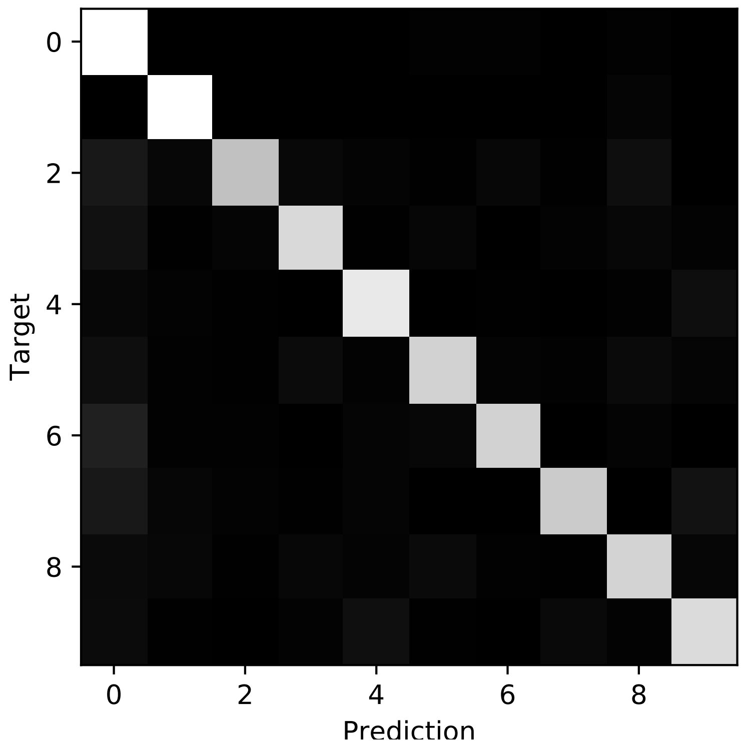

Figure 38: Confusion table and matrix for multiclass LDA on the MNIST dataset.

assumed that all classes are normally distributed and have the same covariance, and will simply resample from the estimated distributions. Ten generated samples from each class are shown in Figure 39. Some of these samples are identifiable, others are less so. This is far from the state-of-the-art, but the distributions are clearly capturing the essence of the data, although certainly not the details.

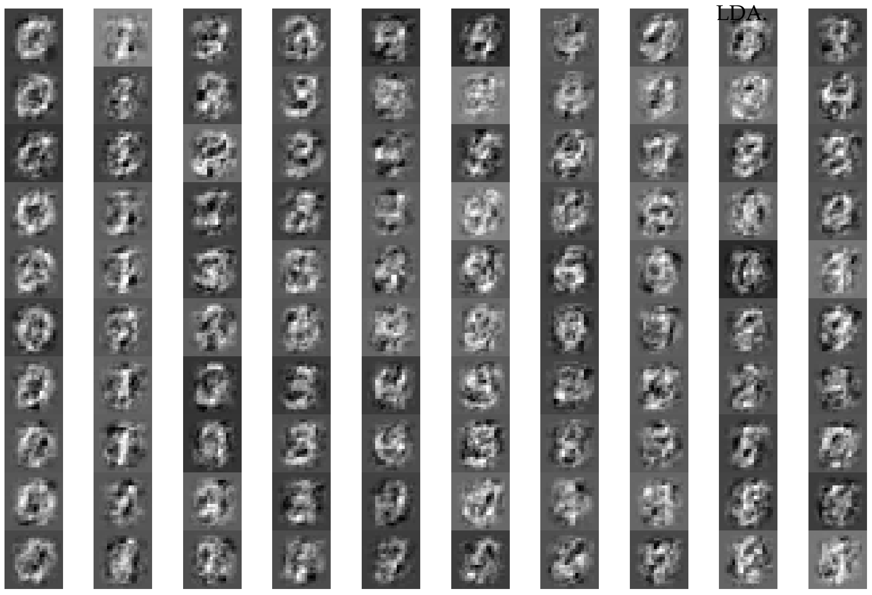
Figure 39: Ten examples of generated samples from MNIST, using the distributions learned by multiclass LDA.

## *Logistic Regression*

In LDA, we determine the decision boundary by explicitly modelling the distribution of each class. We showed that this can be an effective classification method, even if the assumptions of the underlying model (Gaussian class-conditional likelihoods with equal covariance) are not met. We will look at an alternative probabilistic approach to classification that does not rely on explicit modelling of the likelihood: *logistic regression*. As the name suggests, this is related to the regression problem for continuous variables that we studied earlier in the module.

We will formulate logistic regression initially for two-class binary classification. The key idea is that we will model the probability (actually the odds) of an observation being in a class as a linear combination of the independent variables. We will model binary classification as a problem with two possible output values $Y = \{0,1\}$.

The key quantity that we will work with is *odds*, defined as the ratio of the probabilities of the two possible outcomes. If we denote $p_i(\mathbf{x}) = P(\Pi_i|\mathbf{x})$, then the odds that the measurement is in class $\Pi_1$ is $o_1 = p_1/p_0 = p_1/(1-p_1)$ since we have a two-class problem. The logarithm of the odds – the so-called *logit* is then

$$\text{logit}(p_1) = \ln\left(\frac{p_1}{1-p_1}\right) \tag{142}$$

We might reasonably ask why this is a sensible thing to do. The posterior probabilities $P(\Pi_i|\mathbf{x})$ are, of course, limited to the range $[0,1]$. If we wish to formulate the problem in regression terms, which allow us to avoid the assumption of LDA, then we need to map this to $[-\infty,\infty]$. This is what the logit function does, as shown in Figure 40.

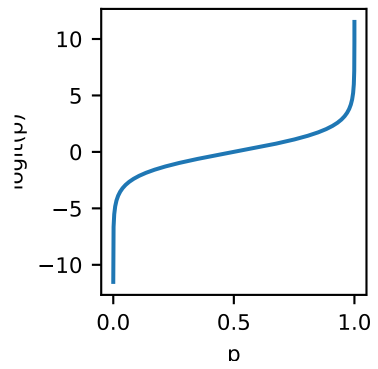

Figure 40: The logit function maps $[0,1]$ to $[-\infty,\infty]$.

Now that we have transformed the probability to a continuous variable in $\mathbb{R}$, we can model it as a regression problem. We write

$$\text{logit}(p_1) = \ln\left(\frac{p_1}{1-p_1}\right) = w_0 + w_1 x_1 + \cdots + w_n x_n = \mathbf{w}^T\mathbf{x} \tag{143}$$

where $\mathbf{x} = 1, x_1, x_2, \ldots, x_M$ are the independent variables. Taking the exponential of both side and rearranging, it is straighforward to show that

$$p_1 = \frac{\exp(\mathbf{w}^T\mathbf{x})}{1+\exp(\mathbf{w}^T\mathbf{x})} \quad \text{and} \quad p_0 = 1 - p_1 = \frac{1}{1+\exp(\mathbf{w}^T\mathbf{x})} \tag{144}$$

The probability $p_1$ is therefore conditioned on $\mathbf{x}$ and $\mathbf{w}$ so we will write this as $p_1(\mathbf{x},\mathbf{w})$.

We now formulate the likelihood over a set of observation to train the model. Given $N$ obervation $\{\mathbf{x}_i, y_i\}_{i=1}^N$, the overall likelihood is the product of the likelihoods of the individual observations. Noting that the observations $y_i$ are binary ($\{0,1\}$), we use the observations themselves to select the appropriate likelihood for each observation:

$$\mathcal{L}(\mathbf{w}) = \prod_{i=1}^{N} p_1(\mathbf{x},\mathbf{w})^{y_i} p_0(\mathbf{x},\mathbf{w})^{1-y_i} \tag{145}$$

$$= \prod_{i=1}^{N} p_1(\mathbf{x},\mathbf{w})^{y_i} \left(1 - p_1(\mathbf{x},\mathbf{w})\right)^{1-y_i} \tag{146}$$

As we did in probability formulation of regression, we choose $\mathbf{w}$ by maximising the likelihood, or equivalently the log-likelihood which is

$$\ln \mathcal{L}(\mathbf{w}) = \ln(\mathcal{L}(\mathbf{w})) = \sum_{i=1}^{N} y_i \ln(p_1(\mathbf{x},\mathbf{w})) + (1-y_i)\ln(1-p_1(\mathbf{x},\mathbf{w})) \tag{147}$$

$$= \sum_{i=1}^{N} y_i \left[\ln(p_1(\mathbf{x},\mathbf{w})) - \ln(1-p_1(\mathbf{x},\mathbf{w}))\right] + \ln(1-p_1(\mathbf{x},\mathbf{w})) \tag{148}$$

$$= \sum_{i=1}^{N} y_i \ln \frac{p_1(\mathbf{x},\mathbf{w}))}{1-p_1(\mathbf{x},\mathbf{w})} + \ln(1-p_1(\mathbf{x},\mathbf{w})) \tag{149}$$

$$= \sum_{i=1}^{N} y_i \mathbf{w}^{\mathrm{T}}\mathbf{x} - \ln(1+\exp(\mathbf{w}^{\mathrm{T}}\mathbf{x})). \tag{150}$$

The optical weights $\mathbf{w}^*$ that maxmimise the likelihood are therefore given by

$$\mathbf{w}^* = \underset{\mathbf{w}}{\arg\max} \left[\sum_{i=1}^{N} y_i \mathbf{w}^{\mathrm{T}}\mathbf{x} - \ln(1+\exp(\mathbf{w}^{\mathrm{T}}\mathbf{x}))\right] \tag{151}$$

This is as far as we can get, and this quantity can't, in general, be maximised analytically to provide a closed-form solution. The normal method for solving this is to use an iterative method of solution called IRLS – Iterative Reweighted Least Squares – but a detailed explanation of this is beyond the scope of this course. The interested reader should refer to Section 8.3 of Izenman[5].

[5] Izenman, Multivariate Statistical Analysis, Springer (New York) 2013

Given the optimal weight vector $\mathbf{w}^*$, we are then in a position to compute the decision boundary, with reference to Equation (143). A data point $\mathbf{x}$ should be assigned to class 1 if $p_1 > 1 - p_1$, which is when $\textbf{logit}(p_1) > 0$. The decision rule is therefore

$$\mathbf{w}^{*\mathrm{T}}\mathbf{x} > 0 \quad \rightarrow \quad \mathbf{x} \in \Pi_1$$

Alternatively, we can compute the probabilities directly, recalling that

$$p_1 = \frac{\exp(\mathbf{w}^{*\mathrm{T}}\mathbf{x})}{1+\exp(\mathbf{w}^{*\mathrm{T}}\mathbf{x})} \quad \text{and} \quad p_0 = \frac{1}{1+\exp(\mathbf{w}^{*\mathrm{T}}\mathbf{x})} \tag{152}$$

and then $\mathbf{x}$ is assigned to the class with the higher probability.

It is useful to draw some comparisons between logistic regression (LR) and LDA. Both are formulated from a statistical perspective, but there are some key differences that have been written about in the literature.

1. The statistical assumptions are much more relaxed in LR as compare to LDA. In particular there is no assumption that the likelihoods are are multivariate Gaussian. In principle, this make LR more robust to non-normality than LDA.

2. There is no assumption that the class distributions have the same covariance.

3. LR is much less efficient than LDA for large sample sizes.

4. LR can require larger dataset sizes to work well.

The generalisation of LR to multiple classes is straighforward. The logit $L_i = \ln(\frac{p_i}{1-p_i}) = \mathbf{w}_i * *\mathrm{T}\mathbf{x}$ is computed for every class boundary $i$, and the class with the highest probability is selected. One common way to do this is to choose a single reference class as a "pivot" and compute all boundaries against that class, choosing the one with the highest probability. Given $M$ classes, we compute the logit of every other class against class M, ie, we compute the logit as $\ln(\frac{p_i}{p_M})$ for all $i$.

$$\ln \frac{p_1}{p_M} = \mathbf{w}_1^{*\mathrm{T}}\mathbf{x} \tag{153}$$

$$\ln \frac{p_2}{p_M} = \mathbf{w}_2^{*\mathrm{T}}\mathbf{x} \tag{154}$$

$$\dots \tag{155}$$

$$\ln \frac{p_{M-1}}{p_M} = \mathbf{w}_{M-1}^{*\mathrm{T}}\mathbf{x} \tag{156}$$

$$\tag{157}$$

Exponentiating, we have

$$p_i = p_M \exp(\mathbf{w}_i^{*\mathrm{T}}\mathbf{x}) \quad \text{for i=\{1,2,...,M-1\}} \tag{158}$$

and because all probabilities must add to one we know that

$$\sum_{i=1}^{M} p_i = 1 \quad \rightarrow \quad p_M = 1 - \sum_{i=1}^{M-1} p_M \exp(\mathbf{w}_i^{*\mathrm{T}}\mathbf{x}) \tag{159}$$

and therefore

$$p_M = \frac{1}{1 + \sum_{i=1}^{M-1} \exp(\mathbf{w}_i^{*\mathrm{T}}\mathbf{x})}. \tag{160}$$

Finally, we substitute this into Equation (158) to obtain

$$p_i = p_M \exp(\mathbf{w}_i^{*\mathrm{T}}\mathbf{x}) = \frac{\exp(\mathbf{w}_i^{*\mathrm{T}}\mathbf{x})}{1 + \sum_{i=1}^{M-1} \exp(\mathbf{w}_i^{*\mathrm{T}}\mathbf{x})} \tag{161}$$

Thus we perform multiple binary LRs of each class against the pivot class $\Pi_M$ to find the parameters $\mathbf{w}_i^*$ for each class $i$, and assign $\mathbf{x}$ to the class with the highest probability $p_i$. As before, these regressions cannot be solved in closed-form and we must do this numerically with IRLS.

LR is implemented in most machine learning libraries. In the accompanying Jupyer Notebook, we use the version implemented in `scikit-learn` to run LR on the MNIST dataset. This can be found at `https://colab.research.google.com/drive/1-vpNgx3PtdyRGv1pC0PYrR-X-vdf8jJT`.

# *Unsupervised Learning*

## *Introduction*

Supervised learning – be that regression of classification – is by farthe best developed form of machine learning and has been able to achieve extraordinary results when the combination of sophisticated methods, high-performance computing, and large annotated datasets are brought together. However, an area that is at least equally important (and will perhaps prove to be far more important) is that ow how to extract structure from data where we do not have any *a priori* knowledge of the data (i.e. no labels or annotations), and our task is to try to identify classes in the data without knowing what they are. In the conext of MNIST, can we "discover" that there are ten classes of digit purely from the data?

By far the most common approach to this problem is the task of *clustering*, in which we try to find groups of points that are in some way similar to other members of the same group, but different to members of other groups. In this section, we will examine a few different ways of finding clusters and the assumptions behind them.

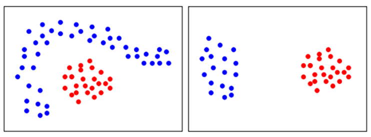
Figure 41: Some simple clusters

There are many hundreds of clustering algorithms, all with different underlying assumptions about the nature of the data. For example, some methods model the data as a probability distribution of a specific form; some are based on computing the local density of points; other are based on constructing a hierarchical representation of the relationships between the points in the dataset. We will consider three approaches:

- A "vector quantisation" approach, in which we try to find a set of prototype vectors, each of which is considered to be typical of a cluster.
- An agglomerative approach, in which points are grouped together hierarchically.
- A mixture modelling approach, in which the data is represented as a mixture of probability density functions.

## *The k-means Algorithm*

Perhaps the simplest and most commonly used clustering technique is the *k*-means algorithm. It is a so-called vector quantisation technique that learns a prototype vector for each class in the data, and then assigns all data points to their nearest prototype. The data space is thus partitioned into cells, one per prototype, in what is known as a *Voronoi Tesselation* of the space. The prototype vectors are often known as the cluster *centroids*. The partioning of space is somewhat similar to that of LDA, except that there are no labels from which we can learn a distribution: $k$-means does no explicitly model the distribution of the data.

This technique aims to learn a set of partition $N$ data points into $k$ clusters such that each data point belongs to the cluster that has the nearest centroid. This amounts to a partitioning of the space by lines that bisect those connecting the means (Figure 42). This is somewhat similar to LDA, and leads to a similar partitioning of space, but note that $k$-means does not explicitly model the distribution of the data. The clusters are defined by the allocation of points to regions.

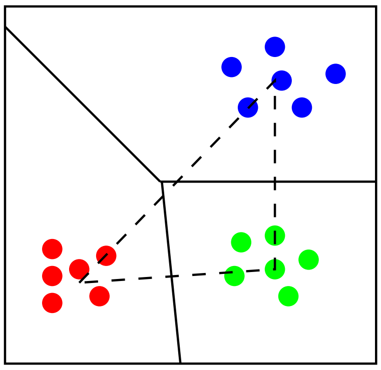
Figure 42: Partitioning of data in the $k$-means algorithm.

The $k$-means algorithm can be described in rather non-precise terms as follows:

1. First, assign every data point to a clusters, at random.
2. Compute the centroid (in the data space) of each cluster by computing the average over its assigned members.
3. For each data point, calculate to which of the cluster centroids it is closest and re-assign the point to that cluster.
4. Return to step 2 and repeat until the clusters are stable.

A formal description of $k$-means can be found in Algorithm 5. This process is illustrated pictorially in Figure 43. There are a few points to note:

- The initial random seeding of the process means that you should perform multiple repeats in order to ensure stability of convergence.
- $k$-means is a *parametric* method: the number of clusters to be found has specified. It is is therefore typical to run the algorithm several times with different values of $k$ and to try to determine which value is optimal. There is no established single method for doing this, it depends very much on the nature of the problem and what is already known about it. If $k$ is too large, single clusters will be split; to small and clusters that should be separate will be combined.
- This is a computationally "hard" algorithm: $\mathcal{O}(kN)$. The number of distance calculations increases rapidly with the number of clusters and the number of data points so it can be very costly for large datasets with large numbers of clusters.
- Although we have used Euclidean distance in the examples, $k$-means can equally be used with other distance metrics.

## *k-means on MNIST*

The application of $k$-means to the MNIST dataset can be found in the accompanying Jupter Notebook at `https://colab.research.`

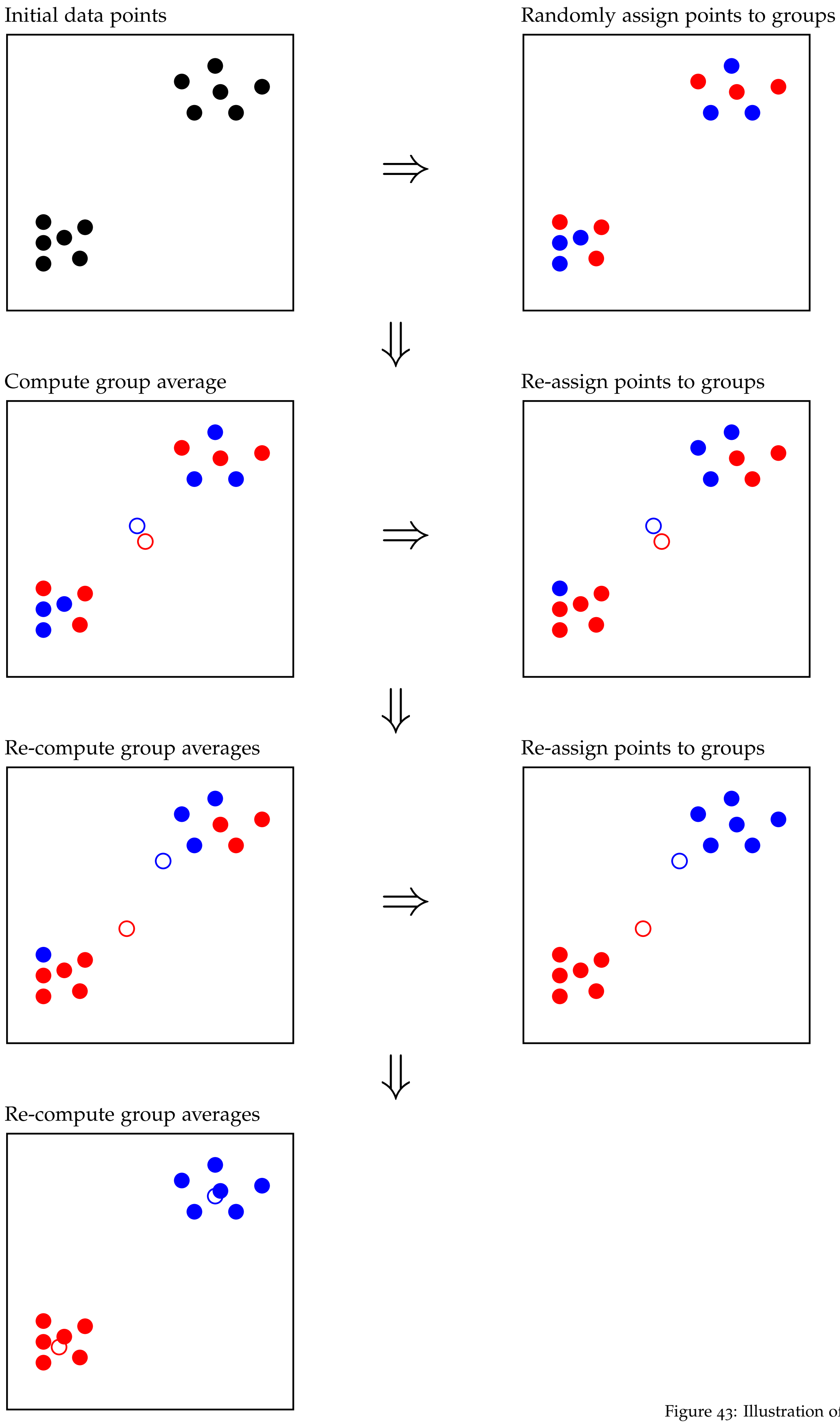
Initial data points
Randomly assign points to groups
Compute group average
Re-assign points to groups
Re-compute group averages
Re-assign points to groups
Re-compute group averages

Figure 43: Illustration of the *k*-means algorithm.

`google.com/drive/1vacPSG5JSi_Rm3g67UPbjvE5qLqsGIjm`.

## *Agglomerative Hierarchical Clustering*

The *k*-means algorithms works by subdividing the whole dataset into "flat" subgroups with no internal structure, and if a different number of clusters is required, the algorithm must be re-run. Whilst this is a very popular approach, it is quite limited in terms of the types of clusters it can find: the strict convex partitioning of space it produces is not appropriate for many types of data. A popular approach that allows some of these problems to be overcome is to build up (agglomerate) clusters in a hierarchical manner, in which we build up a hierarchy of relationships between the data points. At run-time, this is a *non-parametric* approach: all point-point relationships are evaluated, and the resulting hierarchy analysed post-cluster to interpret the data's structure.

In this approach, we start by considering the individual data points. We calculate the separation between every pair of points, and group together the pair that are closest. We then remove that pair of points from the list of points, and replace it with the grouped pair. In the next iteration, all distances are again calculated, but this time we must compute the distance to the pairing computed in the first round. In order to do this, we need a method for calculating the distance between two groups of points. This is known as the *linkage* strategy and there are many approaches to this. Three of the most widely used (illustated in Figure 44) are:

- Single linkage: between the nearest points in the two groups.
- Average linkage: between the centroids of the groups.
- Complete linkage: between the farthest points in the two groups.

```
Data: Data points $Data[N]$; Number of clusters $K$
Result: Cluster assignments $ClusterIndex[N]$, cluster centroids
        $Centroid[N]$
% Randomly assign each point to a cluster
for $i \leftarrow 1$ to $N$ do
    $ClusterIndex[i] \leftarrow randint(1, K)$
end
% Calculate the centroid of each cluster as the mean of
  its members
for $i \leftarrow 1$ to $K$ do
    $Centroid[i] \leftarrow mean(Data[Cluster[i]])$
end
% Iterate until clusters are stable
$Converged \leftarrow False$;
while $Converged == False$ do
    % Compute distance from every point to each cluster
      mean
    for $i \leftarrow 1$ to $K$ do
        for $j \leftarrow 1$ to $N$ do
            $dist[i][j] \leftarrow distance(Centroid[i], Data[j])$
        end
    end
    % Find the nearest mean for each data point
    for $i \leftarrow 1$ to $N$ do
        for $j \leftarrow 1$ to $K$ do
            $NewClusterIndex[i] = nearest(Data[i], Centroid[j])$
        end
    end
    % Generate new clusters
    for $j \leftarrow 1$ to $K$ do
        $NewCentroid[i] \leftarrow mean(Data[NewClusterIndex == i])$
    end
    % Check to see if cluster membership has changed
    $Converged \leftarrow True$
    if $NewClusterIndex$ != $ClusterIndex$ then
        $Converged \leftarrow False$
    end
    % Replace old clusters with new ones
    $ClusterIndex \leftarrow NewClusterIndex$ $Centroid \leftarrow NewCentroid$
end
```

**Algorithm 5**: The *k*-Means Algorithm.

Each of these gives a notably different result. In general, single linkage permits the formation of clusters which are extended in space, because it takes no account of the size of the existing cluster, whilst complete linkage favours more compact clusters. The choice depends on the nature of the dataset.

Once we have selected a linkage strategy, we can proceed to iteratively and hierarchically group points and groups of points together. An example of this is shown in Figure 45 and it is instructive to work through it.

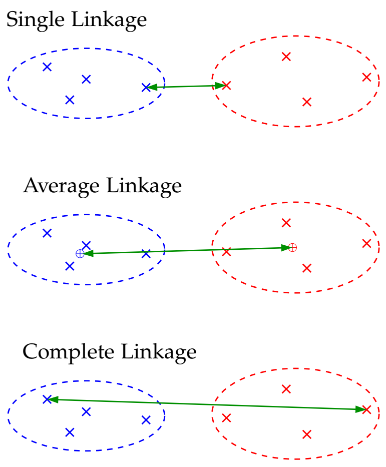

Figure 44: Linkage strategies for clustering of groups.

$$\begin{aligned}
\text{List of points} &\mapsto 0,1,2,3,4,5,6 \\
\text{Group 1 with 2} &\mapsto 0,3,4,5,6,(1,2) \\
\text{Group 3 with 4} &\mapsto 0,5,6,(1,2),(3,4) \\
\text{Group 0 with (1,2)} &\mapsto 5,6,(3,4),(0,(1,2)) \\
\text{Group 5 with 6} &\mapsto (3,4),(0,(1,2)),(5,6) \\
\text{Group (3,4) with (0,(1,2))} &\mapsto (5,6),((3,4),(0,(1,2))) \\
\text{Group (5,6) with ((3,4),(0,(1,2)))} &\mapsto ((5,6),((3,4),(0,(1,2))))
\end{aligned}$$

The most common way to represent this is with the **dendrogram** shown in Figure 45. The height of each "junction" in the dendrogram represents the distance between the points being paired (so $(3,4)$ join at height 1, for example).

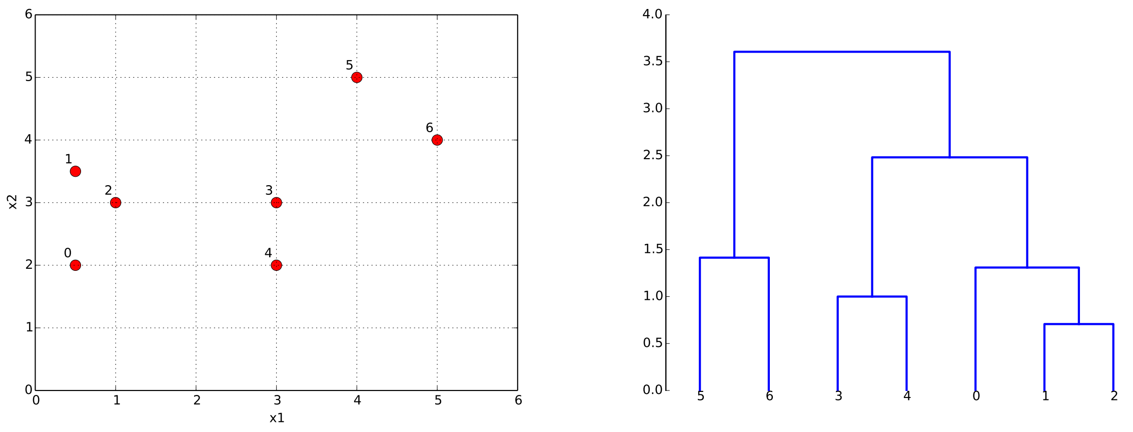

Figure 45: Datapoints for Hierachical Clustering and the resulting dendrogram using euclidean distance with average linkage.

One of the advantages of this method is that it generates all numbers of clusters in one pass. By cutting horizontally across the dendrogram at different heights we can obtain the desired number of clusters just by choosing how many lines we cut. The disadvantage is that it can be very time consuming to compute the dendrogram: if there are a large number of sample points, and pairwise distance calulations have to be done between all of them in the first instance, then this very quickly becomes computationally expensive. Compare this to $k$-means, where the distance calulation is between the $N$ points and the $K$ cluster centres: $N \times K$ evaluations of the distance function vs the $N(N-1)/2$ calculations needed just to compute the first pairing in hierarchical clustering.

### *Hierarchical Clustering on MNIST*

The application of hierarchical clustering to the MNIST dataset can be found in the accompanying Jupter Notebook at `https://colab.research.google.com/drive/1vacPSG5JSi_Rm3g67UPbjvE5qLqsGIjm`.

## *Gaussian Mixture Models*

*k*-means and agglomerative hierarchical clustering are heuristic approaches to clustering that have at best weak roots in statistics. They are also "hard" clustering techniques in the sense that they give a single cluster assignment to each sample. Perhaps advantageously, they make no strong statistical assumptions about the data, but the lack of an underpinning data model means that their is little to no nuance in their prediction. One way to overcome this is to introduce a statistical prior in a similar way to how it is done in LDA. In *Gaussian Mixture Modelling* (GMM), we make an assumption that the data is drawn from a distribution that can be modelled by a finite number of Gaussians, each with a mean and variance that has to be learned from the data in the absence of labels. With this model, it will be possible to predict which of the mixture components a data point is most likely to belong to.

The starting point for a GMM is its underlying model. We will seek to explain the data by assuming it is drawn from a probability density function of the form

$$p(\mathbf{x}) = \sum_{k=1}^{K} A_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{162}$$

The $A_k$ are the *mixing coefficients* of each Gaussian density *component* $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$. What we aim to do in GMM is to

This distribution is a mixture of Gaussian distributions. In GMM, we have to perform essentially two tasks:

1. Learn the parameters of the GMM which best describe the data.
2. Determine from which component a data point is most likely to have been generated.

That is, we assume that each component is the probability density function for an indepedent data generating process. Our goal is to find which process generated a given data point.

To see how we can do this, let us first make some observations about Equation (162). First, we note that each Gaussian component is, by definition normalised. Since $p(\mathbf{x})$ is also a probability density function, we can show that this means that

$$\sum_{k=1}^{K} A_k = 1 \tag{163}$$

It can also be shown that $0 \leq A_k \leq 1$ by noting that $p(\mathbf{x})$ and $\mathcal{N}(\cdot)$ must always be $\geq 0$, and the mixing coefficients therefore can be interpreted as probabilities, since they satisfy the requirements of probability.

We then observe that using the sum and product rules of probability, we can write

$$p(\mathbf{x}) = \sum_{k=1}^{K} p(k)p(\mathbf{x}|k) \tag{164}$$

and the symmetry between this expression and Equation (162) is clear: we can interpret the $A_k = p(k)$ as being the prior probability of choosing a point from component $k$, and $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) = p(\mathbf{x}|k)$ are the class-conditional likelihoods.

We can finally, using Bayes' theorem compute $p(k|\mathbf{x})$, which is know as the *responsibility* of class $k$ for point $\mathbf{x}$, i.e probability that class $k$ "explains" point $\mathbf{x}$:

$$\begin{aligned} r_k(\mathbf{x}) &= p(k|\mathbf{x}) && (165) \\ &= \frac{p(k)p(\mathbf{x}|k)}{\sum_{k'=1}^{K} p(k')p(\mathbf{x}|k')} && (166) \\ &= \frac{A_k\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)}{\sum_{k'=1}^{K} A_{k'}\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_{k'}, \boldsymbol{\Sigma}_{k'})} && (167) \end{aligned}$$

The responsibilities $r_k(\mathbf{x})$ can be viewed as the probability that component $k$ explains $\mathbf{x}$. GMM is therefore a "soft" clustering technique: it predicts the probability with which a data point $\mathbf{x}$ belongs to each of the $K$ classes.

The one remaining question is how we learn the components $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$. This is not a trivial undertaking. One way to do this would be to maxmimise the likelihood as we have done in the past. This is given by the joint distribution over a set of data points

$$p(\mathbf{X}|\mathbf{A}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \prod_{i=1}^{N}\sum_{k=1}^{K} A_k\mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k). \tag{168}$$

Even taking the log does not help us to maxmimise this. We have to solve this numerically. This can be done via some non-linear optimisation process, or using the technique of *expectation maxmisation*, a sophisticated technique that is beyond the scope of this course. We will assume that we have access to a technique for maximising the likelihood.

Let us study a toy problem with this method. Let us create an artificial data generating process with three Gaussian components. These are shown in Figure 46, together with their parameters. The reds dots show 100 data points sampled from the distribution in the proportion given by the values of $A_k$.

We use `sklearn.mixture.GaussianMixture` with expectation maxmisation to learn the parameters of Equation (162). The results are shown in Figure 47.

| $k$ | $A_k$ | $\mu_k$ | $\sigma_k$ |
|---|---|---|---|
| 1 | 0.3 | -2.0 | 1.0 |
| 2 | 0.4 | 1.0 | 0.7 |
| 3 | 0.3 | 4.0 | 0.8 |

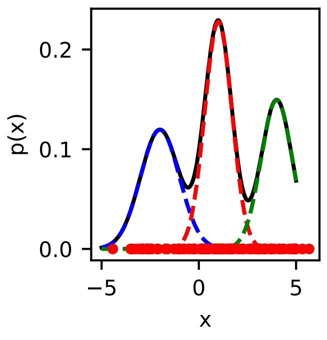

Figure 46: Components of an GMM.

| $k$ | $A_k$ | $\mu_k$ | $\sigma_k$ |
|---|---|---|---|
| 1 | 0.26 | -2.26 | 0.95 |

Cluster assignments can then be made by computing which component has the highest responsibility. We can map the responsibilities out over the range of value of $x$ by evaluating Equation (167) as shown in Figure 48. The convention is to assign point $x$ to the component with the largest responsibility, but we can see from this that more nuanced assignments can be made by noting that the responsibilities are probabilities. There are some clear regions of $x$, where there is significant uncertainty over the cluster assignment because the responsibilities are not binary. This provides a much more nuanced view than hard clusterings such as that produced by $k$-means.

A demonstration of this can be found at `https://colab.research.google.com/drive/1Sk1bvgcT-yShZefMYfK7Nfr2vI9o94ei`.

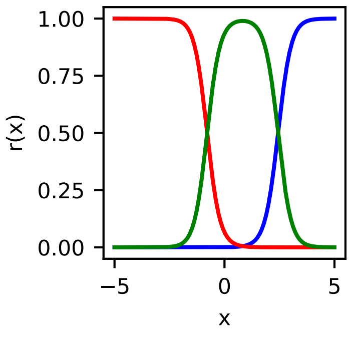

Figure 48: Responsibilities $r_k(x)$of the three classes (1: red, 2: green, 3: blue) in our GMM.

# *Ensemble Methods*

## *Introduction*

It will sometimes be the case that no single classification method – no matter how sophisticated – performs well on a dataset of interest. In these situations it is sometimes that case that multiple learners can be combined to give a better prediction than any single learning method. We will consider two ways in which this can be acheived:

- **Boosting**, in which a number of weak learning methods are combined to produce a single strong learner.
- **Bagging**, in which the training set is divided randomly and the results averaged over the subsets.

We will look at the general principles of these two approaches and consider a specific example of each: the *AdaBoost* algorithm as an examples of boosting, and the *Random Forest* method as an example of bagging.

## *Boosting*

Boosting is a technique that allows a number of weak classification methods (those that perform only a little better than chance) to be combined in such a way that their combination is a strong (accurate) classifier. The boosting method is an extension of a simple *committee machine* (in which one simple takes the majority vote of a number of classifiers), with the modification that in boosting, the classifiers are trained sequentially with incorrectly classified data points reweighted with each iteration to encourage later classifiers to correctly classify those points.

We will present the core ideas behind boosting in the context of the AdaBoost – $\underline{\text{Ada}}$ptive $\underline{\text{Boost}}$ing – algorithm. This is frequently referred to as the best "out of the box" classifier.

The starting point for AdaBoost is the dataset. We will consider this initially as a binary classification task for simplicity, in which we have

- A set of $N$ data points $\mathcal{D} = \{\mathbf{x}_i, t_i\}_{i=1}^{N}$ with binary target variables $t_i = \{-1, 1\}$.

- A set of $M$ learners $\{y_i(\mathbf{x})\}_{i=1}^{M}$
- A set of $N$ weights for each learner, $\left\{\mathbf{w}^{(i)}\right\}_{i=1}^{M}$

These weights are the key to AdaBoost. They are not the model weights that we have been working with (these are not visible to us directly in AdaBoost), they are weights of the data points, and they will be used to help adjust the importance of each data point. The algorithm processed as follows:

1. Initialise the weights $w_i^{(1)} = \frac{1}{N}$ for all $i$.
2. For $m = 1 \to M$
   (a) Fit classifier $y_m(\mathbf{x})$ to the training data. This is done by minimising the loss function
   $$\mathcal{L}_m = \sum_{n=1}^{N} w_n^{(m)} I(y_m(\mathbf{x}_n) - t_n) \tag{169}$$
   where the *indicator function* $I(y_m(\mathbf{x}_n) - t_n) = 1$ when $y_m(\mathbf{x}_n) - t_n \neq 0$ and $0$ otherwise.
   (b) Compute
   $$\epsilon_m = \frac{\mathcal{L}_m}{\sum_{n=1}^{N} w_n^{(m)}} \tag{170}$$
   and
   $$\alpha_m = \ln \frac{1-\epsilon_m}{\epsilon_m} \tag{171}$$
   (c) Update the weights by
   $$w_n^{(m+1)} = w_n^{(m)} \exp\left\{\alpha_m I(y_m(\mathbf{x}_n) - t_n)\right\} \tag{172}$$
3. Predictions are made using a model averaged over all $M$ classifiers:
   $$Y(\mathbf{x}) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m y_m(\mathbf{x})\right) \tag{173}$$

Let us explain the key stages of this method.

In step 1, the initial weights of the data points are set to be equal. This means that when the first classifier $y_1$ is trained, it treats all data points as being of equal importance in the loss function minimised in step 2a. The loss function $\mathcal{L}_m$ picks up a contribution of $0$ from those data items that are correctly predicted, and a contribution of $w_n^{(m)}$ from those terms that are not correctly predicted. The loss function is therefore weighted to emphasise correctly classifying those points that were incorrectly classified by preceding classifiers.

Step 2b compute the average loss function over the dataset. If all of the samples are correctly classified, $\epsilon_m = 0$; if the arey all incorrect, $\epsilon_m = 1$. The quantity $\alpha_m$ is very large and positive if the classifier was accurate, and very large and negative is the classifier is inaccurate.

In step 2c, the weights are updated using an exponential loss term. For the next classifier in the sequence, each data point is reweighted. If the current classifier got it wrong, the weight is increased by a factor $\exp \alpha_m$; if the current classifier got it right, the weight stays the same (because the indicator function is 0.) In this way, data points that are consistently incorrectly classified accumulate additional weight when the next classifier is trained, and this means that they are more heavily penalised in the loss function. The exponential loss means that points that are persistently difficult can gain weight quickly and hence encourage their correct classification.

Furthermore, in step 3, the best classifiers in the ensemble are the most heavily weighted in the final model prediction $Y(\mathbf{x})$.

Adaboost was designed to work with ensembles of a type of learner called a *decision tree*. We have not considered these so far because they are typically not a very effective learning method when used in isolation, but they can be made very powerful when used in an ensemble. They are also at the heart of the next ensemble method we will study, the random forest.

## *Decision Trees*

Decision trees are perhaps the simplest and easiest to understand of all machine learning methods because they correspond, in a weak sense, to the way in which we make decisions (it is rather unnatural for us to think in terms of mapping every problem into a vector notation.)

In a decision tree, we learn an explicit set of binary decisions on the features in the data in order to arrive at a final decision. This is perhaps best illustrated with a (trivial) example. Consider the process that you may go through when considering whether to buy a pair of shoes. You may ask the following questions:

- Do I want them?
- Can I afford them?
- Are they in my size?

These are essentially the *features* based on which we will make a classification decision (buy or not). These questions can be visualised in a tree-like diagram and the path by which one might make a decision is shown in Figure 49

This is, of course, not a unique tree for this example. Figure 50 shows another way in which this tree could be constructed. Each of us, as individuals, will arrive at our decision in different ways and our decision tree will be constructed according to our own personal priorities.

In machine learning, we hope to be a little more objective about the way we make our decisions and in constructing a decision tree,

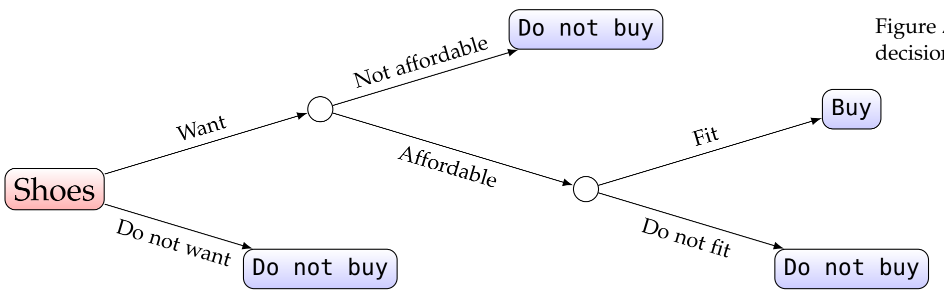

Figure 49: A simple example of a decision tree.

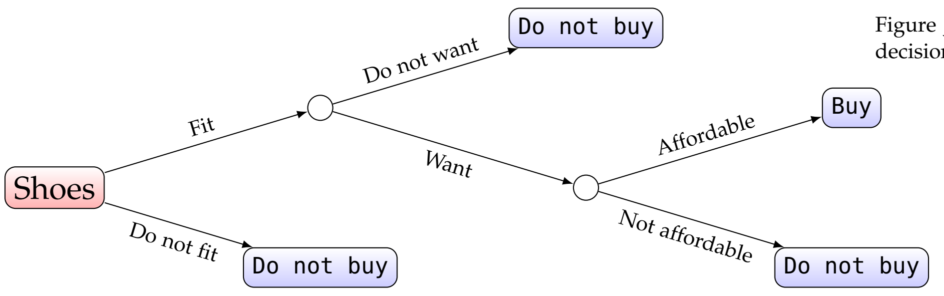

Figure 50: A second example of a decision tree with a different ordering.

we will need to consider how we choose the following aspects of our tree:

- In what order should we consider the features?
- What should our decision making criteria be for each feature?

To illustrate this, a further refinement of our decision tree is shown in Figure 51. In this formulation, we have taken whether the shoe fits or not to be the most informative feature: if they do not fit, we absolutely will not consider buying the shoes and the other features are not considered. If they do fit, then we consider the cost, introducing a specific price above which we will not buy the shoes, regardless of whether we like them. The key idea when learning a decision tree is that we need to learn, from the data:

1. The order in which the features should be applied.
2. What the decision threshold value of each feature should be.

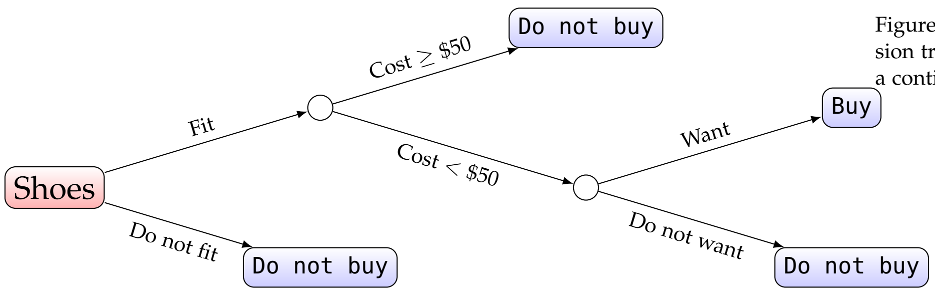

Figure 51: A third example of a decision tree with a different ordering and a continuous variable.

Perhaps the most common method for learning a decision tree is the *C4.5* algorithm developed by Ross Quinlan. This method performs a recursive partioning of the dataset in the following way (a simplified version):

1. Start with a dataset $\mathcal{D} = \{\mathbf{x}_i, t_i\}_{i=1}^N$ with binary target variables $t_i = \{-1, 1\}$. Each data point $\mathbf{x}_i$ is a vector of $P$ features.
2. Determine which feature is best able to split the dataset according to its target values and determine the value on which to split.
3. Split the dataset into two according to the decision learned in the previous step.
4. Recurse on the two partitioned subsets.
5. Stop recursing if a subset contains samples from only one of the target classes

The key issue therefore becomes how to split the dataset into two, and how to determine which feature is the best one to split on next. Remembering that the goal is to split the data such that the target variable can be predicted following the split, we see that the split has to be designed to make the two parts as homogeneous as possible – we ideally want to put all sample with one target value in one group, and all those with the other target value in the other group. The choice of feature on which to split is therefore the one that acheives the greatest homogeneity in each half of the split.

The criterion typically used for splitting is the *information gain* following a split, which is related to the *entropy* of the dataset. Entropy is a concept that originated in statistical thermodynamics as a way to reason about the properties of large collections of atoms/molecules. It is a rough measure of the amount of disorder in a system and its definition, $S = k \ln w$ (where $k$ is Boltzmann's constant) reflects this. It states that the entropy $S$ is the log of the number of possible microscopic configurations $w$ that could give rise to the same macroscopic properties. A classical example of this is to consider a box full of some sort of gas. There are many possible arrrangements of the has molecules in which they are uniformly distributed in the box, and these will be indistinguishable to us from the outside. This state has a high entropy. It is also possible (althought very unlikely) that the atoms could end up all packed very close together in one corner of the box. There are many few ways in which this could be done and this is a low entropy state. The parallel with our current situation is reasonably obvious: we need to

- Split the data on each feature to minimise the entropy
- Choose the feature that minimises the entropy

In information theory, entropy is defined in a slightly different way. The analogue of $w$ is the probability density $p_i$ for discrete outcomes $i$, and in terms of which the *information entropy* is defined as

$$S = -\sum_i p(i) \ln p(i) \tag{174}$$

This definition tells us how much information an event contains: unlikely events (small $p$) give a large value of $S$ and carry more information than highly likely events. To understand this, consider the example of a single binary variable for which $p(0) = p$ and $p(1) = 1 - p$. The entropy of this variable as a function of $p$ is $S = -p \ln p - (1-p)\ln(1-p)$ which we plot in Figure 52. We observe that in the case where the output is entirely predictable ($p = 0$ or $p = 1$) then the entropy is zero: there is no information because the variable is entirely predictable. The entropy is at its highest when $p = 0.5$ and the variable is at its most unpredictable.

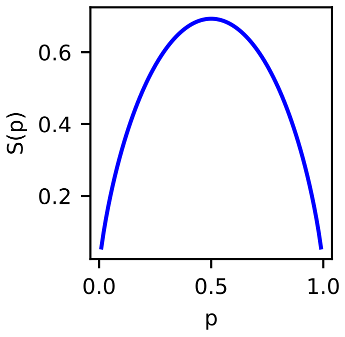

Figure 52: The entropy of a binary variable for which $p(0) = p$ and $p(1) = 1 - p$.

In the context of splitting a dataset into two homogeneous parts, we observe that a dataset that is homogeneous has a low entropy (there are no surprising events). It therefore follows that we should choose to split the data in a way that provides the biggest *reduction in entropy*. This is equivalently referred to as the biggest *information gain*. In a decision tree, we order to decisions by the amount of information that is gained at each split.

Since lowering entropy implies gaining information, the information gain in splitting the data can be is defined as the difference between the entropy of the parent $P$ and the entropy of the children $C = \{c_i\}$. Given an $n$-way split of the sample, we write this as the entropy of the parent minus the weighted (by relative probabilities) sum of the children's entropy:

$$G(P,C) = S(P) - S(C) \tag{175}$$

$$- \sum_{i\in P} p(i)\ln p(i) - \sum_{c\in C} p(c) \sum_{i\in c} -p(i|c)\ln p(i|c) \tag{176}$$

Let us do a concrete example of this, using our shoe-buying example. A sample dataset, perhaps of someone's shoe-buying history, is given in Table 8. Let us begin to construct the decision tree.

| N | Like | Afford | Size | Buy |
|---|---|---|---|---|
| 1 | T | F | T | F |
| 2 | F | T | F | F |
| 3 | T | F | T | T |
| 4 | T | F | T | T |
| 5 | F | T | F | F |
| 6 | T | T | T | T |
| 7 | F | F | F | F |
| 8 | T | T | T | T |
| 9 | F | T | T | T |
| 10 | T | F | F | F |

Table 8: Table for outcomes for the shoe-buying example.

Our first choice is to determine on which variable we should split first. We need to first compute the entropy of the whole dataset. There are ten samples from which the outcome was true on four occasions and false on six. The entropy of the parent is therefore

$$S(P) = -0.4\ln 0.4 - 0.6\ln 0.6 = 0.673 \tag{177}$$

Let us split the dataset on each of the three independent variable. First, we split on "Like", dividing the dataset into two groups: six samples for which "Like" is true, and four for which it is false. Of the six "true's", four lead to "buy" and two to "not buy". Of the four false's, none lead to "buy" and four lead to "not buy". The children's entropy is therefore

$$\begin{aligned} S(C) &= \sum_{c \in C} p(c) \sum_{i \in c} -p(i|c) \ln p(i|c) && (178) \\ &= \left[ p(\text{Like}) \times \sum_{i \in \text{Like}} -p_i \ln p_i \right] + \left[ p(\neg\text{Like}) \times \sum_{i \in \neg\text{Like}} -p_i \ln p_i \right] && (179) \\ &= 0.6 \times \left( -\frac{4}{6} \ln \frac{4}{6} - \frac{2}{6} \ln \frac{2}{6} \right) + 0.4 \times (-1 \ln 1 - 0 \ln 0) && (180) \\ &= 0.382 && (181) \end{aligned}$$

The information gained is therefore $0.673 - 0.382 = 0.291$. Let us do this for the other variables.

For Affordability, the split is 5T, 5F. The 5T lead to 2T, 3F outcomes; the 5F lead to 2T, 3F outcomes. The children's entropy is therefore

$$\begin{aligned} S(C) = 0.5 \times \left( -\frac{2}{5} \ln \frac{2}{5} - \frac{3}{5} \ln \frac{3}{5} \right) + 0.5 \times \left( -\frac{2}{5} \ln \frac{2}{5} - \frac{3}{5} \ln \frac{3}{5} \right) && (182) \\ = 0.5 \times 0.673 + 0.5 \times 0.673 = 0.673 && (183) \end{aligned}$$

No information has been gained. We should not split on this variable.

For Size, the split is 6T, 4F. The 6T give 5T, 1F outcomes; the 4F give 0T, 4F outcomes.

$$S(C) = 0.6 \times \left( -\frac{5}{6} \ln \frac{5}{6} - \frac{1}{6} \ln \frac{1}{6} \right) + 0.4 \times (-1 \ln 1 - 0 \ln 0) \quad = 0.270 + 0 = 0.270 \tag{184}$$

The information gain here is therefore $0.673 - 0.270 = 0.403$.

The best inital predictor for the outcome is therefore whether the shoes fit. This should therefore be the first split in the tree. Subsequent partionings of the data follow the same principle and the tree can be constructed recursively. These arguments can be generalised to deal with continuous variables although we will not consider that here. The aim has been to understand the general principles by which these tree are constructed. Now that we have acheived this, we can look at how trees can be used in concert with each other in an ensemble.

## *Random Forests*

One of the most well-known problems with decision trees is that they are notorious for overfitting to their training data and they do not generalise well (they are high-variance/low-bias). The idea of using multiple trees and somehow combining their results has therefore been studied for many years, and the random forest algorithm is a combination of many of the ideas that have been tried over that time. Random Forests are based on three core ideas:

- Decision Tree learning
- Bagging (bootstrap aggregating)
- Random subspaces

Before we get into the technical detail of the algorithm, it is useful to develop an understanding of the concepts before we get into the mathematical formulation. We have already seen how decision trees work, so let us now consider the other aspects of the problem before we integrate them into the random forest method.

## *Bagging*

Bagging – Bootstrap aggregating is a simple and widely used technique for reducing variance and improving the stability of machine learning methods. At its core is a repeated subsampling of the dataset. This sounds a bit like cross-validation, but there are some key differences.

- Given a dataset $\mathcal{D}$ of $N$ samples and integers $m$ and $n$:
- for $i = 1 \Rightarrow m$
  - Create new dataset $\mathcal{D}_i$ containing $n'$ random samples from $\mathcal{D}$. Sampling should be *uniform* and *with replacement*.
  - Train model $y_i$ on $\mathcal{D}_i$
- Predict by taking the average over the $m$ models.

The key to bagging is the sampling strategy, which is *with replacement*. This means that when a sample is drawn from the dataset, it remains in the the dataset and can be sampled again. Each of the $m$ samples can therefore contain multiple instances of each item in the dataset. On average, it can be shown that if the dataset is large, and the sampling fraction $n/N = 1$, then a fraction $1 - \frac{1}{e} \approx 0.63$ of the data points in the sample will be unique, and the remainder will be repeats of these.

This approach is known as *bootstrap sampling*, and it is a method for improving statistical estimation problems. The central idea is that multiple resampling of the data with replacement can improve an estimate of the true underlying probability distribution compared to the single original sample. When applied to a machine learning problem, it provides a means for controlling the variance of the model.

## *Random Subspaces*

A core feature of random forests as compared to other model averaging techniques is that they apply the principle of bagging to the *features* of the data. The key to doing this is a technique called *random subspace learning* in which the features of the data are randomly sampled, and the models are trained on these subsets. It works in a very similar way to the standard bagging method:

- Given a dataset $\mathcal{D}$ of $N$ samples, $M$ features; and $L$ learners
- for $l = 1 \Rightarrow L$
  - Select $n_l$ as the input size for learner $y_l$
  - Draw $n_l$ features with replacement and train $y_l$ on that sample
- Predict by taking the average over the $L$ models.

Notice that random subspaces are conceptually quite similar to random projections, but with a key difference: random projections use a weighted sampling of the features rather than a binary sampling with replacement.

## *The Random Forest Algorithm*

Bagging, by means of its sampling strategy, allows high-variance, low bias models such as decision trees to be combined to produce a lower variance model by means of a simple averaging strategy. Although this is an attractive proposition due to its simplicity, it is typically outperformed by the more sophisticated boosting method in which a committee of weak learners are evolved over time to perform better on hard-to-classify data points.

The random forest algorithm[6] is a modification of the bagging method that is typically seen to match or improve on the performance of boosting whilst being substantially simpler to train because it is a simple committee machine. The central idea is to average the prediction of many decision trees that are *de-correlated* by means of sampling random subspaces on bagged samples from the dataset. This approach works very well for ensembles of decision trees because they can each individually produce a low-bias estimate on the sample that they are trained on, capturing complex structure in the data. The averaging process then renders the ensemble with a much lower variance than any individual tree.

[6] Breiman, L. Machine Learning (2001) 45: 5. https://doi.org/10.1023/A:1010933404324

The key to the success of random forests lies in the behaviour of the averages of random variables. In a simple committee machine (including standard bagging aproaches), each learner is trained on a different subsample of the dataset. We consider the output of each of $K$ learners (with bagged samples) to be a random variable with variance $\sigma^2$. If the learner's outputs are i.i.d. (independent, identically distributed, then their average has variance $\frac{1}{K}\sigma^2$. Averaging therefore has a variance-reducing effect. However, their ouputs will typically not be i.i.d. because their inputs will be correlated, and thus their decisions will also be correlated. If the learners are only i.d. (identically distributed, not necessarily independent), then the variance of the average is

$$\rho\sigma^2 + \frac{1-\rho}{K}\sigma^2 \tag{185}$$

where $\rho$ is the positive pairwise correlation between learners. Correlations between the predictions of the learners therefore limits the benefits of averaging and hence of bagging: when $\rho \approx 1$, this expression reduce to $\approx \sigma^2$ and there is no benefit to averaging (equivalent to $K = 1$). However, if $\rho \to 0$ the learners are uncorrelated and the variance of the average tends to $\frac{1}{K}\sigma^2$, the same as if they were i.i.d. The Random Forest algorithm therefore aims to reduce the pairwise correlation between learners to enhance the effect of the averaging.

The basic random forest algorithm is described as follows:

1. For $k = 1 \to K$
   (a) Draw a "bootstrap" sample (with replacement) of $N$ samples from the training data (of size $N$).
   (b) Build a decision tree $T_k$ to a desired depth on the bootstrapped sample by:
      i. Randomly selecting $m$ from $M$ variables to select a random subspace.
      ii. Split the data on the $m$ variables using (for example, by maximising information gain).
      iii. Repeat recursively, splitting the data using a different random subspace each time.
      iv. Stop when the tree reaches the desired depth.
2. Return the ensemble of trees $\{T_k\}_{k=1}^{K}$.

Predictions are made by: for regression, taking the average of the ensemble predictions; for classification, taking a majority vote. Note that the bootstrap samples will contain roughly $1 - \frac{1}{e} \approx 0.63$ unique points, as per the discussion on bagging.

The key to the random forest method is that the random selection of variables on which to split at each stage means that the correlations between the trees is considerably reduced. The smaller the value of $m$, the greater the reduction in correlation.

Comparing random forests to boosting is generally rather difficult. There are a few generalisations one can make:

- If only a small number of variables are relevant, small $m$ may give poor results because many of the trees will give poor estimates.
- As long as $m$ is large enough to capture the relevant variables, random forests are robust to the presence of uninformative noise variables.
- Random forests are often claimed to not overfit, but growing large numbers of very deep trees can result in a very powerful model that can overfit. Controlling the depth of the trees by premature stopping can alleviate this, but this also naturally increases the bias (a fully grown tree is unbiased on its training data). A general result is that the bias of the forest is the same as the bias of any individual tree, if the trees are grown to full depth.

- Random forests are naturally amenable to parallel training so can be substantially faster than boosting, which is an inherently sequential algorithm.
- Random forests can often perform very well with relatively little tuning and, like Adaboost, are frequently considered to be one of the best out-of-the box classifiers. This is, of course, somewhat problem dependent.

*Further reading: Chapter 15, Hastie, Tibshirani and Friedman, The Elements of Statistical Learning, Second Edition, Springer.*