LAST CHANCE STATS

JOSEPH BOYD

Contents

1. Warm Up

This section presents an assortment of mathematical problems, each embodying an idea that is useful or instructive to the material that follows.

1.1. **Proof of Pythagorean Theorem.** Consider a square with sides of length c rotated inside a larger square, such that the four corners of the smaller square meet a distinct edge of the larger at a point a units along that edge. The sides of the larger square are therefore of length $a+b$ for some $a, b > 0$. The area of the four resulting right-angled triangles that fill the empty space are $\frac{1}{2}ab$ apiece. Thus, we have it that,

$$(a+b)^2 = c^2 + 4 \cdot \frac{1}{2}ab,$$

expanding to,

$$a^2 + 2ab + b^2 = c^2 + 2ab,$$

and finally,

$$a^2 + b^2 = c^2.$$

1.2. **Irrationality of $\sqrt{2}$.** Suppose $\sqrt{2}$ is rational. Then there exist $a, b \in \mathbb{Z}$, with $a > b$ and $gcd(a, b) = 1$[1] such that,

$$\frac{a}{b} = \sqrt{2}.$$

Consequently,

$$\frac{a^2}{b^2} = 2.$$

This implies that a^2 is even, and therefore that a is even. That is,

$$a = 2k,$$

for some $k \in \mathbb{Z}$. So, by substitution,

$$\frac{4k^2}{b^2} = 2.$$

Rearranging,

$$b^2 = 2k^2.$$

However, this now implies that b^2 is even, hence b also, implying a common factor between a and b, though by assumption they have no common factors, and clearly we can repeat the argument ad infinitum. This contradiction proves $\sqrt{2}$ is irrational[2].

[1] That is, $\frac{a}{b}$ is an irreducible fraction. This can be obtained by cancelling out the common factors. Every integer has a unique prime factorisation due to the fundamental theorem of arithmetic.

[2] This is an example of a proof by contradiction.

1.3. **Euclid's Proof of the Infinitude of Primes.** Take any finite set of primes, $p_1, p_2, \ldots, p_n$. Then, the number, $q = p_1 \times p_2 \times \cdots \times p_n + 1$, is either prime or not prime. If it is prime, we have generated a new prime not in our set. If it is not, it has a prime factor not in our set, as $q \equiv 1 \pmod{p_i}$ for $i = 1, 2, \ldots, n$. Using this method we can always generate a new prime[3]. It follows that there are infinitely many primes.

1.4. **The Potato Paradox.** A potato that is 99% water weighs 100g. It is dried until it is only 98% water. It now weighs only 50g. How can this be?

The mistake people typically make is to assume there has been a 1% reduction in the volume of water, rather than a change to the water/non-water ratio. In fact, the ratio of water to non-water content has changed from $1 : 99$ to $2 : 98$ or $1 : 49$. Since the non-water content of the potato has not changed, the water must have lost 50% of its volume, and so the potato now weighs half as much!

1.5. **The Monty Hall Problem.** This famous paradox, named after an American game show host, has the following problem statement:

A game show presents you with the choice of selecting one of three doors. Two of the doors contain a goat (undesirable), whilst the third contains a car. Upon your choice, the game show host, who knows the contents of each of the doors, opens one of the two remaining doors to reveal a goat. He then gives you the option of either sticking with your initial choice, or switching to the one remaining door. Is it advantageous to switch doors?

The answer is that it *is* advantageous, increasing your winning chances from $1/3$ to $2/3$. The essential realisation is that the host (Monty Hall) has knowledge of the door contents and will always reveal a goat, whether you have chosen the car or not.

The solution can be understood as a decision theory problem. We designate two strategies: changing and not changing doors. In either case, there is a $2/3$ chance of initially having a goat. Because the other goat is eliminated independently by the host, changing wins whenever a goat was initially chosen, that is, with probability $2/3$. Not changing wins only if the car was initially chosen, that is, with probability $1/3$. Therefore changing will double your winning chances!

Another (admittedly more complicated) way to solve the problem is using Bayes' theorem. Here we designate C_1, C_2, and C_3 as the events the car is behind doors 1, 2, and 3 respectively; X_1 the event the player chooses door 1; and H_3 the event the host opens door 3. Note the numbering of the doors does not matter. For example, 'door 2' simply refers to 'the other door'. First, $\Pr(H_3|C_1, X_1) = 1/2$, $\Pr(H_3|C_2, X_1) = 1$, and $\Pr(H_3|C_3, X_1) = 0$, reflecting the knowledge that the host always opens a goat door (never the car). Then,

[3]This is an example of a constructive proof.

$$\begin{aligned}\Pr(C_2|X_1,H_3) &= \frac{\Pr(H_3|C_2,X_1)\Pr(C_2|X_1)}{\Pr(H_3|X_1)}\\ &= \frac{\Pr(H_3|C_2,X_1)\Pr(C_2|X_1)}{\Pr(H_3|C_1,X_1)\Pr(C_1|X_1)+\Pr(H_3|C_2,X_1)\Pr(C_2|X_1)+\Pr(H_3|C_3,X_1)\Pr(C_3|X_1)}\\ &= \frac{1\cdot 1/3}{1/2\cdot 1/3+1\cdot 1/3+0\cdot 1/3}\\ &= \frac{2}{3},\end{aligned}$$

so the winning odds for changing doors is 2/3, hence not changing only 1/3.

1.6. **Finding Hamlet in the Digits of π.** It is supposed, though not known for certain, that π is a normal number. This means that in the infinitely many decimal digits of π, every sequence of n digits appears as often as every other length n sequence. So, 123 occurs as often as 321 or 666, but more often than 123456789, because it is a shorter sequence. In this way, the behaviour of the digits of π is indistinguishable to what would arise from a uniform random distribution. Of course, the digits of π are not random, as many formulas exist for generating them. The same rules of normalness apply if π is converted to a different base, for example, binary code–the number system of computer arithmetic. The number π in binary is,

$$\begin{aligned}\pi &= 3.14...\\ &= 1\cdot 2^1+1\cdot 2^0+0\cdot 2^{-1}+0\cdot 2^{-2}+1\cdot 2^{-3}+0\cdot 2^{-4}+0\cdot 2^{-5}+1\cdot 2^{-6}+\cdots\\ &= 11.001001...\end{aligned}$$

Note that to express π in binary rather than decimal notation requires more digits, because it takes $\log_2(10)\approx 3.3$ binary digits (bits) to express each power of ten. Nevertheless, the same properties about the normalness of π hold in its binary form–000 is just as likely as 111, and so on.

Now, to find Hamlet, we would need to start with a common base. Hamlet is written in Latin characters. One way to convert Latin characters to numbers is to use a character encoding standard, such as ASCII (American Standard Code for Information Interchange). People use ASCII indirectly every day–it is how our data comes to be represented by computers and communication systems. ASCII gives a representation of each character in 8 binary digits. For example, in ASCII,

$$\text{“pie”} = 01110000\ 01101001\ 01100101.$$

So, Hamlet, like any text, has an ASCII, and therefore binary representation. Without specifying exactly what it is we can determine its length. There are 186,391 characters in the 31,842 word Hamlet, including spaces and indentation. At 8 bits per character, this is 1,491,128 bits.

To develop a formula, we first consider a simpler problem: what is the expected number of digits we should pass over in a random binary sequence before we encounter a specific binary sub-sequence, say, 11? If we consider a binary tree covering all the possibilities, and let the expected length to find 11 be denoted by $\mathbb{E}_{11}$, then we may first see 0 with probability $\frac{1}{2}$, in which case we are back to where we started. That is, the first digit is a failure, because it is not the one we are looking for. The expected length from there is therefore $1 + \mathbb{E}_{11}$. We might alternatively see 1 in the first digit–a match–also with probability $\frac{1}{2}$. In this case, the next digit is 0 with probability $\frac{1}{2}$, which would mean a failure on the second digit, and again, we return to the starting point, with expected length now $2+\mathbb{E}_{11}$. Otherwise, we could get a 1 in the second digit, and we have found the match straight away. In this case, the expected length is simply 2. Therefore we may write,

$$\mathbb{E}_{11} = \frac{1}{2} \times (1 + \mathbb{E}_{11}) + \frac{1}{4} \times (2 + \mathbb{E}_{11}) + \frac{1}{4} \times 2,$$

which we may solve to find $\mathbb{E}_{11} = 6$. That is, we expect to pass over 6 binary digits (on average) before we see the particular sequence, 11. We see that in general, it is possible to fail on each of the n digits in the sequence. Thus, we have the following expression for the general n-bit sequence,

$$\frac{1}{2^n}\mathbb{E}_{\{0,1\}^n} = 1 \times \frac{1}{2^1} + 2 \times \frac{1}{2^2} + 3 \times \frac{1}{2^3} + \cdots + 2n \times \frac{1}{2^n}.$$

If we denote this sum S then clearly,

$$\begin{aligned} S - \frac{1}{2}S = 1 \times \frac{1}{2^1} + 1 \times \frac{1}{2^2} + 1 \times \frac{1}{2^3} + \cdots + 1 \times \frac{1}{2^n} = 1 - \frac{1}{2^n} \\ \implies S = 2 - \frac{1}{2^{n-1}}, \end{aligned}$$

hence, $\mathbb{E}_{\{0,1\}^n} = 2^{n+1} - 2$. Applying this formula, we can expect to find Hamlet somewhere within the first $2^{1,491,129} - 2 \approx 3.6 \times 10^{448,874}$ digits of π.

2. Differential Calculus

This section presents the fundamental results of differential calculus: first the formal definition of a limit, then that of a derivative. In particular, we note the relationship between the binomial theorem and the differentiation of polynomials. The section culminates with a proof of the mean value theorem.

2.1. **Binomial Theorem.**

The binomial theorem expresses a formula for expanding powers of a binomial[4]. For example, $(x+y)^3 = x^3+3x^2y+3xy^2+y^3$. In general, the binomial expansion is,

[4]A binomial is the sum of two *monomials*, that is, a polynomial with a single term.

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k},$$

where,

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

is known as the binomial coefficient. Note the binomial coefficient is symmetric, that is, $\binom{n}{k} = \binom{n}{n-k}$. This symmetry may be seen in Pascal's triangle,

$$\begin{array}{ccccccccccccc}
 & & & & & & 1 & & & & & & \\
 & & & & & 1 & & 1 & & & & & \\
 & & & & 1 & & 2 & & 1 & & & & \\
 & & & 1 & & 3 & & 3 & & 1 & & & \\
 & & 1 & & 4 & & 6 & & 4 & & 1 & & \\
 & 1 & & 5 & & 10 & & 10 & & 5 & & 1 & \\
1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1
\end{array}$$

where each number is the binomial coefficient with n the row number and k the column. Pascal's triangle is named after French mathematician Blaise Pascal (1623-1662). Note that each number is the sum of the two numbers diagonally above. In terms of binomial expansion, this is equivalent to collecting like terms after expanding. To illustrate, consider the coefficients for the quadratic expansion, $x^2 + 2xy + y^2$,

$$1 \quad 2 \quad 1$$

Multiplying again by the binomial, $(x+y)$, gives the coefficients,

$$1 \quad 1 \quad 2 \quad 2 \quad 1 \quad 1$$

which corresponds to $x^3 + x^2y + 2x^2y + 2xy^2 + xy^2 + y^3$, and clearly each of the interior pairs are like terms, so these are summed to arrive at,

$$1 \quad 3 \quad 3 \quad 1$$

In general, this relation is reflected in Pascal's rule,

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k},$$

for $1 \leq k \leq n$. Isaac Newton (1642-1727) extended the notion of binomial expansion to real powers in 1665. This was an important stepping-stone towards the definition of a derivative, as we will see.

2.2. **(ϵ-δ) Definition of a Limit.** The standard formal (ϵ-δ) definition of a limit is due to German mathematician, Karl Weierstrass (1815-1897). We say that,

$$\lim_{x \to a} f(x) = L,$$

if for any value, $\epsilon > 0$, specifically where ϵ is arbitrarily small, we can find a value $\delta > 0$ such that,

$$0 < |x - a| < \delta \implies |f(x) - L| < \epsilon.$$

This somewhat off-putting notation captures the notion of a function approaching a limiting value continuously. It states that however close (within an infinitesimal ϵ units) we wish to get to the limiting value, L, we can always find a correspondingly tiny interval of radius δ around the limiting point a such that the function is closer. Note that this accounts also for the case where $f(x)$ is undefined at $x = a$.

An example is always useful for such subtle ideas. Therefore, let $f(x) = 2x^2 + 2$. Then clearly, $\lim_{x\to 0} f(x) = 2$. To satisfy the inequality, $|2x^2 + 2 - 4| < \epsilon$, we can select $\delta = \sqrt{\epsilon/2 + 1}$. Clearly this is always possible, no matter how small we take ϵ to be, and so the limit exists.

A function is said to be *continuous* at a point a if the limit L exists, and $f(a) = L$, which is not always the case. Suppose $g(x) = 1$ for $x < 0$ and $g(x) = 2$ for $x \geq 0$. Such a function makes an instantaneous jump at $x = 0$ to a value distinct from the limit to which it was approaching, and is therefore *discontinuous* and does not satisfy the ϵ-δ condition. Alternatively, the limit may exist, but the function may not be defined at the limiting point, thus also a discontinuity. A function is continuous if it is continuous at all points in its domain. A function that is *differentiable* at a point, a, implies continuity at a, but not vice-versa. For example, the absolute value function, $f(x) = \text{abs}(x)$ is continuous but not differentiable. Differentiability of a function at a point roughly means it is possible to measure the *trajectory* of the function at that point, so that it is possible to draw a tangent line at that point.

2.3. **Differentiating From First Principles.** The gradient of a linear function, $y = f(x)$, over an interval, $[x, \Delta x]$, is defined as $\Delta y/\Delta x$, that is, 'rise over run'. The derivative of a function is simply the gradient taken at a point. Formally, we define a derivative as,

$$\frac{d}{dx} f(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x},$$

and it is clear this is just 'rise over run', albeit taken on an infinitesimal interval. It represents the gradient of a line tangent to the curve at x. To see how we can differentiate a polynomial, consider the general nth degree polynomial[5], $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$. Then, from the definition,

$$\begin{aligned}
\frac{d}{dx} f(x) &= \lim_{\Delta x \to 0} a_n \frac{(x + \Delta x)^n - x^n}{\Delta x} + \cdots + a_1 \frac{(x + \Delta x) - x}{\Delta x} + a_0 \frac{1 - 1}{\Delta x} \\
&= \lim_{\Delta x \to 0} a_n \frac{\binom{n}{0} x^n + \binom{n}{1} x^{n-1} \Delta x + \cdots + \binom{n}{n} (\Delta x)^n - x^n}{\Delta x} + \cdots + a_1 \frac{\Delta x}{\Delta x} + a_0 \frac{0}{\Delta x} \\
&= n a_n x^{n-1} + (n-1) a_{n-1} x^{n-2} + \cdots + a_1,
\end{aligned}$$

and it is clear that only the second term of each expansion survives cancelation, or annihilation by the infinitesimal Δx. From this we can infer that $\frac{d}{dx} a_k x^k = k a_k x^{k-1}$, the

[5]Note that the powers of a polynomial are non-negative integers by definition.

rule that every student of calculus is familiar with, though usually does not realise it to be a consequence of the binomial theorem! It is only when one tries to 'plug and play' from first principles that these connections become apparent. As noted before, Newton extended the definition of the binomial theorem to real-valued powers.

2.4. **Chain Rule.** The chain rule for differentiation describes how to differentiate function compositions, $f \cdot u \triangleq f(u(x))$. Assuming $u(x)$ is differentiable at a, and $f(u)$ is differentiable at $u(a)$, the derivative,

$$\begin{aligned}(f \cdot u)'(a) &= \lim_{x \to a} \frac{f(u(x)) - f(u(a))}{x - a} \\ &= \lim_{x \to a} \frac{f(u(x)) - f(u(a))}{u(x) - u(a)} \cdot \frac{u(x) - u(a)}{x - a} \\ &= f'(u(a))u'(a).\end{aligned}$$

An extension to this expression is needed for a rigorous proof, as a function may oscillate as it tends towards a limit, and $u(x)$ may equal $u(a)$ infinitely many times, resulting in a inexhaustible number of undefined points as $x \to a$, but we will leave it as a sketch.

In general, the chain rule may be applied repeatedly for any number of nested functions,

$$\frac{d}{dx} f_n(f_{n-1}(\cdots(f_1(x)))) = \frac{df_n}{df_{n-1}} \cdot \frac{df_{n-1}}{df_{n-2}} \cdot \cdots \cdot \frac{df_2}{df_1} \cdot \frac{df_1}{dx}.$$

The corresponding technique for anti-differentiation is *integration by substitution.*

2.5. **Product Rule.** The product rule for differentiation allows us to differentiate products of functions of x. Suppose $f(x) = u(x)v(x)$. Then, from first principles,

$$\begin{aligned}f'(x) &= \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \\ &= \lim_{\Delta x \to 0} \frac{u(x + \Delta x)v(x + \Delta x) - u(x)v(x)}{\Delta x} \\ &= \lim_{\Delta x \to 0} \frac{u(x + \Delta x)v(x + \Delta x) - u(x)v(x + \Delta x) + u(x)v(x + \Delta x) - u(x)v(x)}{\Delta x} \\ &= \lim_{\Delta x \to 0} \frac{u(x + \Delta x) - u(x)}{\Delta x} \cdot v(x + \Delta x) + \frac{v(x + \Delta x) - v(x)}{\Delta x} \cdot u(x) \\ &= u'(x)v(x) + v'(x)u(x).\end{aligned}$$

The quotient rule can be derived by applying both the product rule and the chain rule. For $f(x) = u(x)/v(x)$, write, $f(x) = u(x)v(x)^{-1}$. Then,

$$\begin{aligned}\frac{d}{dx}f(x) &= u'(x)v(x)^{-1} - u(x)v'(x)v(x)^{-2} \\ &= \frac{u'(x)v(x) - u(x)v'(x)}{v(x)^2}.\end{aligned}$$

The corresponding technique for anti-differentiation is *integration by parts*.

2.6. **Rolle's Theorem.** Rolle's theorem by Michel Rolle (1652-1719) asserts that for a differentiable function, $f(x)$, if $f(a) = f(b)$ for some a and b, then there must be a stationary point on the interval $[a, b]$, unless $f = c$ for some constant, c. Plainly speaking, it says that if a curve returns to its starting point at the end of an interval, there must be at least one turning point somewhere in that interval. To prove this, first consider that there must be at least one extreme point c on the interval $[a, b]$, be it maximum or minimum. For brevity, we assume that c is a maximum. So, for all points after the interior point, c, $c + h$, such that $h > 0$, the function is upper-bound by $f(c)$. That is,

$$\frac{f(c+h) - f(c)}{h} \leq 0.$$

Taking the limit gives,

$$f'(c) = \lim_{h \to 0^+} \frac{f(c+h) - f(c)}{h} \leq 0.$$

Likewise, for $h < 0$ we have,

$$f'(c) = \lim_{h \to 0^-} \frac{f(c+h) - f(c)}{h} \geq 0,$$

from which it follows that the derivative of the function, $f'(x)$, is 0 at c.

2.7. **Mean Value Theorem.** The mean value theorem states that for a differentiable function defined on an interval $[a, b]$, at least one point on the curve must have gradient equal to that of the chord connecting the endpoints, $(a, f(a))$ and $(b, f(b))$. It is an intuitive result, as any curve which succeeds in climbing (or descending) from $f(a)$ to $f(b)$, must match the rise over run of that slope at least at one point, otherwise it will not complete the ascent.

First, define $g(x) = f(x) - rx$. This must be differentiable on $[a, b]$ since $f(x)$ is by construction. We choose r such that $g(x)$ satisfies Rolle's theorem, that is, $g(a) = g(b)$, hence $f(a) - ra = f(b) - rb$, and $r = (f(a) - f(b))/(a - b)$, that is, the slope of the chord connecting the endpoints. Because of Rolle's theorem, for some $c \in [a, b], g'(c) = 0$, hence $f'(c) = r$, and the proof is complete.

3. Linear Algebra

A set of M equations in N variables can be referred to as a *system of linear equations*. Each variable may feature in one or more of the M equations. One is typically interested in finding a *simultaneous solution*, that is, values for each of the variables satisfying all the equations at the same time. If $M < N$ (more variables than equations), the system is *underdetermined*. In this case, there will not be enough information to identify a unique solution, and the solution will be expressed in terms of at least one of the variables, giving an infinite family of solutions. If any equations are mutually contradictory[6], however, the system will have no solution. If $M > N$, the system is *overdetermined*. Unless the excess equations are in fact linear combinations of others, the solution will necessarily have mutually contradictory equations[7]. A system of linear equations may alternatively be written as a matrix equation,

$$\mathbf{Ax} = \mathbf{b},$$

where $\mathbf{A}$ is the matrix of coefficients, $\mathbf{x}$ is the vector form of the simultaneous solution, and $\mathbf{b}$ is the vector of constants. The solution, $\mathbf{x}$, may therefore be found by inverting $\mathbf{A}$. A square matrix $\mathbf{A} \in \mathbb{R}^{N\times N}$ is invertible or non-singular if there is a matrix $\mathbf{A}^{-1} \in \mathbb{R}^{N\times N}$ such that $\mathbf{AA}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. Non-square matrices can have at most distinct left or right inverses. There are many matrix properties necessary and sufficient for invertibility. In particular, the rows and columns of $\mathbf{A}$ are linearly independent and span[8] the vector space $\mathbb{R}^N$. This is equivalent to having full rank, which is in turn equivalent to having zero nullity[9]. These properties are connected in the rank-nullity theorem[10], and further in the fundamental theorem of linear algebra. If the values of a matrix are randomly sampled from a uniform distribution, the matrix has an infinitesimal chance of having linearly dependent columns, and hence of being singular. For this reason, singular matrices are less a practical concern in least squares regression than is *multi-collinearity*, that is, highly correlated columns that cause ill-conditioning, risking numerical errors.

3.1. Analytic Form of Inverse.

Another necessary and sufficient property of a non-singular matrix is a non-zero determinant. The determinant has an analytic form recursive in the *minors* of the matrix, that is, each of the N sub-matrices formed by removing a fixed

[6]For example, the equations $x + y + z = 1$ and $x + y + z = 2$ have no simultaneous solution.

[7]If the left-hand sides of some group of N of the M equations are linearly independent, they form a basis in $\mathbb{R}^N$. Therefore, the left-hand side of any excess equation is some linear combination of this group of N. The right-hand side of the excess equation must be equal to the corresponding linear combination of the N right-hand sides, or else it is in contradiction, and the overall system has no solution.

[8]The span of a set of vectors is intuitively its expressiveness as a basis. N linearly independent vectors form a basis for the vector space $\mathbb{R}^N$. On the other hand, if there are linear dependencies in the system, the span will be of lower dimension.

[9]The rank of a matrix, $\mathbf{A}$ is the dimensionality of the vector space spanned with its columns as a basis. This is always equivalent to the span of its rows (row rank of $\mathbf{A}$). The nullity of a matrix is the dimensionality of its null space, that is, the set of all vectors solving $\mathbf{Ax} = \mathbf{0}$.

[10]The rank-nullity theorem states that for $N \times N$ matrix, $\mathbf{A}$, its rank and nullity sum to its number of columns, formally, $\text{rank}(\mathbf{A}) + \text{null}(\mathbf{A}) = N$

row and each of the N columns in turn. Thus, for $\mathbf{A} \in \mathbb{R}^{2\times 2}, \det(\mathbf{A}) = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$. For $\mathbf{A} \in \mathbb{R}^{3\times 3}$,

$$\det(\mathbf{A}) = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix},$$

and for $\mathbf{A} \in \mathbb{R}^{4\times 4}$,

$$\det(\mathbf{A}) = \begin{vmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{vmatrix} = a \begin{vmatrix} f & g & h \\ j & k & l \\ n & o & p \end{vmatrix} - b \begin{vmatrix} e & g & h \\ i & k & l \\ m & o & p \end{vmatrix} + c \begin{vmatrix} e & g & h \\ i & k & l \\ m & o & p \end{vmatrix} - d \begin{vmatrix} e & g & h \\ i & k & l \\ m & o & p \end{vmatrix}.$$

A naive approach to computing a determinant would involve $\mathcal{O}(N!)$ operations. If required, a determinant can be computed as a byproduct of matrix inversion. The determinant is not usually needed in practice, however. The analytic form of the inverse of matrix $\mathbf{A} \in \mathbb{R}^{N\times N}$ is,

$$\mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})}\mathbf{C}^T,$$

where $\mathbf{C} \in \mathbb{R}^{N\times N}$ is the *cofactor* matrix of $\mathbf{A}$, whose elements are each of the N^2 minors of $\mathbf{A}$, that is, the determinants of each of the $(N-1)\times(N-1)$ sub-matrices of $\mathbf{A}$, and with alternating signs. The transpose of the cofactor matrix is denoted the *adjugate* matrix. This form rarely has use outside of theory, as its computation is intractable like the analytic determinant.

3.2. **Positive-definite Matrices.** Non-singular matrices have many interesting properties. One such property is that for non-singular $\mathbf{A}, \mathbf{Ax} = \mathbf{0} \implies \mathbf{x} = \mathbf{0}$. Otherwise put, the null space of $\mathbf{A}$, Null($\mathbf{A}$) is zero-dimensional (and hence, $\mathbf{A}$ has full rank, by the fundamental theorem of linear algebra). Clearly, if there exist two distinct solutions, $\mathbf{x}$ and $\mathbf{y}$ for invertible $\mathbf{A}$, we have it that $\mathbf{A}(\mathbf{x}-\mathbf{y}) = \mathbf{0} \implies \mathbf{A}^{-1}\mathbf{A}(\mathbf{x}-\mathbf{y}) = \mathbf{0} \implies \mathbf{x} = \mathbf{y}$, a contradiction. A matrix $\mathbf{A}$ is positive-definite if $\mathbf{x}^T\mathbf{Ax} > 0, \forall \mathbf{x} \in \mathbb{R}^N n\{0\}$. Note that any eigenvalue λ of $\mathbf{A}$ gives,

$$\begin{aligned} &\mathbf{Ax} = \lambda\mathbf{x} \\ \implies &\mathbf{x}^T\mathbf{Ax} = \lambda\mathbf{x}^T\mathbf{x} > 0, \end{aligned}$$

thus implying $\lambda > 0$ for all λ eigenvalues of positive-definite $\mathbf{A}$. Now, clearly, for positive-definite $\mathbf{A}$, $\mathbf{Ax} = \mathbf{0} = 0 \cdot \mathbf{x} \implies \mathbf{x} = \mathbf{0}$ since we know the eigenvalues of $\mathbf{A}$ are all positive. $\mathbf{A}$ is therefore non-singular. Positive-definite matrices appear in machine learning, for example the *Gram* matrix, and the *Hessian* matrix used in Newton's method for optimisation.

3.3. **Gaussian Elimination.** Gaussian elimination is an ancient technique for solving systems of linear equations, but has in modern times been erroneously attributed to Gauss, who merely contributed to its notation. The technique consists of performing a series of *elementary row operations* to transform the system into a form where its solution or solutions may be extracted directly. The row operations are: swapping two rows; scaling a row by a constant factor, and; adding one row to another. With a finite sequence of these operations, a system may be reduced to *row echelon form*, where the matrix has an upper-triangular form, allowing solutions to be found by back-substituting each variable. When the system is further reduced to *reduced echelon form*, the technique is known as Gauss-Jordan elimination. It may be seen easily that these row operations correspond with multiplying the matrix by slight modifications of the identity matrix, and that the inverse matrix is just the product of these. The technique has a number of useful applications, in particular computing a matrix inverse. This can be found by applying the technique to the system,

$$\mathbf{AX} = \mathbf{I},$$

and reducing $\mathbf{A}$ to $\mathbf{I}$ transforms $\mathbf{I}$ to $\mathbf{A}$ in a product of elementary row operation matrices. Gaussian elimination can also be used to determine the rank of a matrix, as well as to efficiently compute a matrix determinant.

3.4. **Matrix Decompositions.** More efficient algorithms for solving systems of linear equations and inverting matrices come in the form of matrix decompositions. The most widely used, LU decomposition, consists of factorising the matrix into lower and upper triangular matrices. Inverting these triangular matrices may be done very efficiently. QR decomposition is another decomposition where $\mathbf{Q}$ is orthogonal and $\mathbf{R}$ is upper-triangular.

3.4.1. *The Gram-Schmidt Process.* Given a basis set of vectors, $V = \{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_k\}$, we can construct an orthonormal basis with the following approach, first defining

$$\text{proj}_{\mathbf{u}}(\mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}|}\mathbf{u},$$

that is, the vector of the orthogonal projection of the vector $\mathbf{v}$ onto the line of $\mathbf{u}$ (its length given by the inner product). Then,

$$\begin{aligned}
\mathbf{u}_1 &= \mathbf{v}_1 & \mathbf{e}_1 &= \tfrac{\mathbf{u}_1}{|\mathbf{u}_1|} \\
\mathbf{u}_2 &= \mathbf{v}_2 - \text{proj}_{\mathbf{u}_1}(\mathbf{v}_2) & \mathbf{e}_2 &= \tfrac{\mathbf{u}_2}{|\mathbf{u}_2|} \\
\mathbf{u}_3 &= \mathbf{v}_3 - \text{proj}_{\mathbf{u}_1}(\mathbf{v}_3) - \text{proj}_{\mathbf{u}_2}(\mathbf{v}_3) & \mathbf{e}_3 &= \tfrac{\mathbf{u}_3}{|\mathbf{u}_3|} \\
&\vdots & &\vdots
\end{aligned}$$

It can be shown by substitution and mathematical induction that set $E = \{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_k\}$ is orthonormal. It is easy to see, however, that $\mathbf{u}_1$ and $\mathbf{u}_2$ are orthogonal by definition of the projection. Likewise, $(\mathbf{v}_3 - \text{proj}_{\mathbf{u}_1}(\mathbf{v}_3))$ is orthogonal to $\mathbf{u}_1$, and since $\text{proj}_{\mathbf{u}_2}(\mathbf{v}_3)$ is now

also orthogonal to $\mathbf{u}_1$, $\mathbf{u}_3$ is orthogonal to $\mathbf{u}_1$. We can then swap the order to show $\mathbf{u}_3$ is also orthogonal to $\mathbf{u}_2$. With this reasoning, the tedium of substitution may be avoided.

3.4.2. *QR Decomposition.* Given a full-rank matrix, $A = [\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n]$, we can factorise it by first applying the Gram-Schmidt process on its columns. The columns may then be expressed as linear combinations of the resulting orthonormal basis,

$$\begin{aligned}
\mathbf{a}_1 &= (\mathbf{e}_1 \cdot \mathbf{a}_1)\mathbf{e}_1 \\
\mathbf{a}_2 &= (\mathbf{e}_1 \cdot \mathbf{a}_2)\mathbf{e}_1 + (\mathbf{e}_2 \cdot \mathbf{a}_2)\mathbf{e}_2 \\
\mathbf{a}_3 &= (\mathbf{e}_1 \cdot \mathbf{a}_3)\mathbf{e}_1 + (\mathbf{e}_2 \cdot \mathbf{a}_3)\mathbf{e}_2 + (\mathbf{e}_3 \cdot \mathbf{a}_3)\mathbf{e}_3 \\
\vdots & \qquad\qquad\qquad\qquad \vdots
\end{aligned}$$

Thus, we may write the factorisation $A = QR$, where $Q = [\mathbf{e}_1, \mathbf{e}_2, \dots \mathbf{e}_k]$ and

$$R = \begin{pmatrix} \mathbf{e}_1 \cdot \mathbf{a}_1 & \mathbf{e}_1 \cdot \mathbf{a}_2 & \mathbf{e}_1 \cdot \mathbf{a}_3 & \cdots \\ 0 & \mathbf{e}_2 \cdot \mathbf{a}_2 & \mathbf{e}_2 \cdot \mathbf{a}_3 & \cdots \\ 0 & 0 & \mathbf{e}_3 \cdot \mathbf{a}_3 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}.$$

Thus, in order to solve $\mathbf{Ax} = \mathbf{b}$ for full rank $\mathbf{A}$, we write $\mathbf{QRx} = \mathbf{b} \implies \mathbf{Rx} = \mathbf{Q}^T\mathbf{b}$ (since orthonormality implies $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$), which is directly solvable by back-substitution as $\mathbf{R}$ is upper-triangular. This technique can be used to solve the least-squares problem in closed-form, where the Gram matrix is positive semi-definite.

3.4.3. *Cholesky Decomposition.* In the special case of a Hermitian, positive-definite matrix, we can perform a Cholesky[11] decomposition faster than the previous methods. Note that a Hermitian matrix, named after French mathematician, Charles Hermite (1822-1901), is a matrix of complex numbers equal to its transpose in its real parts, and complex conjugate in imaginary parts. If a Hermitian matrix is purely real-valued, it is simply equal to its transpose. If a matrix $\mathbf{A}$ is Hermitian positive-definite, its Cholesky decomposition is,

$$\mathbf{A} = \mathbf{L}\mathbf{L}^*,$$

where $\mathbf{L}$ is a lower triangular matrix and $\mathbf{L}^*$ is its conjugate transpose. Deriving equations for the elements of $\mathbf{L}$ is a simple exercise of multiplying a lower-triangular matrix of unknowns with its transpose, equating it with $\mathbf{A}$ and solving the resulting equations by substitution. Solving a linear system can then be derived in the usual way using back-substitution. Symmetric positive-definite matrices arise in machine learning models, thus Cholesky decomposition and the related, iterative *conjugate gradient method* are of interest.

[11] André-Louis Cholesky was a French mathematician (1875-1918).

3.4.4. *The Conjugate Gradient Method.* Two vectors, $\mathbf{p}_i$ and $\mathbf{p}_j$ are said to be conjugate with respect to a matrix $\mathbf{A}$ if $\mathbf{p}_i\mathbf{A}\mathbf{p}_j = 0$. A set of N mutually conjugate vectors forms a basis for $\mathbb{R}^N$. For a given equation, $\mathbf{A}\mathbf{x} = \mathbf{b}$, we can write the solution in terms of the conjugate set,

$$\mathbf{x}^* = \sum_{n=1}^{N} \alpha_n \mathbf{p}_n,$$

for a set of constants α_n. Thus, for a given $\mathbf{p}_k$, $\mathbf{p}_k^T\mathbf{A}\mathbf{x}^* = \sum_{n=1}^{N} \alpha_n \mathbf{p}_k^T\mathbf{A}\mathbf{p}_n = \mathbf{p}_k^T\alpha_k\mathbf{p}_k$, giving the rule, $\alpha_k = \frac{\mathbf{p}_k^T\mathbf{b}}{\mathbf{p}_k^T\mathbf{p}_k}$. It is possible to construct a conjugate set on the fly, following a pattern similar to that in Gram-Schmidt orthonormalisation, in the framework of an iterative algorithm. For a large system, this can be used to find an approximate solution consisting of only the first few conjugate vectors. In machine learning, this method is potentially an alternative to gradient descent.

4. Vector Calculus

Vector calculus is an important tool for understanding multivariate probability distributions.

4.1. **Preliminary Results.**

4.1.1. *Sums and Matrices.* It is useful to note how certain matrix expressions may be written as sums. There are many occasions in machine learning where linear models are written interchangeably using summation notation and with vectors. Consider the simplest case–the dot or inner product[12],

$$\mathbf{a}\cdot\mathbf{b} = \mathbf{a}^T\mathbf{b} = a_1b_1 + a_2b_2 + \cdots + a_nb_n = \sum_i a_ib_i.$$

Geometrically, a dot product relates to how one vector projects onto another. From the cosine formula, $(\mathbf{a}\cdot\mathbf{b})/|\mathbf{b}| = |\mathbf{a}|\cdot\cos\theta$, and $|\mathbf{a}|\cdot\cos\theta$ is the length along the vector $\mathbf{b}$ at which point the perpendicular projection of $\mathbf{a}$ intersects. Now consider a matrix-vector multiplication,

$$\mathbf{A}\mathbf{b} = \begin{bmatrix} A_{:,1}, A_{:,2}, \ldots, A_{:,N} \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = b_1A_{:,1} + b_2A_{:,2} + \cdots + b_NA_{:,N} = \sum_i b_iA_{:,i},$$

which represents a projection onto the hyperplane defined by the span of $\mathbf{A}$. Note that if instead we sum over rows rather than columns of $\mathbf{A}$, we have,

[12]Not to be confused with the outer product, which is simply $\mathbf{a}\mathbf{b}^T$, nor with the cross product, which is used to find a vector normal to the plane spanned by two initial vectors.

$$\sum_i b_i \mathbf{a}_i = \mathbf{A}^T \mathbf{b}.$$

Now consider a matrix product,

$$\mathbf{AB} = \left[A_{:,1}, A_{:,2}, \ldots, A_{:,N}\right] \begin{bmatrix} \mathbf{b}_1^T \\ \mathbf{b}_2^T \\ \vdots \\ \mathbf{b}_n^T \end{bmatrix} = \sum_i A_{:,i} \mathbf{b}_i^T.$$

Again working backwards, should we have rows rather than columns, we have the rule,

$$\sum_i \mathbf{a}_i \mathbf{b}_i^T = \mathbf{A}^T \mathbf{B}.$$

4.1.2. *Transposition Chain Rule.* For matrices $\mathbf{A}$ and $\mathbf{B}$, the elements of their product, $\mathbf{AB}$, are formed by multiplying each row of $\mathbf{A}$ with each column of $\mathbf{B}$. Thus, $(\mathbf{AB})_{ij} = A(i,:) \cdot B(:,j)$, that is, the dot product of row i of matrix $\mathbf{A}$ with column j of matrix $\mathbf{B}$. Clearly, $C^T(i,j) = C(j,i) = A(j,:) \cdot B(:,i) = B^T(i,:) \cdot A^T(:,j) = (B^T A^T)(i,j)$, from which the rule,

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T.$$

It is further clear that due to associativity, $(\mathbf{ABC})^T = (\mathbf{A}(\mathbf{BC}))^T = (\mathbf{BC})^T \mathbf{A}^T = \mathbf{C}^T \mathbf{B}^T \mathbf{A}^T$. From this, the chain rule,

$$(\mathbf{A}_1 \mathbf{A}_2 \cdots \mathbf{A}_n)^T = \mathbf{A}_n^T \mathbf{A}_{n-1}^T \cdots \mathbf{A}_1^T.$$

4.1.3. *Inversion Chain Rule.* Given invertible square matrices $\mathbf{A}$ and $\mathbf{B}$, it is clear that if we form the expression, $\mathbf{ABB}^{-1}\mathbf{A}^{-1} = \mathbf{AIA}^{-1} = \mathbf{I}$, then $\mathbf{B}^{-1}\mathbf{A}^{-1}$ is the right inverse of $\mathbf{AB}$. It is further clear that this is also the left inverse. We have therefore,

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1} \mathbf{A}^{-1}.$$

Just as with transposition, we see that due to associativity $(\mathbf{ABC})^{-1} = (\mathbf{A}(\mathbf{BC}))^{-1} = (\mathbf{BC})^{-1}\mathbf{A}^{-1} = \mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1}$, and in general,

$$(\mathbf{A}_1 \mathbf{A}_2 \cdots \mathbf{A}_n)^{-1} = \mathbf{A}_n^{-1} \mathbf{A}_{n-1}^{-1} \cdots \mathbf{A}_1^{-1}.$$

4.1.4. *Trace Cyclic Permutation Property.* The trace of a matrix product is $\mathrm{tr}(\mathbf{AB}) = \sum_{j=1}^N \sum_{i=1}^N a_{ij} b_{ji}$, that is, the sum of the diagonal elements of the product. First consider the product,

$$\operatorname{tr}(\mathbf{ABC}) = \operatorname{tr}\left(\begin{bmatrix} \sum_{j=1}^{N} c_{j1} \sum_{i=1}^{N} a_{1i} b_{ij} & \cdots & \cdots \\ \cdots & \sum_{j=1}^{N} c_{j2} \sum_{i=1}^{N} a_{2i} b_{ij} & \cdots \\ & & \ddots \\ \cdots & \cdots & \sum_{j=1}^{N} c_{jN} \sum_{i=1}^{N} a_{Ni} b_{ij} \end{bmatrix}\right)$$
$$= \sum_{k=1}^{N} \sum_{j=1}^{N} \sum_{i=1}^{N} a_{ki} b_{ij} c_{jk}.$$

Now consider the trace of the permutation of this matrix product,

$$\begin{aligned} \operatorname{tr}(\mathbf{CAB}) &= \sum_{k=1}^{N} \sum_{j=1}^{N} \sum_{i=1}^{N} a_{ji} b_{ik} c_{kj} \\ &= \operatorname{tr}(\mathbf{ABC}), \end{aligned}$$

where j and k are swapped. This result is called the trace cyclic permutation property, as it shows that $\operatorname{tr}(\mathbf{ABC}) = \operatorname{tr}(\operatorname{ROR}(\mathbf{ABC}))$. A related result is known as the *trace trick*. Note that $\mathbf{x}^T\mathbf{Ax} = \operatorname{tr}(\mathbf{x}^T\mathbf{Ax})$, since the result of the product is a 1×1 matrix (i.e. a scalar). By the permutation property we have, $\mathbf{x}^T\mathbf{Ax} = \operatorname{tr}(\mathbf{xx}^T\mathbf{A}) = \operatorname{tr}(\mathbf{Axx}^T)$.

4.2. **Vector Calculus.** The foremost definition in vector calculus is the vector derivative or gradient. In general, the gradient of a multi-variate function is the vector of partial derivatives, that is,

$$\nabla \mathbf{f} \triangleq \frac{\partial f}{\partial (x_1, \ldots, x_n)} = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right].$$

The gradient at a point of a multivariate function is the vector lying normal to the *level curve*, the contour on the hyper surface for which the function is constant. From our definition of the gradient, it is easy to derive some of the fundamental formulas of vector calculus. For function $f : \mathbb{R}^n \to \mathbb{R}^m$, that is, a vector-valued function in n variables, its derivative, known as the Jacobian $\mathbf{J}$, is,

$$\mathbf{J} \triangleq \frac{\partial (f_1, \ldots, f_m)}{\partial (x_1, \ldots, x_n)} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix},$$

that is, the matrix of partial derivatives corresponding to each of the input and output pairs. In the case that f is scalar-valued, the Jacobian becomes the row vector given above. The Jacobian features as the first-order term in the Taylor expansion of a multivariate function. For a *scalar*-valued function, f, the Hessian matrix is the matrix of all second order partial derivatives, written,

$$\mathbf{H} \triangleq \nabla^2 \mathbf{f} = \mathbf{J}(\nabla \mathbf{f}(\mathbf{x})) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}.$$

Note that unlike the Jacobian, the Hessian is not defined for vector-valued functions.

4.2.1. *Differentiating a Dot Product.* Following our rule from above, we have,

$$\frac{\partial(\mathbf{b}^T\mathbf{a})}{\partial \mathbf{a}} = \begin{bmatrix} \frac{\partial}{\partial a_1}\mathbf{b}^T\mathbf{a} \\ \frac{\partial}{\partial a_2}\mathbf{b}^T\mathbf{a} \\ \vdots \\ \frac{\partial}{\partial a_n}\mathbf{b}^T\mathbf{a} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \mathbf{b}.$$

Of course, if we replace $\mathbf{b}$ with $\mathbf{a}$, we are differentiating the squared l_2 norm[13] of $\mathbf{a}$ $\frac{\partial(\mathbf{a}^T\mathbf{a})}{\partial \mathbf{a}} = 2\mathbf{a}$.

4.2.2. *Differentiating a Vector Quadratic.* Beginning with a single element we have, $\partial(\mathbf{a}^T\mathbf{A}\mathbf{a})/\partial a_1 = 2A_{i,i}a_i + \sum_{j\neq i} A_{ij}a_j + \sum_{j\neq i} A_{ji}a_j = \mathbf{a}^T A(i,:) + \mathbf{a}^T A(:,i)$. Thus,

$$\frac{\partial(\mathbf{a}^T\mathbf{A}\mathbf{a})}{\partial \mathbf{a}} = \begin{bmatrix} \frac{\partial}{\partial a_1}\mathbf{a}^T\mathbf{A}\mathbf{a} \\ \frac{\partial}{\partial a_2}\mathbf{a}^T\mathbf{A}\mathbf{a} \\ \vdots \\ \frac{\partial}{\partial a_n}\mathbf{a}^T\mathbf{A}\mathbf{a} \end{bmatrix} = \begin{bmatrix} \mathbf{a}^T A(1,:) + \mathbf{a}^T A(:,1) \\ \mathbf{a}^T A(2,:) + \mathbf{a}^T A(:,2) \\ \vdots \\ \mathbf{a}^T A(n,:) + \mathbf{a}^T A(:,n) \end{bmatrix} = \mathbf{a}^T\mathbf{A}^T + \mathbf{a}^T\mathbf{A}$$
$$= (\mathbf{A} + \mathbf{A}^T)\mathbf{a}$$

4.2.3. *Trace Formula.* Considering firstly the partial derivative of a single element, it is clear that,

$$\frac{\partial}{\partial A_{ij}}\big(\mathrm{tr}(\mathbf{B}\mathbf{A})\big) = \frac{\partial}{\partial A_{ij}}\big(\mathbf{B}(1,:)\mathbf{A}(:,1) + \mathbf{B}(2,:)\mathbf{A}(:,2) + \cdots + \mathbf{B}(n,:)\mathbf{A}(:,n)\big) = B_{ji}.$$

Thus, it is clear that in general,

$$\frac{\partial}{\partial A}(\mathrm{tr}(\mathbf{B}\mathbf{A})) = \mathbf{B}^T.$$

[13]The l_2 norm of a vector, $\mathbf{a}$, is $|\mathbf{a}|_2 = \sqrt{\sum_i a_i^2}$. The l_1 norm is $|\mathbf{a}|_1 = \sum_i |a_i|$.

4.2.4. *Log Formula.* Considering firstly the partial derivative of a single element, it is clear that,

$$\frac{\partial}{\partial A_{ij}} \log |\mathbf{A}| = \frac{1}{|\mathbf{A}|} C_{ij},$$

where C_{ij} is row i, column j, of the cofactor matrix, $\mathbf{C}$, the matrix whose elements are each of the sub-determinants used to calculate the determinant of $\mathbf{A}$. Noting the general inverse formula, $\mathbf{A}^{-1} = \frac{1}{|\mathbf{A}|}\mathbf{C}^T$, we have,

$$\frac{\partial}{\partial \mathbf{A}} \log |\mathbf{A}| = \frac{1}{|\mathbf{A}|}\mathbf{C} = (\mathbf{A}^T)^{-1}.$$

5. Series

This section introduces series and infinite sums. Notably, it presents Taylor series, a technique with many uses in the material to come. The section culminates in deriving an infinite sum for π.

5.1. **Geometric Series.** A geometric series[14] is a series, s, of the form,

$$s = a + ar + ar^2 + ar^3 + \cdots + ar^{n-1} = \sum_{k=0}^{n-1} ar^k,$$

for some choice of a, r, and n. Multiplying this sum by the factor r and adding a gives,

$$rs + a = a + ar + arr + ar^2 r + \cdots + ar^{n-1} r = s + ar^n.$$

Therefore,

$$s = a\frac{1 - r^n}{1 - r},$$

$r \neq 1$. Thus we have a closed-form expression[15] for the value of a geometric series, which will converge provided $|r| < 1$.

5.2. **Taylor Series.** A Taylor series, named after English mathematician Brook Taylor (1685-1731), is a technique for writing an infinitely differentiable function[16], $f(x)$, as an infinite order polynomial,

$$\begin{aligned} f(x) &= \sum_{i=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n \\ &= f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots, \end{aligned}$$

[14] A series is the sum of a sequence of terms.

[15] A closed-form expression is one that may be evaluated in a finite number of operations.

[16] A function is infinitely differentiable at a point if all its derivatives exist at that point.

for some choice of a. A Taylor series is an example of a power series. When $a = 0$, we have what is called a Maclaurin series, named after Scottish mathematician Colin Maclaurin (1698 - 1746). The first n terms of the series may then be used to compute an approximation to the function at a point. For example, a calculator (which is only capable of addition and multiplication) will use a Taylor series approximation for its trigonometric functions. As we will see, Taylor series allow us to manipulate functions that have no closed form, and are in fact the most important technique to our analysis.

5.2.1. *Taylor's Theorem.* Taylor series follow from Taylor's theorem, which quantifies the remainder term for a kth order Taylor polynomial of a function. The theorem states that given a function $f : \mathbb{R} \to \mathbb{R}$ is k times differentiable at point $a \in \mathbb{R}$, then there exists a function $h_k : \mathbb{R} \to \mathbb{R}$ such that,

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f(k)(a)}{k!}(x-a)^k + h_k(x)(x-a)^k,$$

where

$$\lim_{a \to 0} h_k(a) = 0.$$

To show this, define $h_k(x) = \frac{f(x)-P(x)}{(x-a)^k}$ for $x \neq a$ and $h_k(x) = 0$ for $x = 0$, where $P(x)$ is the order k Taylor polynomial for $f(x)$ centred on a. First note that $f^{(j)}(a) = P^{(j)}(a)$ for $j = 1, \ldots k$. Applying L'Hôpital's rule[17] repeatedly,

$$\begin{aligned}
\lim_{x \to a} \frac{f(x) - P(x)}{(x-a)^k} &= \lim_{x \to a} \frac{\frac{d^k}{dx^k}(f(x) - P(x))}{\frac{d^k}{dx^k}(x-a)^k} \\
&= \frac{1}{k!} \lim_{x \to a} \frac{f^{(k-1)}(x) - P^{(k-1)}(x)}{x-a} \\
&= \frac{1}{k!} \lim_{x \to a} \frac{f^{(k-1)}(x) - f^{(k-1)}(a)}{x-a} - \frac{P^{(k-1)}(x) - P^{(k-1)}(a)}{x-a} \\
&= \frac{1}{k!}(f^{(k)}(a) - P^{(k)}(a)) \qquad (1) \\
&= 0,
\end{aligned}$$

where (1) comes from the definition of a derivative. Thus, we have shown the error term converges to 0 as $x \to a$.

[17] L'Hôpital's rule, named after French mathematician Guillaume de L'Hôpital (1661-1704), is that $\lim_{x \to a} f(x)/g(x) = \lim_{x \to a} f'(x)/g'(x)$. The rule is useful for evaluating functions with *indeterminate forms*, that is, involving divisions by 0 or infinities. A typical use case is in evaluating the sinc function at 0. The rule might also illustrate the fact that higher order functions ultimately overwhelm lower order functions as $x \to \infty$, regardless of their coefficients, a key point in complexity analysis.

5.2.2. *Convergence.* Holomorphic functions are a key interest in complex analysis, the study of functions of complex variables. Holomorphic functions are functions that are complex differentiable at every point on their domain. A consequence of complex differentiability is that the function is both infinitely differentiable and equal to its own Taylor series on that domain. A major result from complex analysis is that all homomorphic functions are analytic. A function is analytic if and only if its Taylor series converges to the function in some neighbourhood of points around x_0 for every x_0 in the function's domain. That is, there is always a locally convergent power series for the function, and so a local approximation can always be made around any point on the domain. The radius of convergence of a power series,

$$f(z) = \sum_{n=0}^{\infty} c_n (z-a)^n$$

defined around a point a is r, defined to be,

$$r = \sup \left\{ |z-a| \;\middle|\; \sum_{n=0}^{\infty} c_n (z-a)^n \text{ converges} \right\},$$

beyond which the series diverges. An elementary function is a function in one variable expressible in terms of a finite number of arithmetic operators, exponentials, logarithms and roots. This includes trigonometric and hyperbolic functions, as these are expressible as complex exponential functions. Elementary functions are analytic at all but a finite number of points (singularities). Functions that are analytic everywhere (on all complex points) are said to be *entire* functions, and have no singularities. These include polynomials (whose Taylor expansions are of course equal to themselves), trigonometric, hyperbolic, and exponential functions, and many others. Such functions have an infinite radius of convergence.

5.2.3. *Taylor Expansions of Sine and Cosine.* In the analysis to come, we will use the Taylor series for $\sin x$,

$$\begin{aligned}
\sin x &= \sin 0 + \frac{\cos 0}{1!}(x-0) - \frac{\sin 0}{2!}(x-0)^2 - \frac{\cos 0}{3!}(x-0)^3 + \cdots \\
&= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \\
&= \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1},
\end{aligned}$$

as well as for $\cos x$,

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}.$$

These are entire functions, hence the Taylor series converge for all real numbers.

5.2.4. *Bernoulli Numbers.* The Bernoulli numbers are an infinite sequence of numbers that appear sufficiently often in number theory to merit a name (as do π and e). They originate as the coefficients of the Bernoulli polynomials, but make various other appearances in number theory. They were first discovered in Europe[18] by Swiss mathematician, Jakob Bernoulli (1655-1705), a member of the eminent Bernoulli family that produced several noted mathematicians. The first few Bernoulli numbers are,

$$B_0 = 1, B_1 = \pm\frac{1}{2}, B_2 = \frac{1}{6}, B_3 = 0, B_4 = -\frac{1}{30}, B_5 = 0, B_6 = \frac{1}{42}, B_7 = 0, B_8 = -\frac{1}{30}.$$

Curiously, the second Bernoulli number, B_1, may be plus or minus $\frac{1}{2}$. Therefore, there are strictly speaking *two* sequences, the *first* Bernoulli numbers (with $B_1 = -\frac{1}{2}$) and the *second* Bernoulli numbers (with $B_1 = \frac{1}{2}$). The numbers may be defined by a recurrence relation. For example, the second Bernoulli numbers are related by,

$$B_n = 1 - \sum_{k=0}^{n-1} \binom{n}{k} \frac{B_k}{n-k+1}, B_0 = 1.$$

The Bernoulli numbers appear within some Taylor expansions, such as that of the hyperbolic tangent function, $\tanh x$,

$$\tanh x = x - \frac{1}{3}x^3 + \frac{2}{15}x^5 - \frac{17}{315}x^7 + \cdots = \sum_{n=1}^{\infty} \frac{B_{2n} 4^n (4^n - 1)}{(2n)!} x^{2n-1}.$$

The Bernoulli numbers also appear in the Euler-Maclaurin formula, which specifies the error incurred by approximating integrals with sums, such as in the trapezoidal rule.

5.3. π as an Infinite Series.

5.3.1. *Pythagorean Trigonometric Identity.* From the Pythagorean theorem, we may derive an important trigonometric identity. Given a right-angle triangle, we denote the sides a, b, c such that $\cos x = \frac{a}{c}$ and $\sin x = \frac{b}{c}$. Note that this implies that,

$$\tan x = \frac{a}{b} = \frac{\sin x}{\cos x}.$$

Now, dividing the Pythagorean equation by c^2 gives,

$$\frac{a^2}{c^2} + \frac{b^2}{c^2} = \frac{c^2}{c^2},$$

hence,

$$\cos^2 x + \sin^2 x = 1.$$

[18]The Bernoulli numbers were independently discovered by the great Japanese mathematician Seki Takakazu around the same time. It is unknown which discovery came first.

From this it may be seen that the coordinates $(\cos x, \sin x)$ sketch a unit circle on the real plane.

5.3.2. *Differentiating Inverse Trigonometric Functions.* The function, $\arctan x$ (otherwise written $\tan^{-1} x$) is an inverse trigonometric function, the inverse of the function $\tan x$. If we let $y = \arctan x$, then we have $\tan y = x$. Differentiating both sides gives,

$$\frac{d}{dx} \tan y = 1.$$

Now,

$$\frac{d}{dx} \tan y = \frac{dy}{dx}\frac{d}{dy} \tan y = \frac{dy}{dx}\frac{1}{\cos^2 y},$$

so,

$$\frac{dy}{dx} = \cos^2(\arctan x).$$

Rearranging the Pythagorean trigonometric identity, we have $\cos^2 x = 1 - \sin^2 x$, and squaring and rearranging our other identity, $\sin^2 x = \cos^2 x \tan^2 x$. By substitution we obtain, $\cos^2 x = \frac{1}{1+\tan^2 x}$. Thus, we have,

$$\begin{aligned} \frac{d}{dx} \arctan x &= \frac{1}{1 + \tan^2(\arctan x)} \\ &= \frac{1}{1 + x^2} \end{aligned}$$

5.3.3. *An Infinite Series For π.* From the previous result, we have

$$\arctan x = \int \frac{1}{1 + x^2} dx.$$

Replacing the integrand with its Maclaurin series gives,

$$\begin{aligned} \arctan x &= \int \sum_{n=0}^{\infty} (-1)^n x^{2n} dx \\ &= \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} x^{2n+1}. \end{aligned}$$

Now, we know that an isosceles right triangle has opposite and adjacent sides of equal length. Hence, $\arctan(1) = \pi/4$. Evaluating our infinite series at $x = 1$ gives,

$$\pi/4 = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

Though pretty, this series is extremely slow to converge. Other series representations for π exist, deriving from other trigonometric identities.

6. Infinite Products

Infinite products are defined for a sequence of numbers, $a_1, a_2, a_3, \ldots$, as,

$$\prod_{n=1}^{\infty} a_n = a_1 a_2 a_3 \ldots$$

Just like infinite sums they may converge or diverge. The product converges if the limit of the partial product (the first n factors) exists as $n \to \infty$. In this section we present some of the more stupendous mathematical results involving infinite products.

6.1. **Viète's Formula.** The first recorded infinite product was formulated by French mathematician François Viète (1540-1603), and gives an expression for π. The expression derives from an old method for approximating the area of a circle: suppose we have a circle with radius 1. Its area is π units. Suppose then that we inscribe a square within this circle. The area of the square (4-sided polygon) is $(\sqrt{2})^2 = 2$. This gives a first, rough approximation to the area of the circle. If we then inscribe a hexagon (8-sided polygon) in the circle, it is clear we get a better approximation. A hexadecahedron (16-sided polygon) would give a better approximation again, and so on. As the number of sides goes to infinity, the inscribed shapes 'telescope' to the shape of the circle.

We can therefore derive an expression for the area of the circle, π, by taking the area of the square and scaling it by the ratio of the area of the octagon to the area of the square, and scaling this result by the ratio of hexadecahedron to octagon, icosidodecahedron to hexadecahedron, and so on. We choose shapes whose number of sides are powers of two because there is a useful relation between these, as we shall see. For brevity, we will denote the area of the 2^n-sided polygon as A_n. Thus, A_2 is the area of the square, A_3 the octagon, and so on. Our technique for calculating the area of the circle is therefore described by,

$$\pi = 2 \cdot \frac{A_3}{A_2} \cdot \frac{A_4}{A_3} \cdot \frac{A_5}{A_4} \cdot \ldots$$

We will rearrange this as,

$$\frac{2}{\pi} = \frac{A_2}{A_3} \cdot \frac{A_3}{A_4} \cdot \frac{A_4}{A_5} \cdot \ldots$$

Now, it can be seen that a 2^n-sided polygon is an arrangement of 2^n equal isosceles triangles, with arm length 1 and angle $2\pi/2^n$. Applying the formula for the area of an isosceles triangle gives, $A_n = 2^n \cdot \frac{1}{2}\sin(\pi/2^{n-1})$. Thus, we have,

$$\frac{2}{\pi} = \frac{1}{2\sin(\pi/4)} \cdot \frac{\sin(\pi/4)}{2\sin(\pi/8)} \cdot \frac{\sin(\pi/8)}{2\sin(\pi/16)} \cdot \ldots$$

Applying the trigonometric identity, $\sin 2x = 2\sin x \cos x$, our expression simplifies to,

$$\frac{2}{\pi} = \frac{\sqrt{2}}{2} \cdot \cos(\pi/8) \cdot \cos(\pi/16) \cdot \cos(\pi/32) \cdot \ldots$$

A useful trigonometric identity can be derived from the angle sum identity, $\cos(\alpha + \beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta$. Namely that $\cos x = \cos^2(x/2) - \sin^2(x/2)$. Applying the pythagorean identity, $\cos x = 2\cos^2(x/2) - 1$. Finally, rearranging gives,

$$\cos\left(\frac{x}{2}\right) = \pm\sqrt{\frac{1+\cos x}{2}}.$$

Exploiting this identity gives us a recurrence relationship from $\cos x$ to $\cos(x/2)$, finally yielding the startling,

$$\frac{2}{\pi} = \frac{\sqrt{2}}{2}\cdot\frac{\sqrt{2+\sqrt{2}}}{2}\cdot\frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2}\cdots.$$

6.2. **The Basel Problem.** In 1734, Leonhard Euler[19], found the elegant solution to the sum,

$$\sum_{n=0}^{\infty}\frac{1}{n^2} = 1 + \frac{1}{4} + \frac{1}{9} + \cdots,$$

a problem posed 90 years earlier by Italian mathematician, Pietro Mengoli (1626-1686). Its solution had previously defied the attempts of the eminent Bernoulli dynasty of mathematicians. The problem is named after Euler's hometown, and first place of employment in Switzerland. The solution involves infinite products, and a mathematical leap of faith, that was only later confirmed rigorously. Incidentally, this sum is the value of $\zeta(2)$, where $\zeta(s)$ is the then undiscovered Riemann zeta function[20]. Euler's solution starts by considering the sinc function,

$$\begin{aligned}\frac{\sin x}{x} &= \frac{x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots}{x}\\ &= 1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \cdots\end{aligned} \tag{2}$$

Having written the infinite series expansion of this function, it is obvious that the limit, $\lim_{x\to 0}\sin x/x = 1$[21]. Euler then found an infinite *product* expansion for the same function,

[19]Euler (1707-1783) was a Swiss mathematician, considered one of the greatest of all time.

[20]The Riemann zeta function, defined over the complex numbers, formulated by Bernhard Riemann (1826-1866) has proven to be of great mathematical interest. The Riemann hypothesis, which conjectures the placement of zeta function roots has great implications for number theory, particularly the distribution of prime numbers. The hypothesis is the subject of one of the seven Clay Institute Millennium problems.

[21]As noted previously, this can also be found using L'Hôpital's rule.

$$\begin{aligned}\frac{\sin x}{x} &= \prod_{k=1}^{\infty} 1 - \frac{x^2}{k^2\pi^2},\\ &= \Big(1-\frac{x}{\pi}\Big)\Big(1+\frac{x}{\pi}\Big)\Big(1-\frac{x}{2\pi}\Big)\Big(1+\frac{x}{2\pi}\Big)\Big(1-\frac{x}{3\pi}\Big)\Big(1+\frac{x}{3\pi}\Big)\cdots\end{aligned}$$

reasoning that the infinitely many roots $(\pm\pi, \pm 2\pi, \pm 3\pi, \dots)$ of the left- and right-hand functions were the same, hence the functions themselves. This was not an entirely rigorous piece of reasoning, but it was later confirmed to be correct[22]. Multiplying this product out shows that the coefficient for x^2 is,

$$-\frac{1}{\pi^2}\Big(1+\frac{1}{4}+\frac{1}{9}+\cdots\Big) = -\frac{1}{\pi^2}\sum_{n=0}^{\infty}\frac{1}{n^2}.$$

But we know from (2) that the coefficient for x^2 is $-\frac{1}{3!}$, hence, equating the two gives the solution to the Basel problem,

$$\sum_{n=1}^{\infty}\frac{1}{n^2} = \frac{\pi^2}{6}.$$

6.3. **Wallis' Product.** Wallis's product is an infinite product expression for π discovered by the English mathematician John Wallis (1616-1703). It can be derived starting from the same expression from the Basel problem,

$$\frac{\sin x}{x} = \prod_{k=1}^{\infty} 1 - \frac{x^2}{k^2\pi^2}.$$

If we let $x = \pi/2$ we have,

$$\begin{aligned}\frac{2}{\pi} &= \prod_{k=1}^{\infty} 1 - \frac{1}{4k^2}\\ \implies \frac{\pi}{2} &= \prod_{k=1}^{\infty}\frac{4k^2}{4k^2-1}\\ &= \prod_{k=1}^{\infty}\frac{2k}{2k-1}\cdot\frac{2k}{2k+1}\\ &= \frac{2}{1}\cdot\frac{2}{3}\cdot\frac{4}{3}\cdot\frac{4}{5}\cdot\frac{6}{5}\cdot\frac{6}{7}\cdots\end{aligned}$$

If we consider the partial product, we derive a related expression,

[22]See the Weierstrass factorisation theorem.

$$\begin{aligned} p_k &= \prod_{n=1}^{k} \frac{2n}{2n-1} \cdot \frac{2n}{2n+1} \\ &= \frac{1}{2k+1} \prod_{n=1}^{k} \frac{(2n)^4}{[2n(2n-1)]^2} \\ &= \frac{1}{2k+1} \cdot \frac{2^{4k}(k!)^4}{[(2k)!]^2} \to \frac{\pi}{2}, \end{aligned}$$

as $k \to \infty$.

6.4. **The Method of Eratosthenes.** The method of Eratosthenes is a prime number sieve algorithm discovered by Eratosthenes of Cyrene (c. 276 BC - 195/194 BC). It is central to the derivation of the Euler product formula. It is a method for identifying all primes up to some natural number, N. For example, let $N = 15$, so that we have the list of candidate primes,

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15$$

2 must be prime as there are no smaller numbers in the list to divide it. We can then eliminate all multiples of 2, as they are composite by definition,

$$2, 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, 9, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, 15$$

3 was not eliminated by this step, and as the new smallest candidate, it must be prime. Likewise, we may eliminate all multiples of 3,

$$2, 3, \cancel{4}, 5, \cancel{6}, 7, \cancel{8}, \cancel{9}, \cancel{10}, 11, \cancel{12}, 13, \cancel{14}, \cancel{15}$$

Continuing in this way (though by now all non-primes have been eliminated), we find 2, 3, 5, 7, 11, and 13 are the primes up to 15.

6.5. **The Euler Product Formula.** The Riemann zeta function analytically continues the infinite sum,

$$\zeta(s) = \sum_{i=1}^{\infty} \frac{1}{n^s} = 1 + \frac{1}{2^s} + \frac{1}{3^s} + \frac{1}{4^s} + \frac{1}{5^s} + \cdots, \tag{3}$$

for complex numbers, s, with $\mathrm{Re}(s) > 1$. The case of $s = 2$ is the form of the Basel problem. Consider,

$$\frac{1}{2^s}\zeta(s) = \frac{1}{2^s} + \frac{1}{4^s} + \frac{1}{6^s} + \frac{1}{8^s} + \frac{1}{10^s} + \cdots \tag{4}$$

Subtracting (4) from (3) removes all multiples of 2 leaving,

$$\Big(1-\frac{1}{2^s}\Big)\zeta(s) = 1+\frac{1}{3^s}+\frac{1}{5^s}+\frac{1}{7^s}+\frac{1}{9^s}+\cdots \tag{5}$$

Multiplying (5) by $1/3^s$ and subtracting it from itself gives,

$$\Big(1-\frac{1}{3^s}\Big)\Big(1-\frac{1}{2^s}\Big)\zeta(s) = 1+\frac{1}{5^s}+\frac{1}{7^s}+\frac{1}{11^s}+\frac{1}{13^s}+\cdots$$

If we continue in this fashion, eliminating the primes and all their remaining multiples, we are following the method of Eratosthenes, leaving,

$$\cdots\Big(1-\frac{1}{11^s}\Big)\Big(1-\frac{1}{7^s}\Big)\Big(1-\frac{1}{5^s}\Big)\Big(1-\frac{1}{3^s}\Big)\Big(1-\frac{1}{2^s}\Big)\zeta(s) = 1.$$

Dividing through leaves,

$$\zeta(s) = \sum_{i=1}^{\infty}\frac{1}{n^s} = \prod_{p \text{ prime}}\left(\frac{1}{1-p^{-s}}\right),$$

which is an infinite product, since we know from Euclid that there are infinitely many primes.

7. Deriving the Golden Ratio

This section presents the concept of continued fractions and the distinctive continued fraction expression for the golden ratio.

7.1. **Continued Fractions.** Continued fractions are a way of specifying fractional numbers as a sequence of nested fractions,

$$a_0+\cfrac{1}{a_1+\cfrac{1}{a_2+\cfrac{1}{\ddots+\cfrac{1}{a_n}}}},$$

where $a_0, a_1, \ldots, a_n$ are integers. For example, take the fraction, $f = \frac{387}{259}$. We can then write, $f = 1+\frac{128}{259} = 1+\frac{1}{259/128}$. Following this procedure,

$$f = \frac{387}{259} = 1+\frac{1}{259/128} = 1+\cfrac{1}{2+\cfrac{1}{128/3}} = 1+\cfrac{1}{2+\cfrac{1}{42+\cfrac{1}{3/2}}} = 1+\cfrac{1}{2+\cfrac{1}{42+\cfrac{1}{1+\cfrac{1}{2}}}}.$$

Standard notation takes the values on the diagonal and writes them as, $f = [1; 2, 42, 1, 2]$.

7.2. **Fibonacci Sequences.** A Fibonacci sequence is a sequence of numbers for initial values, F_0 and F_1, and recurrence relationship,

$$F_n = F_{n-1} + F_{n-2},$$

for $n \geq 2$. If we take the ratio of the $(n+1)$th and nth terms in the sequence we have,

$$\frac{F_{n+1}}{F_n} = \frac{F_n + F_{n-1}}{F_n} = 1 + \frac{1}{F_n/F_{n-1}} = 1 + \cfrac{1}{1 + \cfrac{1}{F_{n-1}/F_{n-2}}} = \cdots = 1 + \cfrac{1}{1 + \cfrac{1}{\ddots + \cfrac{F_0}{F_1}}},$$

the fraction continuing for $n-1$ levels.

7.3. **The Golden Ratio.** The golden ratio is a number with many interesting properties, and with some mythical connections to the natural world. It is the ratio that arises between two numbers, a and b, when $a : b = (a+b) : a$. It is also the limiting value of the ratio of successive terms in a Fibonacci sequence. We may discover the self-referential golden-ratio, φ, by looking at the self-referential infinite continued fraction, $\varphi = [1; 1, 1, 1, \dots]$. That is, the ratio of F_{n+1}/F_n as $n \to \infty$.

$$\varphi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{\ddots}}} = 1 + \frac{1}{\varphi},$$

noting the recursion. Multiplying through gives the quadratic, $\varphi^2 - \varphi - 1 = 0$. Applying the quadratic formula gives,

$$\varphi = \frac{1 \pm \sqrt{5}}{2}.$$

8. The Exponential Function

The exponential function is a transcendental entire function, with its base e a transcendental number[23]. It makes many appearances throughout number theory and statistics.

[23]A transcendental number is one that is not algebraic. An algebraic number is a number that is the root of a non-zero polynomial function with rational coefficients, and is a generalisation of the ancient Greek concept of *constructible* numbers. Another way of putting this is that if a number is transcendental there exists no expression containing a finite number of integers and elementary operations that can express the number. To illustrate, $\sqrt{17/4}$ is not transcendental (though it is irrational) as it is the square root of a quotient of integers. Formally, it would solve the expression, $4x^2 = 17$. It is known that most numbers are transcendental.

8.1. **Deriving the Exponential Function.** The starting point for deriving the exponential function is to consider the differential equation,

$$f(x) = \frac{df}{dx},$$

that is, a function that is its own derivative. This equation models many common physical problems. One solution to this is the infinite series,

$$f(x) = \sum_{n=0}^{\infty} \frac{x^n}{n!},$$

since $f'(x) = \sum_{n=1}^{\infty} \frac{x^{n-1}}{(n-1)!} = \sum_{n=0}^{\infty} \frac{x^n}{n!} = f(x)$. Note that this is the Maclaurin series for a function whose derivatives are all equal to 1 at 0. Evaluating the function at 1 gives,

$$f(1) = \sum_{n=0}^{\infty} \frac{1}{n!} = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots,$$

which must converge since the rate of growth after the second term is inferior to that of geometric series with base $1/2$. The number it converges to is know as $e = \sum_{k=0}^{\infty} \frac{1}{k!} \approx 2.718$. With the binomial theorem, an equivalence may be shown between this infinite sum, and the product,

$$e = \lim_{n \to +\infty} \left(1 + \frac{1}{n}\right)^n.$$

This form is particularly useful in financial modelling. Compound interest gives an expression for the future value of a financial device, F, based on its present value, P, the interest rate, r, and the compounding frequency, n, over t time periods, with $F = P(1 + r/n)^{nt}$. As $n \to \infty$, we have what is called *continuous* compound interest, and the formula clearly converges to $F = Pe^{rt}$.

The closed form of the function may be deduced by considering how it grows. Consider,

$$
\begin{aligned}
f(x+1) &= \sum_{n=0}^{\infty} \frac{(x+1)^n}{n!} \\
&= \sum_{n=0}^{\infty} \sum_{k=0}^{n} \binom{n}{k} \frac{x^k \cdot 1^{n-k}}{n!}, \qquad (6) \\
&= \sum_{n=0}^{\infty} \sum_{k=0}^{n} \frac{x^k}{k!(n-k)!}, \qquad (7) \\
&= \begin{array}{ccccccccc}
\frac{1}{0!\cdot 0!} & + & & & & & & & \\
\frac{1}{0!\cdot 1!} & + & \frac{x}{1!\cdot 0!} & + & & & & & \\
\frac{1}{0!\cdot 2!} & + & \frac{x}{1!\cdot 1!} & + & \frac{x^2}{2!\cdot 0!} & + & & & \\
\frac{1}{0!\cdot 3!} & + & \frac{x}{1!\cdot 2!} & + & \frac{x^2}{2!\cdot 1!} & + & \frac{x^3}{3!\cdot 0!} & + \\
\vdots & & \vdots & & \vdots & & \vdots & & \vdots
\end{array} \\
&= \sum_{n=0}^{\infty} \frac{x^n}{n!} \sum_{k=0}^{\infty} \frac{1}{k!}, \\
&= f(x) \cdot e,
\end{aligned}
$$

where (6) comes from the binomial theorem, and (7) substitutes the binomial coefficient. Therefore, $f(1+\delta)/f(1) = f(\delta)$. Thus, the ratio of growth of the function is fixed for an increase δ in x, regardless of the value of x. Therefore,

$$\frac{f(\delta)}{f(0)} = \frac{f(2\delta)}{f(\delta)} = \cdots = \frac{f(1)}{f((n-1)\delta)},$$

where $\delta = 1/n$, for any choice of n. Multiplying the n terms together gives $f(1/n)^n = f(1)/f(0) = e$. Therefore, $f(x/n)$ is the x/n^{th} root of e. Thus, $f(x/n) = \sqrt[n]{e^x}$. Or simply, $f(x) = e^x$.

8.2. **Other Solutions.** Could any other function solve our differential equation? No, as can be seen from separation of variables,

$$\frac{1}{f} df = dx.$$

Integrating both sides gives

$$\ln f = x + C.$$

So,

$$f = Ce^x.$$

8.3. **Euler's Formula.** Here we take the opportunity to derive a very famous equation[24], based on what we have covered so far. Consider the Maclaurin series,

$$\begin{aligned} e^{ix} &= 1 + ix + \frac{(ix)^2}{2!} + \frac{(ix)^3}{3!} + \frac{(ix)^4}{4!} + \frac{(ix)^5}{5!} + \cdots \\ &= 1 + ix - \frac{x^2}{2!} - i\frac{x^3}{3!} + \frac{x^4}{4!} + i\frac{x^5}{5!} - \cdots \end{aligned}$$

The reader may now confirm that the even terms correspond with $\cos x$, and the odd terms with $i \sin x$. Thus,

$$e^{ix} = \cos x + i \sin x,$$

a fantastic result! Now, just as coordinates $(\cos x, \sin x)$ trace a unit circle on the real plane, the complex numbers arising from $e^{ix} = \cos x + i \sin x$ do so on the complex plane (Figure 1). Thus, we have a basis on which to perform complex exponentiation, for example, $z^{a+bi} = z^a \cdot (e^{\ln z \cdot bi}) = z^a \cdot (\cos(b \ln z) + i \sin(b \ln z))$. Furthermore, for any odd integer k,

$$e^{i(k\pi)} = \cos(k\pi) + i \sin(k\pi) = -1.$$

For the case of $k = 1$, we may rearrange to acquire Euler's identity,

$$e^{i\pi} + 1 = 0,$$

uniting the base of the natural logarithm, e, the unit of imaginary numbers, i, the ratio of a circle's circumference to its diameter, π, the multiplicative identity, 1, and the additive identity, 0, in an equation involving a single sum, product, and exponentiation.

8.3.1. *Angle Sum Identities.* A nice way to derive the trigonometric identities for sums of angles is with Euler's formula. First note that,

$$e^{i(\alpha+\beta)} = \cos(\alpha + \beta) + i \sin(\alpha + \beta).$$

But also that,

$$\begin{aligned} e^{i(\alpha+\beta)} = e^{i\alpha} e^{i\beta} &= (\cos\alpha + i \sin\alpha)(\cos\beta + i \sin\beta) \\ &= (\cos\alpha \cos\beta - \sin\alpha \sin\beta) + i(\sin\alpha \cos\beta + \cos\alpha \sin\beta). \end{aligned}$$

Equating real and imaginary parts gives the trigonometric identities for sums of angles,

$$\cos(\alpha + \beta) = \cos\alpha \cos\beta - \sin\alpha \sin\beta,$$

and,

$$\sin(\alpha + \beta) = \sin\alpha \cos\beta + \cos\alpha \sin\beta.$$

[24]The equation is related to the earlier de Moivre's formula.

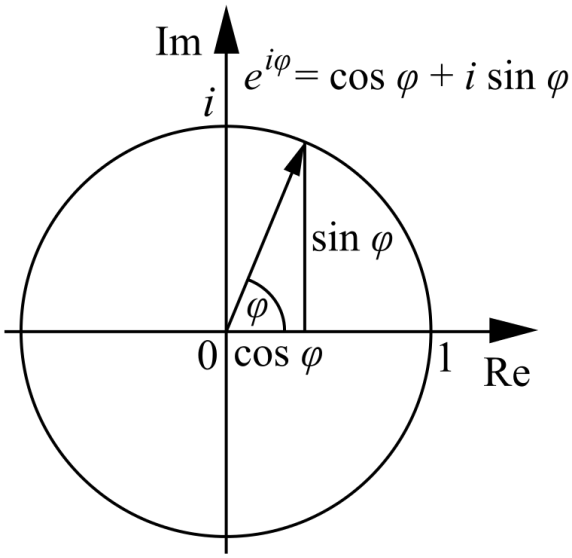


FIGURE 1. e^{ix} traces a unit circle on the complex plane (https://commons.wikimedia.org/wiki/File:Euler's_formula.svg).

9. Integral Transforms

Integral transforms are a technique for transforming a function in one domain to an equivalent function is another domain. It just so happens that some problems are more easily solved in one domain than the other.

9.1. **The Laplace Transform.** The Laplace transform is an integral transform defined as,

$$F(s) = \mathcal{L}\{f(t)\} = \int_0^\infty f(t)e^{-st}\,dt\,.$$

It is a more generalised transform than the Fourier transform, and maps a function, $f(t)$, from the time (t) domain to the s domain. The inverse Laplace transform is a more complicated function and usually it is more practical to take inverse transforms from a reference table. For example, let $f(x) = \cos kt$, we first use a trick to convert it to a more amenable form,

$$\begin{aligned}\cos(kt) &= \frac{1}{2}\cos(kt) + \frac{1}{2}i\sin(kt) + \frac{1}{2}\cos(kt) - \frac{1}{2}i\sin(kt)\\ &= \frac{1}{2}(\cos(kt) + i\sin(kt)) + \frac{1}{2}(\cos(-kt) + i\sin(-kt))\\ &= \frac{1}{2}e^{kit} + \frac{1}{2}e^{-kit}.\end{aligned}$$

Then,

$$\begin{aligned}\mathcal{L}\{\cos(kt)\} &= \int_0^\infty \frac{1}{2}e^{kit}e^{-st}\,dx + \int_0^\infty \frac{1}{2}e^{-kit}e^{-st}\,dt \\ &= \int_0^\infty \frac{1}{2}e^{(-s+ki)t}\,dx + \int_0^\infty \frac{1}{2}e^{(-s-ki)t}\,dt \\ &= \frac{1/2}{s-ki} + \frac{1/2}{s+ki} = \frac{s}{s^2+k^2}.\end{aligned}$$

If we are to apply the Laplace transform to differential equations, it is useful to establish a result for the transform of a derivative. By parts,

$$\begin{aligned}\mathcal{L}\{f(t)\} = \int_0^\infty f(t)e^{-st}dt &= \left[\frac{f(t)e^{-st}}{-s}\right]_0^\infty - \int_0^\infty f'(t)\frac{e^{-st}}{-s}dt \\ &= \frac{f(0)}{s} + \frac{1}{s}\mathcal{L}\{f'(t)\},\end{aligned}$$

thus in general, $\mathcal{L}\{f^{(n)}(t)\} = s^n\mathcal{L}\{f(t)\} - s^{n-1}f(0) - \cdots - f^{(n-1)}(0)$. Now, the linear homogeneous ordinary differential equation,

$$y'' - 5y' + 6y = 0,$$

with initial conditions $y(0) = 2$, and $y'(0) = 2$, can be solved with the Laplace transform. Using the derivatives formula on the left-hand side of the equation,

$$\mathcal{L}\{y\} - 5\mathcal{L}\{y'\} + 6\mathcal{L}\{y''\} = s^2\mathcal{L}\{y\} - 2s - 2 - 5s\mathcal{L}\{y\} + 10 + 6\mathcal{L}\{y\},$$

from which we have,

$$\mathcal{L}\{y\} = \frac{2s-8}{s^2-5s+6} = \frac{4}{s-2} - \frac{2}{s-3},$$

hence

$$y = \mathcal{L}^{-1}\left\{\frac{4}{s-2}\right\} - \mathcal{L}^{1}\left\{\frac{-2}{s-3}\right\} = 4e^{2t} - 2e^{3t}.$$

9.2. **Fourier Series.** A Fourier series, $g(t)$, is a way of representing any periodic function, $f(t)$, as the sum of a (possibly infinite) set of sines and cosines,

$$g(t) = \sum_{n=0}^{N} a_n \cos\left(\frac{2\pi nt}{T}\right) + \sum_{m=1}^{M} b_m \sin\left(\frac{2\pi mt}{T}\right),$$

where a_0 is an offset term, and T is the fundamental frequency, that is, the lowest value for which the function is periodic, that is, the lowest value for which $f(t+T) = f(t)$ for all t. Deriving a Fourier series representation involves solving for the a_n and a_m parameters. It is possible to isolate each parameter by multiplying by the corresponding wave and

integrating over the fundamental frequency. This eliminates all other terms in the series leaving,

$$a_n = \frac{2}{T}\int_0^T g(t)\cos\left(\frac{2\pi nt}{T}\right)dt\,.$$

A Fourier series can also be thought of as a technique for decomposing the harmonics of a wave. For example, a square wave function can be represented as an infinite sum of cosines, each higher frequency part converging asymptotically to the square wave form. A Fourier series is often written more succinctly in terms of complex exponential functions,

$$g(t) = \sum_{n=-N}^{N} c_n e^{i\frac{2\pi}{T}nx}$$

9.3. **Fourier Transform.** Fourier transforms are widely used in statistics, engineering, and physics. The Fourier transform maps a function in a time domain to a function in a frequency domain. The Fourier transform of a function $f(t)$ is defined as,

$$\hat{f}(s) = \int_{-\infty}^{\infty} f(t)e^{-2\pi its}\,dt,$$

The transformed function $\hat{f}(s)$ gives the amplitude for any given frequency, and can be interpreted as the level of agreement between a wave of that frequency and the target signal. Intuitively, what cross-correlation is to phase, the Fourier transform is to frequency. For example, for $f(x) = \sin(x)$, the transform becomes a Dirac delta function, that is, a single point on the real plane, corresponding to the single frequency of the function. A more complex sine wave, comprising multiple frequencies, will transform to a sum of delta functions, each representing a frequency. For the square wave function, composed of an infinite summation of frequencies, the transform takes the form $\hat{f}(\xi) = \mathrm{sinc}(x) = \sin(x)/x$, a contiunous sine wave with ever decreasing amplitude.

The Fourier inversion theorem defines an inverse Fourier transform that recovers the original function,

$$f(t) = \int_{-\infty}^{\infty} \hat{f}(s)e^{2\pi ist}\,ds\,.$$

Note the resemblance to the exponential form of the Fourier series–it is its real-valued, infinite integral counterpart.

9.4. **Discrete-time Fourier Transform.** The discrete-time Fourier transform (DTFT) transforms an infinite series of samples into a continuous freuquency function,

$$X(s) = \sum_{n=-\infty}^{\infty} x_n e^{-sn}.$$

Its inverse is a Fourier series, where the coefficients are simply the frequency amplitudes. The Shannon-Nyquist theorem relates continuous and discrete signals, proving that a sample rate of at least twice the highest frequency (the bandwidth) of the continuous wave is sufficient for an exact reproduction of the continuous signal (intuitively, no sinusoid of frequency below this will be able to interpolate the sample points).

9.5. **Discrete Fourier Transform.** The discrete Fourier transform (DFT, distinct from the DTFT) converts a finite sequence of N samples into a corresponding series of samples in the frequency or DTFT domain,

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{i2\pi}{N}kn}.$$

Note that a multi-dimensional DFT analogue exists. For images, this is two-dimensional. The convolution theorem says that the convolution of two functions is the point-wise product of their Fourier transforms. This holds for the DFT case, as well as the multi-dimensional DFT. An image convolution can therefore be performed as a pixel-wise product in the Fourier representations of the images (which should then be mapped back to pixel space). Other interesting operations are possible. For example, removing frequency information of a desired range involves simply masking regions of the frequency image (just as it would be with a 1D signal).

9.6. **Fast Fourier Transform.** One of the most important algorithms of the 20th century is the fast fourier transform (FFT) for computing the DFT. It is an efficient algorithm $\mathcal{O}(\log N)$ for the N samples, that improves upon the naive $\mathcal{O}(N^2)$ approach. There are many variants, and, like backpropagation, was discovered earlier, but was rediscovered and popularised by others.

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} e^{-\frac{2\pi i}{N/2}mk} + e^{-\frac{2\pi i}{N}k} \sum_{n=0}^{N/2-1} x_{2n+1} e^{-\frac{2\pi i}{N/2}mk}$$

The Cooley-Tukey version exploits a number of symmetries in the DFT by separating even and odd power terms into two DFTs of equal length (N/2), which are solved for each sample. It may be shown that,

$$\begin{aligned} X_k &= E_k + e^{-2\pi i/Nk} O_k \\ X_{k+N/2} &= E_k - e^{-2\pi i/Nk} O_k \end{aligned} \tag{8}$$

where E_k and O_k are the even and odd DFTs. This can be used to establish a recurrence relation, wherein DFTs of progressively halving length are computed. The FFT efficiency gain arises from the sharing of computation in the above equations. First note that progressive halving induces a computational tree structure of depth $\log(N)$. However, each node (representing a partial-length DFT) feeds two nodes above it. Hence, rather than a

naive binary tree, the branching factor of the is 1, implying $\mathcal{O}(N)$ operations only at each level.

10. The Factorial Function

The factorial function, written $n!$, for a positive integer n is the product of all natural numbers less than or equal to n. That is,

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1.$$

The value of 0! is 1, not as a matter of mathematical necessity, but because it continues the pattern: $0! = (n-n)! = n!/n! = 1$. Otherwise, we can note that the factorial function models the number of permutations of n objects, that is, ${}^nP_n = n!$. The number of ways of permuting *no* objects is assuredly 1, as no reorderings of *no* objects are possible. There is also such a thing as the *double* factorial function,

$$n!! = n \times (n-2) \times (n-4) \times \cdots,$$

which multiples all even numbers $\leq n$ if n is even, and all odd numbers $\leq n$ if n is odd. The factorial function is undefined for non-natural numbers, but the gamma function extends it to all real and complex numbers. The factorial function grows faster than exponentiation, but not as fast as double exponentiation or tetration, the hyperoperation of repeated exponentiation. That is,

$$\mathcal{O}(a^x) < \mathcal{O}(x!) < \mathcal{O}(a^{b^x}) < \mathcal{O}({}^nx = x^{x^{\cdot^{\cdot^{\cdot^x}}}})$$

10.1. **Stirling's Approximation.** Stirling's approximation is a powerful approximation for the factorial function that converges as $n \to \infty$. It was first discovered by de Moivre, but Scottish mathematician James Stirling (1692-1770) found a nice simplification. Its derivation begins by taking the log of the factorial function,

$$\ln(n!) = \ln(1) + \ln(2) + \ln(3) + \cdots + \ln(n).$$

Now we notice that subtracting $\frac{1}{2}(\ln(1) + \ln(n))$ gives the trapezoidal rule[25] approximation to the integral of $\ln(n)$ with $\Delta x = 1$. That is,

$$\ln(n!) - \frac{1}{2}\cancelto{0}{\ln(1)} - \frac{1}{2}\ln(n) = \frac{\cancelto{0}{\ln(1)} + \ln(2)}{2} + \frac{\ln(2)+\ln(3)}{2} + \frac{\ln(3)+\ln(4)}{2} \cdots + \frac{\ln(n+1)+\ln(n)}{2}$$
$$\approx \int_1^n \ln x \, dx = n\ln n - n + 1 + \sum_{k=2}^{m} \frac{(-1)^k B_k}{k(k-1)} \left(\frac{1}{n^{k-1}} - 1 \right) + R_{m,n}.$$

[25]The trapezoidal rule is a part of a family of techniques for approximating integrals known as Riemann sums.

The error term is expressed in terms of Bernoulli numbers, B_k and the remainder term, $R_{m,n}$, from the Euler-Maclaurin formula. The index, m, may be chosen freely, with $R_{m,n} \to 0$ as $m \to \infty$. We next consider the behaviour of the error term as n tends to ∞,

$$\lim_{n\to\infty}\Big(\ln(n!) - \frac{1}{2}\ln n - n\ln n + n\Big) = 1 - \sum_{k=2}^{m}\frac{(-1)^k B_k}{k(k-1)} + \lim_{n\to\infty} R_{m,n},$$

and it is known that $\lim_{n\to\infty} R_{m,n} = R_{m,n} + O\Big(\frac{1}{n^m}\Big)$. Thus, the difference tends toward a constant, denoted c. Returning to our equation, we have,

$$\ln(n!) = n\ln n - n + \frac{1}{2}\ln n + 1 + c + \sum_{k=2}^{m}\frac{(-1)^k B_k}{k(k-1)}\left(\frac{1}{n^{k-1}}\right) + \mathcal{O}\left(\frac{1}{n^m}\right).$$

To eliminate the bothersome sum, we can choose the parameter, $m = 1$, incurring an error term of greater magnitude, $O(1/n)$. Exponentiating both sides gives,

$$n! = e^c \cdot \left(\frac{n}{e}\right)^n \cdot \sqrt{n} \cdot \left(1 + \mathcal{O}\left(\frac{1}{n}\right)\right),$$

as $n \to \infty$. Note that the error term comes from the fact that $e^x \approx 1 + x$, for x close to 0. To deal with the constant, e^c, which we will denote, C, recall from Wallis' product that,

$$\frac{1}{2n+1} \cdot \frac{2^{4n}(n!)^4}{[(2n)!]^2} \to \frac{\pi}{2},$$

as $n \to \infty$. Substituting our convergent expression for $n!$,

$$\frac{1}{2n+1} \cdot \frac{2^{4n}[C(n/e)^n\sqrt{n}]^4}{[C(2n/e)^{2n}\sqrt{2n}]^2} = \frac{n}{4n+2} \cdot C^2 \to \frac{\pi}{2},$$

from which we can see $C \to \sqrt{2n}$, as $n \to \infty$. Finding the convergent value through application of Wallis' product was Sterling's contribution. Our approximation therefore simplifies to Sterling's approximation,

$$n! \to \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

10.2. **The Gamma Function.** The gamma function is a continuous function that interpolates the factorial function (Figure 2), and extends its domain from the natural numbers to the real and complex numbers. It is defined as,

$$\Gamma(t) = \int_0^\infty x^{t-1}e^{-x}\,dx\,.$$

To show its connection to the factorial function, we integrate by parts,

$$\begin{aligned}\Gamma(n) &= \left\{ -x^{n-1}e^{-x} \right\}_{x=0}^{\infty} + (n-1)\int_0^{\infty} x^{n-2}e^{-x}\,dx \\ &= (n-1)\Gamma(n-1).\end{aligned}$$

Noting that $\Gamma(1) = 1$, it is clear that $\Gamma(n) = (n-1)!$ for all positive integers, n. However, the gamma function is further defined for all complex numbers, t. It is worth nothing that there exists a related function, the pi function, $\Pi(t) = \Gamma(t+1)$, that does away with the offset term, so as to coincide exactly with the factorial function. Another related function, the beta function, is a composite of the gamma function, where $B(x,y) = \Gamma(x)\Gamma(y)/\Gamma(x+y)$. The gamma function makes many appearances in the domain of statistics, for example in Student's t-distribution. There is also an intimate link to the Normal distribution, which we can show by first observing that the form of the function $\varphi(x) = e^{-x^2}$ is that of the bell curve. Now, the infinite integral[26] of this function is,

$$\int_{-\infty}^{\infty} e^{-t^2}\,dt = 2\cdot\int_0^{\infty} e^{-t^2}\,dt\,.$$

Defining $t = u^{1/2}$, we have it that $dt = \frac{1}{2}u^{-1/2}\,du$ and integrating by substitution gives,

$$\int_{-\infty}^{\infty} e^{-t^2}\,dt = 2\cdot\frac{1}{2}\cdot\int_0^{\infty} u^{1/2-1}e^{-u}\,du = \Gamma(1/2).$$

Given that we know the area under the unnormalised bell curve to be $\sqrt{\pi}$, we may write, $\Gamma(1/2) = \sqrt{\pi}$. Though there exist many such identities for points on the gamma function, to evaluate the gamma function at any positive real point, the Stirling approximation was historically used as a good approximation. Its use has been supplanted in recent decades by the numerical algorithm, the Lanczos approximation. An interesting property of the gamma function is that it is the only function that interpolates the factorial function and is also log-convex. This is according to a result known as the Bohr-Mollerup theorem.

11. Fundamentals of Statistics

11.1. **Philosophical Notions of Probability.** Pierre-Simon Laplace (1749-1827) described probability as, 'nothing but common sense reduced to calculation.'[27] As it is, there are two competing interpretations of the meaning of probabilities: the *frequentist* interpretation, and the *Bayesian* interpretation. In the frequentist interpretation, probabilities represent frequencies or rates. Thus, a probability of 0.5 that a coin comes up

[26] An integral taken over an infinite integral is known as *improper* and is an abuse of notation to do away with limits.

[27] That is not to say probability theory is intuitive (it is often not), but rather that probabilities model real world phenomena in an intuitive, simplified way, without addressing the potentially complex physical interactions behind them. A good example is the central limit theorem and the bell curve, which smooths over the uncountable complexities of reality.

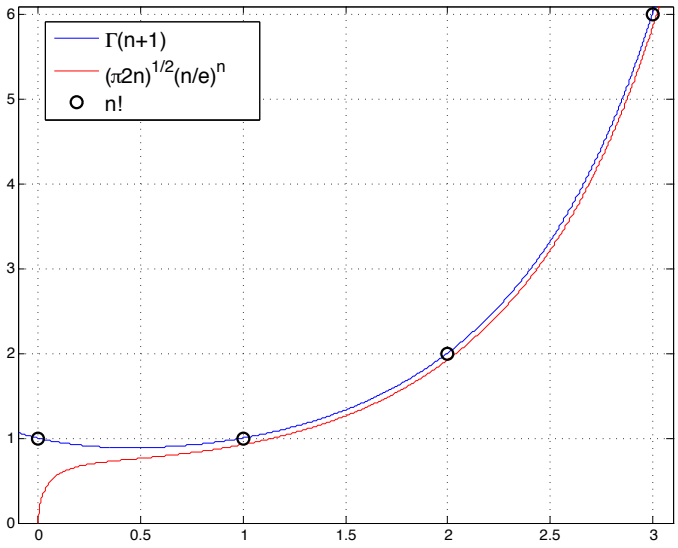


FIGURE 2. Comparison of the discrete factorial points, the gamma function, which interpolates them, and Stirling's approximation.

heads[28] means that, with sufficient throws, this proportion of heads will prevail. In the Bayesian (or epistemological) interpretation, probabilities quantify an *uncertainty* or *degree of belief* in an outcome, and so is more closely aligned with information theory. In information theory, a fair coin flip (50-50) has maximum entropy, because a uniform distribution maximises uncertainty. Though both views are fully compatible, adopting one or the other may change the emphasis.

11.2. **Fundamental Rules.**

11.2.1. *Probability Functions.* Probabilities express the uncertainty (or frequency) of a random variable, X, of assuming states (events) $A, B, C, \ldots$, as a numeric value between 0 (impossible) and 1 (certain). A distribution may be defined by a function to allocate the probabilities. We write $p(X = A)$ or simply $p(A)$ for the probability X assumes state A.

Discrete random variables take values in a finite or countably infinite set (such as the set of integers). The function that assigns each event a probability is called the probability mass function (pmf). The probabilities of events in a distribution must sum to 1. Thus,

$$\left(\sum_{k=1}^{K} p(X = k) \right) = 1.$$

[28] Although recent research has shown the probability is slightly biased towards the upward-facing side of the coin when flipped, with about a 51% chance of landing on that same side. This is linked somehow to the physics of the problem. See the Diaconis-Holmes-Montgomery coin-tossing theorem.

Continuous probability distributions express probabilities for random variables taking values in a continuum. For this reason, it no longer makes sense to model probabilities of the form $p(X = k)$. Such an event taken apart has an infinitesimal probability. Rather, we compute the probability of intervals, such as $p(X \leq k)$. The function that assigns probabilities is the cumulative density function (cdf),

$$P(X \leq a) = \int_{-\infty}^{a} p(x)\, dx\,.$$

Should we wish to compute the probability of a closed interval, we take,

$$P(a \leq X \leq b) = P(X \leq b) - P(X \leq a).$$

The cdf is the integral of the *probability density function* (pdf), written $p(x)$, a function that expresses *densities*. A density is a quantity related to probability by the expression, $P(x \leq X \leq x + dx) \approx p(x)dx$, so that the value of $p(x)$ can be though of as the relative weight of probabilities measured on intervals local to x. In actual fact, the pdf is the derivative of the cdf with,

$$p(x) = \lim_{\Delta x \to 0} \frac{P(x + \Delta x) - P(x)}{\Delta x},$$

and for this reason, the pdf must be non-negative, and must integrate to 1.

11.2.2. *Union of Events.* The probability of a union of events A and B, that is, the probability that A *or* B occurs is,

$$p(A \vee B) = p(A) + p(B) - p(A \wedge B).$$

When A and B are *mutually exclusive*, the joint probability, $p(A \wedge B) = 0$. Mutually exclusive events are events that cannot happen simultaneously, such as A, the event of rolling a 2 on a die, and B, the event of rolling a 3 on a die. An example of events that are *not* mutually exclusive is A, the event we draw a king from a deck of cards, and B, the event we draw a diamond. The events can happen either separately or together. Measuring the probability of the union of events seems rarely to be of interest in machine learning, however.

11.2.3. *Inclusion-Exclusion Principle.* The generalisation to the union of events formula is arrived at using the inclusion-exclusion principle, used for counting unions of sets. The case corresponding to the probability of the union of two events stated above is,

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

for sets A and B. If we include a third set, C, we acquire,

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|,$$

These ideas are most easily understand by looking at Venn diagrams (Figure 3), where the probability spaces of the events are depicted as sets. In general, the probability of the union of n events is,

$$p(X_1 \vee \cdots \vee X_N) = \sum_{i=1}^{N} p(X_i) - \sum_{1 \le i < j \le N} p(X_i \wedge X_j) + \sum_{1 \le i < j < k \le N} p(X_i \wedge X_i \wedge X_j) - \cdots + (-1)^{N-1} p(X_1 \wedge \cdots \wedge X_N)$$

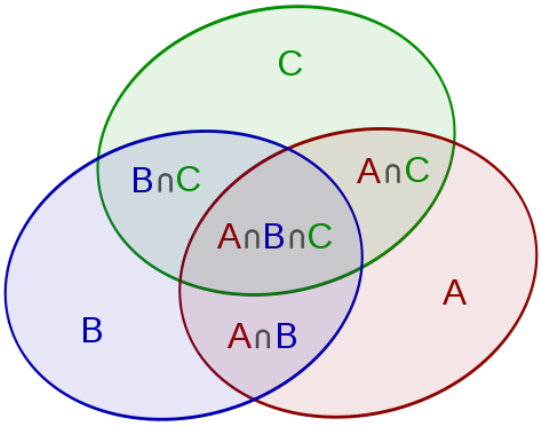


FIGURE 3. The inclusion-exclusion principle is best understood with Venn diagrams (https://commons.wikimedia.org/wiki/File:Inclusion-exclusion.svg).

11.2.4. *Intersection of Events.* The joint distribution, $p(A \wedge B)$, or simply, $p(A, B)$, is the probability that events A and B occur together. The joint distribution is factorised in accordance with the dependencies of the events. If two events, A and B, are independent, we write $A \perp\!\!\!\perp B$. Independence is distinct from mutual exclusivity, which is exclusive or. Independent events may occur apart or together, but the essence of independence is that the occurrence of one event does not affect the occurrence of the other. Hence, the joint distribution of independent A and B is,

$$p(A, B) = p(A)p(B),$$

that is, the product of the probabilities of the events taken separately. When A and B are *dependent*, the occurrence of one changes the probability of the other occurring. In this case we write,

$$p(A, B) = p(A|B)p(B) \tag{9}$$

$$= p(B|A)p(A). \tag{10}$$

The probability, $p(A|B)$, is the conditional probability of the event A *given* the occurrence of event B. Notice the symmetry of the conditioning–we can condition on any variable we like. The definition of independence is that $p(A|B) = p(A)$.

11.3. **Bayes' Theorem.** We may derive the all-important Bayes' theorem by equating equations (9) and (10) and rearranging,

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}.$$

Bayesians interpret this as an expression relating event A and evidence B. The probability $p(A)$ is known as the 'prior' distribution for the event taken alone (that is, 'before' the evidence is presented). The 'posterior' distribution, $p(A|B)$, represents the change in the uncertainty of the event after the evidence of the likelihood, $p(B|A)$, has been presented.

11.3.1. *Chain Rule.* Extending the intersection rule gives us a general result for N random variables, $X_!, X_2, \ldots, X_N$, known as the chain rule. The joint distribution may be factorised,

$$\begin{aligned} p(X_1, X_2, X_3, \ldots, X_N) &= p(X_1)p(X_2, X_3, \ldots, X_N|X_1) \\ &= p(X_1)p(X_2|X_1)P(X_3, \ldots, X_N|X_1, X_2) \\ &= p(X_1)p(X_2|X_1)P(X_3|X_1, X_2)\cdots P(X_N|X_1, X_2, \ldots, X_{N-1}) \end{aligned}$$

11.3.2. *Law of Total Probability.* The law of total probability allows us to calculate the probability of an event by summing over a joint distribution,

$$\begin{aligned} p(X = x) &= \sum_{y\in Y} p(X = x, Y = y) \\ &= \sum_{y\in Y} p(X = x|Y = y)p(Y = y). \end{aligned}$$

This is called the marginal distribution on X, and we say that Y has been 'marginalised out'.

11.3.3. *Markov Chain.* A Markov chain is a sequence of random variables, $X_1, X_2, \ldots, X_N$ written,

$$X_1 - X_2 - \cdots - X_N.$$

A Markov chain exhibits the Markov property, meaning each variable is dependent only on the previous variable in the sequence. Otherwise put, $p(X_{N+1}|X_N, X_{N-1}) = p(X_{N+1}|X_N)$. The chain rule applied to a Markov chain therefore simplifies to,

$$\begin{aligned} p(X_1, X_2, X_3, \ldots, X_N) &= p(X_1)p(X_2|X_1)P(X_3|X_1, X_2)\cdots P(X_N|X_1, X_2, \ldots, X_{N-1}) \\ &= p(X_1)p(X_2|X_1)P(X_3|X_2)\cdots P(X_N|X_{N-1}) \end{aligned}$$

To give an example of what a Markov chain might model, let X be the event that we see clouds on the horizon this morning; Y it is raining by noon; Z it is raining this evening. These events form a Markov chain, $X - Y - Z$. The probability of rain at noon, Y, is

clearly dependent on there being clouds in the morning, X. That it rains in the evening, Z, also depends on there being clouds in the morning, X, but not once we have knowledge of Y. Thus, Z is independent of X given (conditioned on) the more recent and informative event Y.

11.4. **Quantiles.** Quantiles mark milestones in the cumulative distribution, that is, the values for which certain cumulative probabilities are reached. Quantiles may be taken anywhere, but the most common ones are:

- median - the value for which there is a 0.5 probability of exceeding and 0.5 probability of falling short, that is, the midpoint of the distribution;
- quartile (Q_1, Q_2, Q_3) - the values marking 0.25, 0.5, and 0.75, probabilities in the cumulative distribution (Q_2 is the median). Note that in certain fields that apply statistics, such as finance, a quartile is rather one of the four intervals separated by the quartile points;
- percentile - the values marking the $0.01, 0.02, \ldots, 0.99$ cumulative probabilities.

For a continuous distribution, any quartile, q, can be computed by solving the integral, for the desired probability,

$$p = \int_{-\infty}^{q} p(x)\, dx\,.$$

11.5. **Moments.** Moments are quantitative measures (statistics) of a distribution first formulated by English statistician Karl Pearson (1857-1936). The general form is,

$$\mathbb{E}[(X - c)^N],$$

for some constants, c and N. When $c = 0$, we have what is called the *raw* moment. Usually $c = \mathbb{E}[X]$, as we are chiefly interested in how X moves about its mean, and these are called *central* moments. Further, standardised moments are normally used in higher orders, presumably to moderate the exaggeration of higher powers.

11.5.1. *Expected Value.* The expected value (or mean, or average), denoted μ, of a discrete random variable is the average of the random variable, weighted by its probabilities. It is the first-order ($N = 1$) *raw* moment (the central moment is 0 by definition). It is defined as,

$$\mathbb{E}[X] \triangleq \sum_x p(x) \cdot x.$$

Similarly, for continuous, X,

$$\mathbb{E}[X] \triangleq \int_{-\infty}^{\infty} p(x) x\, dx\,.$$

In general, the expectation of a transformation, $f(X)$, is $\mathbb{E}[f(X)] = \sum_x p(x) f(x)$

11.5.2. *Variance.* Variance, $\mathrm{Var}[X]$, or σ^2, is the expected (average) squared deviation from the mean, and therefore a measure of spread. Clearly, we need a second order measure to indicate spread since the first moment will be 0 for a symmetric distribution (the negative scores cancelling out the positive scores). Thus, variance,

$$\begin{aligned}\mathrm{Var}[X] &= \mathbb{E}[(X - \mathbb{E}[X])^2] \\ &= \mathbb{E}[X^2 - 2\mathbb{E}[X]X + \mathbb{E}[X]^2] \\ &= \mathbb{E}[X^2] - 2\mathbb{E}[X]^2 + \mathbb{E}[X]^2 \\ &= \mathbb{E}[X^2] - \mathbb{E}[X]^2.\end{aligned}$$

Another measure of spread is the standard deviation, σ, defined as the square root of the variance, $\sqrt{\sigma^2}$.

11.5.3. *Skewness.* Skewenss, γ_1, is the third order *standardised* moment. It is a measure of the skew or *asymmetry* of a distribution, and is simply the standardised third order central moment,

$$\gamma_1 = \mathbb{E}\left[\frac{(X - \mathbb{E}[X])^3}{\sigma^3}\right]$$

11.5.4. *Kurtosis.* Kurtosis, $\mathrm{Kurt}[X]$, is the fourth-order standardised central moment. It is a measure of the heaviness of the tails of the distribution. It is defined to be,

$$\mathrm{Kurt}[X] = \frac{\mathbb{E}[(X - \mathbb{E}[X])^4]}{\mathbb{E}[(X - \mathbb{E}[X])^2]^2}$$

11.6. **Sample Statistics.** Given a set of sample data $x_1, x_2, \ldots, x_N$, we can compute estimates of the distribution's statistics. Samples are drawn at random *with* replacement. The sample mean is simply the arithmetic average,

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i.$$

At first it may appear that through the $\frac{1}{N}$ term that we are assuming a uniform distribution. This is not the case, however. Wherever non-uniformity exists, the data will promote the right proportions as a matter of course, by supplying more likely scores in greater number. That is, the higher probabilities will feature more heavily, and receive more weight, effecting higher probabilities. The sample variance is a lot more curious. We might take the sample variance in a similar way to the mean,

$$\sigma_x^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

This would be, surprisingly, not quite correct. This is due to the fact we are using the sample mean only, not the true mean. Given all possible samples, the expectation of the sample variance is,

$$\begin{aligned}
\mathbb{E}[\sigma_x^2] &= \mathbb{E}\left[\frac{1}{N}\sum_{i=1}^{N}\left(x_i - \frac{1}{N}\sum_{j=1}^{N} x_j\right)^2\right] \\
&= \frac{1}{N}\sum_{i=1}^{N}\mathbb{E}\left[x_i^2 - \frac{2}{N}x_i\sum_{j=1}^{N}x_j + \frac{1}{N^2}\sum_{j=1}^{N}x_j\sum_{k=1}^{N}x_k\right] \\
&= \frac{1}{N}\sum_{i=1}^{N}\frac{N-2}{N}\mathbb{E}[x_i^2] - \frac{2}{N}\sum_{j\neq i}^{N}\mathbb{E}[x_i x_j] + \frac{1}{N^2}\sum_{j=1}^{N}\sum_{k\neq j}^{N}\mathbb{E}[x_j x_k] + \frac{1}{N^2}\sum_{j=1}^{N}\mathbb{E}[x_j^2] \\
&= \frac{1}{N}\sum_{i=1}^{N}\frac{N-2}{N}(\sigma^2+\mu^2) - \frac{2(N-1)}{N}\mu^2 + \frac{N(N-1)}{N^2}\mu^2 + \frac{1}{N}(\sigma^2+\mu^2) \\
&= \frac{N-1}{N}\sigma^2,
\end{aligned}$$

where σ^2 and μ are the true variance and mean. Note the substitution in the second to last step comes from the definition of variance, $\mathrm{Var}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2$. Thus, our formula becomes,

$$\sigma_x^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2.$$

This is called the unbiased sample variance, and the leading quotient is known as Bessel's[29] correction. This formula is usually presented without the above justification, and may be baffling to the student. The same correction is used for the sample covariance and other statistics. Given independent samples, these statistics converge to the true statistics as the sample size increases, as per the law of large numbers. That is, the sample variance converges to the expected sample variance, and the Bessel's correction converges to 1.

11.7. **Covariance.** Covariance measures the interaction between random variables, that is, it quantifies how much two random variables move together. It is defined as,

$$\begin{aligned}
\mathrm{Cov}[X,Y] &= \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] \\
&= \mathbb{E}[XY - \mathbb{E}[Y]X - \mathbb{E}[X]Y + \mathbb{E}[X]\mathbb{E}[Y]] \\
&= \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y].
\end{aligned}$$

In the multivariate case, we have the covariance matrix as the outer product,

[29]Friedrich Bessel (1784-1846) was a German mathematician and scientist.

$$\begin{aligned}\mathrm{Cov}[\mathbf{x}] &\triangleq \mathbb{E}[(\mathbf{x}-\mathbb{E}[\mathbf{x}])(\mathbf{x}-\mathbb{E}[\mathbf{x}])^T] \\ &= \begin{bmatrix} \mathrm{Var}[X_1] & \mathrm{Cov}[X_1,X_2] & \dots & \mathrm{Cov}[X_1,X_n] \\ \mathrm{Cov}[X_2,X_1] & \mathrm{Var}[X_2] & \dots & \mathrm{Cov}[X_2,X_n] \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{Cov}[X_n,X_1] & \mathrm{Cov}[X_n,X_2] & \dots & \mathrm{Var}[X_n] \end{bmatrix}\end{aligned}$$

11.8. **Correlation.** Correlation is covariance normalised by the variance of each of the variables taken apart. It measures the extent to which two variables are *linearly* related and computes a correlation coefficient in the range $[-1,1]$. A correlation coefficient of 0 indicates no relation. The formula for correlation is,

$$\mathrm{corr}(X,Y) = \frac{\mathrm{Cov}[X,Y]}{\sqrt{\mathrm{Var}[X]\mathrm{Var}[Y]}}.$$

Similary, for the multivariate case, the correlation matrix is,

$$\mathrm{Corr}[\mathbf{x}] = \begin{bmatrix} \mathrm{Corr}[X_1,X_1] & \mathrm{Corr}[X_1,X_2] & \dots & \mathrm{Corr}[X_1,X_n] \\ \mathrm{Corr}[X_2,X_1] & \mathrm{Corr}[X_2,X_2] & \dots & \mathrm{Corr}[X_2,X_n] \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{Corr}[X_n,X_1] & \mathrm{Corr}[X_n,X_2] & \dots & \mathrm{Corr}[X_n,X_n] \end{bmatrix}.$$

Each of the diagonal elements of the correlation matrix will be equal to 1.

11.9. **Transformation of Random Variables.** For random variable, X, define a new random variable, $Y = aX + b$. How have its properties changed? Recall for discrete variables that $\mathbb{E}[f(X)] = \sum_x p(x)f(x)$. Then we have,

$$\begin{aligned}\mathbb{E}[X] &= \sum_x p(x)(ax+b) \\ &= a\sum_x p(x)x + b\sum_x 1 = a\mathbb{E}[X] + b.\end{aligned}$$

The same result is arrived at for continuous random variables. This effect is known as the 'linearity of expectation'. Variance behaves a little differently,

$$\begin{aligned}\mathrm{Var}[X] &= \mathbb{E}[(X-\mathbb{E}[X])^2] \\ &= \mathbb{E}[(aX+b-(a\mathbb{E}[X]+b))^2] \\ &= \mathbb{E}[a^2(X-\mathbb{E}[X])^2] \\ &= a^2\mathrm{Var}[X]\end{aligned}$$

Thus, adding a constant to a random variable does not change its variance, but scaling a random variable scales its variance by the square of that factor.

11.10. **Change of Variables.** In the general case, given transformation, $Y = f(X)$, a discrete distribution can be derived by,

$$p_y(y) = \sum_{x:f(x)=y} p_x(x),$$

that is, by mapping the probabilities of the values of x corresponding to y via the mapping. Things are not so simple for continuous random variables, in which case we must address the cdf. Being a cumulative function, it is monotonic[30]. If it is monotonic, it is bijective[31], and if it is bijective, it is invertible. Therefore, we can write the cumulative distribution,

$$P_y(y) = P(f(X) \leq y) = P(X \leq f^{-1}(y)) = P_x(f^{-1}(y))$$

Now,

$$p_y(y) \triangleq \frac{d}{dy} P_y(y) = \frac{d}{dy} P_x(f^{-1}(y)) = \frac{dx}{dy}\frac{d}{dx} P_x(x),$$

giving us the *change of variables* formula,

$$p_y(y) = p_x(x)\Big|\frac{dx}{dy}\Big|.$$

An example here would perhaps be useful, so let $X \sim U(-1, 1)$ and $Y = X^2$. Since X is uniformly distributed on an interval of length 2, its pdf, $p_x(x) = 1/2$. Therefore, by the change of variables formula, $p_y(y) = \frac{1}{2} \cdot \frac{1}{2} y^{-1/2} = \frac{1}{4} y^{-1/2}$.

In the multivariate case, we have need of the Jacobian matrix from vector calculus, and the change of variables formula becomes,

$$p_y(\mathbf{y}) = p_x(\mathbf{x})\Big|\det\left(\frac{\partial \mathbf{x}}{\partial \mathbf{y}}\right)\Big| = p_x(\mathbf{x})|\det \mathbf{J}_{\mathbf{y}\to\mathbf{x}}|.$$

11.11. **Monte Carlo Statistics.** Monte Carlo statistics are a branch of numerical techniques for providing estimations using random sampling. The name comes from the famous casino in the city in the principality of Monaco near the south of France. One famous example of applying Monte Carlo techniques is that of sampling from a unit circle inscribed inside a 2×2 unit square. The ratio of the areas of circle to square is $\pi : 4$. We can therefore estimate the value of π by randomly sampling points within the square, and comparing the number of points that fall inside and outside the circle (Figure 4). A sufficiently large sample would converge to the correct result by the law of large numbers.

[30] A monotonic function holds the property that $f(b) > f(a)$ for any $b > a$.

[31] If f is bijective, it is a one-to-one mapping.

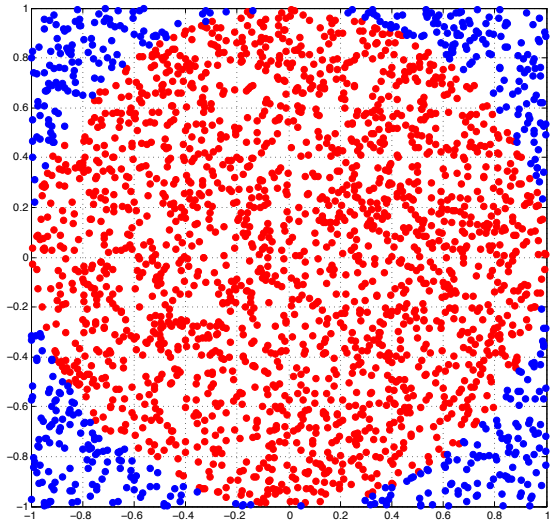


FIGURE 4. Monte Carlo approach to estimating π. 1963 of the 2500 random points landed in the circle, giving an estimation of $\pi \approx 3.1408$. Generated with `monteCarloDemo.m`.

11.11.1. *Buffon's Needles.* An extremely nice and demonstrative Monte Carlo problem is know as the Buffon's needle problem. It also derives a technique for numerically computing the digits of π using probabilities, but without invoking any circles explicitly. The problem consists of a set of parallel lines, separated by a distance of $2L$, where L is the length of each of the needles. The needles are then scattered randomly between the parallel lines. The proportion of needles intersecting a line gives us an approximation of π! There are two random variables at play here: the distance, x, of a needle from the nearest line; and the orientation, θ, of the needle with respect to the lines. The assumption we make is that these are uniformly randomly distributed, hence x, which varies between 0 and L, has probability function $p_x = 1/L$, and θ, which varies between 0, and $2/\pi$. To determine the probability of a needle crossing a line, we note that it is sufficient that $x < \frac{L}{2}\sin\theta$. This is most easily seen with a diagram. These conditions alone are enough to calculate the probability,

$$p = \int_0^{\pi/2} \int_0^{\frac{L}{2}\sin\theta} p_x p_\theta \, dx \, d\theta = \frac{2}{L\pi} \int_0^{\pi/2} \frac{L}{2} \sin\theta \, d\theta = \frac{1}{\pi}.$$

Therefore, the proportion of a sample of n scattered needles crossing a line gives, $\frac{\#\text{ crosses}}{n} \approx \frac{1}{\pi} \implies \pi \approx \frac{n}{\#\text{ crosses}}$

12. Common Discrete Probability Distributions

12.1. **Uniform Distribution.** A uniform distribution models a coin toss, or a die roll, or any other probability of K events with equal chance, defining an interval, $[a, b]$, on some

ordered encoding of the events. The probability mass function for a uniform distribution is,

$$p(k) = \frac{1}{K}.$$

The mean of this distribution is, $\mu = \frac{a+b}{2}$. Note there is also a continuous version of the uniform distribution. Laplace's *principle of insufficient reason* argues to assume a uniform distribution for a discrete variable in the absence of further information (Gaussian for continuous variables).

12.2. **Bernoulli Distribution.** A Bernoulli distribution models an event with two alternative outcomes. It may therefore model a coin toss, but is parameterised with probability θ of outcome 1, and $1-\theta$ of outcome 2, so any binary event may be modelled. We write $X \sim \text{Ber}(\theta)$. Thus,

$$\text{Ber}(x|\theta) = \theta^{1_{x=1}}(1-\theta)^{1_{x=0}}.$$

The expected value is therefore,

$$\mathbb{E}[X] = \theta \cdot 1 + (1-\theta) \cdot 0 = \theta$$

12.3. **Binomial Distribution.** A Binomial distribution may model the number of heads, k, from the flipping of a coin (or any other binary event) n times. The Binomial distribution generalises the Bernoulli distribution to n binary trials (rather than a single trial). Its pmf is,

$$\text{Bin}(k|n,\theta) \triangleq \binom{n}{k}\theta^k(1-\theta)^{n-k}$$

Note the resemblance in form to the Bernoulli distribution. The multinomial distribution further generalises the binomial distribution to k-ary events n times.

13. Common Continuous Probability Distributions

13.1. **Laplace Distribution.** The Laplace pdf resembles two opposing exponential functions that meet at the origin, and is written,

$$f(x;\mu,b) = \frac{1}{2b}\exp\left\{-\frac{|x-\mu|}{b}\right\},$$

where b is a normalising constant. This distribution is similar in form to the Gaussian distribution, but the exponent is linear, and so the log probability decreases linearly, whereas a Gaussian's 'tails' decrease quadratically. Thus, we say the Laplace distribution has 'heavy' tails. This has implications when we fit a distribution to data (such as in machine learning), where the Laplace distribution is less sensitive to outliers, as extreme events have higher probability in a Laplace distribution than in a Gaussian. One advantage

is it is easily integrable, whereas the Gaussian has no closed-form solution, as we shall see. Integrating the pdf gives,

$$\begin{aligned} F(x) &= \int_{-\infty}^{+\infty} \frac{1}{2b} \exp\left\{ - \frac{|x-\mu|}{b} \right\} dx \\ &= \int_{-\infty}^{\mu} \frac{1}{2b} \exp\left\{ \frac{x-\mu}{b} \right\} dx + \int_{\mu}^{+\infty} \frac{1}{2b} \exp\left\{ \frac{\mu - x}{b} \right\} dx \\ &= \frac{1}{2}\left[\exp\left\{ \frac{x-\mu}{b} \right\} \right]_{x=-\infty}^{x=\mu} + \frac{1}{2}\left[\exp\left\{ \frac{\mu - x}{b} \right\} \right]_{x=\mu}^{x=+\infty} \\ &= \frac{1}{2}(1 - -0) + \frac{1}{2}(0 - -1) = 1, \end{aligned}$$

which confirms its validity as a probability distribution.

13.2. **Gaussian Distribution.** The Gaussian or Normal distribution was first proposed by Karl Frederich Gauss (1777-1855) in 1809. In the same paper, he presented other fundamental tenets of statistics–the method of least squares (for fitting data with Gaussian error), and maximum likelihood estimation. The pdf of a Gaussian distribution is,

$$f(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left\{ \frac{-(x-\mu)^2}{2\sigma^2} \right\}.$$

Its shape is the well-known bell curve, the distribution so often seen in real-world data (the profound central limit theorem offers an explanation for why that is). The cdf for a Gaussian distribution,

$$\Phi(z; \mu, \sigma^2) = \int_{-\infty}^{x} \mathcal{N}(z|\mu, \sigma^2)\, dz\,.$$

Defining $u = \frac{z-\mu}{\sqrt{2}\sigma}$, and so $du = dz/\sqrt{2}\sigma$, we can integrate by substitution,

$$\begin{aligned} \Phi(z; \mu, \sigma^2) &= \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\frac{z-\mu}{\sqrt{2}\sigma}} e^{-u^2} \sqrt{2}\sigma\, du \\ &= \frac{1}{2}\Big[\operatorname{erf}(u) \Big]_{u=-\infty}^{u=(z-\mu)/\sqrt{2}\sigma} \\ &= \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{z-u}{\sqrt{2}\sigma} \right) \right), \end{aligned}$$

where $\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$ is the error function. This is a dead-end for closed-form solutions, and we must have recourse to an infinite series. Substituting the Taylor expansion for e^{-t^2},

$$\begin{aligned}\operatorname{erf}(x) &= \frac{2}{\sqrt{\pi}} \int_0^x \sum_{k=0}^{\infty} \frac{(-t^2)^k}{k!}\, dt \\ &= \frac{2}{\sqrt{\pi}} \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{k!(2k+1)},\end{aligned}$$

giving us an infinite series for the error function. Thus, thanks to Taylor series, approximations to the Gaussian CDF can be computed by statistical software, and placed in those mysterious statistics tables with which every student of statistics is familiar.

13.2.1. *Standardisation.* The standard Normal distribution is a Normal distribution with mean 0 and variance 1, $\mathcal{N}(0,1)$. It is possible to *standardise* a random variable, $X \sim \mathcal{N}(\mu, \sigma^2)$, by first subtracting its mean, μ (which adjusts the mean to 0 without changing the variance), then dividing by its standard deviation, σ, (which does not change the mean but shrinks the variance from σ^2 to 1), yielding,

$$Z = \frac{X - \mu}{\sigma}.$$

Since these transformations do not change the type of distribution, and a Gaussian is uniquely defined by its mean and variance, the transformed random variable, Z, is distributed according to the standard normal distribution.

13.3. **Chi-squared Distribution.** The chi-squared (χ^2) distribution expresses a probability distribution over the sums of squares of k normally distributed random variables. That is, the distribution of Y where $Y = X_1^2 + X_2^2 + \cdots + X_k^2$ for k, and where the $X_i \sim \mathcal{N}(0,1)$ are i.i.d[32] random variables. Note that by the central limit theorem, the chi-squared distribution converges to a normal distribution as $k \to \infty$. We write $Y \sim \chi_k^2$ where k is the number of variables in the sum, also known as the *degrees of freedom*. When the random variable Y takes on a value, that value is like the dial of radius of a k-dimensional sphere, where the k random variables are free to vary. The chi-squared distribution is the basis of some common statistical tests, and is also crucial to the derivation of other distributions, such as Student's t-distribution. The pdf of a chi-squared distribution is,

$$p(y;k) = \frac{1}{2^{\frac{k}{2}}\Gamma\left(\frac{k}{2}\right)} y^{\frac{k}{2}-1} e^{-\frac{y}{2}},$$

where the parameter, k, is the number of degrees of freedom (number of variables in the sum), and Γ is the gamma function. In the unique case that $k = 2$, the expression simplifies to $p(y;2) = \frac{1}{2}e^{-y/2}$. We take the time to derive the pdf of the chi-squared distribution, as it involves several of the topics we have covered already.

[32] independent and identically distributed

In the simplest case, $k = 1$, we have $Y = X^2$. Since X is a standardised Gaussian variable, its square makes all previously negative values positive. The shape of the pdf is therefore something like a stretched half-bell curve. We can derive the pdf of the transformed variable using the change of variables formula. Because of the symmetry of the squared variable, we have, $f_Y(y) = 2f_X(f^{-1}(y))\left|\left(\frac{dx}{dy}\right)\right|$. Now, $X = \sqrt{Y}$, so $dx/dy = 1/2 \cdot y^{-1/2}$. Hence,

$$f(y;1) = 2\frac{1}{\sqrt{2\pi}}e^{-(\sqrt{y})^2/2}\frac{1}{2}y^{-1/2} = \frac{1}{\sqrt{2}\Gamma(1/2)}e^{-y/2}y^{-1/2},$$

where we recall that $\Gamma(1/2) = \sqrt{\pi}$. The kth order case may be derived with a more involved version of the same technique. In the general case, it becomes clear that the variable Y, is directly related to a k-dimensional sphere, with surface area, S, given by,

$$S = \frac{2\pi^{\frac{n}{2}}}{\Gamma(\frac{n}{2})}.$$

This is how the gamma function comes to be part of the function.

13.4. **Student's T-Distribution.** The Student's t-distribution has an interesting history. Student is a pseudonym for the English mathematician who designed it, William Sealy Gosset (1876-1937), unable to attribute his true name due to his employment as a statistician at the Guiness brewery in Dublin. According to Gosset, the t-distribution expresses a distribution of the value of the sample mean of a Gaussian variable given the true standard deviation of the samples is unknown. The probability density function is,

$$f(t) = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}\Gamma(\frac{\nu}{2})}\left(1 + \frac{t^2}{\nu}\right)^{-\frac{\nu+1}{2}},$$

where $\Gamma(v)$ is the gamma function, and ν, the Greek letter nu, are the degrees of freedom. The variable, $t = \frac{\hat{x}-\mu}{s/\sqrt{n}}$, where $\hat{x} = (x_1 + \cdots + x_n)/n$ is the sample mean, μ is the true mean, $s^2 = \frac{1}{N-1}\sum_{i=1}^{n}(x_i - \hat{x})^2$ is the sample variance, and n is the sample size. The degrees of freedom is therefore $\nu = n - 1$. Thus, the Student's t-distribution distributes t, the sample mean.

14. Hypothesis Testing

14.1. **Standard Error.** Standard error is the standard deviation of the sample mean, $\bar{X} = \frac{1}{n}(X_1 + X_2 + \cdots X_n)$ from n independent samples, $X_1, X_2, \ldots, X_n$. The total,

$$T = X_1 + X_2 + \cdots + X_n,$$

has variance, $\sigma_T^2 = n\sigma^2$ for true standard deviation σ. The variance of $\bar{X} = T/n$ is therefore,

$$\sigma_{\bar{X}}^2 = \frac{1}{n^2}n\sigma^2 = \sigma^2/n.$$

We write finally the standard error as,

$$\mathrm{SE}_{\bar{X}} = \sigma/\sqrt{n}.$$

Clearly as $n \to \infty$, the standard error goes to 0, reflecting that the sample mean converges to the true mean. Note that in practice σ is unknown and must be approximated by the sample variance.

14.2. **Confidence Intervals.** A confidence interval is the interval, expressed in standard deviations, around a sample mean, $\bar{X}$, known to contain the true mean, μ, to a desired degree of certainty. Because of the central limit theorem, we know the sum of N random variables is normally distributed, hence the sample mean also. The standard error statistic tells us the standard deviation of this distribution. Because the Normal cdf has no closed form, it must be computed numerically. In practice, it is necessary to have a lookup table of precomputed probabilities from a standard normal distribution. The value of our variable can then be normalised and the corresponding probability retrieved. In a standard Normal distribution, $Pr(Z \leq 1.96) = 0.9775$ (approximately), and due to symmetry, $Pr(Z \geq -1.96) = 0.9775$. That is, 95% of the distribution sits in the interval $[-1.96, 1.96]$. In other words, Z has a 95% chance of being within 1.96 standard deviations ($\sigma = 1$ when standardised) of the mean. Thus, we may solve for our own 95% confidence interval by writing,

$$1.96 = \frac{\bar{X} - \mu}{\sigma}.$$

Rearranging gives,

$$\mu = [\bar{X} - 1.96 \times \mathrm{SE}, \bar{X} + 1.96 \times \mathrm{SE}],$$

which is the general form for a 95% confidence interval. Confidence intervals of arbitrary size can be constructed by looking up the normalised value of the desired probability.

14.3. **Degrees of Freedom.** Degrees of freedom refer to the dimensionality of randomness of a statistic. For N random variables, the degree of freedom is N, as this probabilistic vector is random in every dimension. If we were to calculate the sample variance, however, involving the sample mean implies a constraint on the data: given the first $N-1$ observations, we can derive the final one from the value of the mean (known in advance). The variance statistic therefore has $N-1$ degrees of freedom. In a simple (bivariate) linear regression, the normal equations for optimising the bias and gradient weights impose two constraints on the data, thus these statistics have $N-2$ degrees of freedom[33]

[33]In multiple regression with a general p dimensions, each coefficient has $n-p$ degrees of freedom.

14.4. **Hypothesis Testing.** We can use our knowledge of statistics to perform tests on sample data. The formal framework for this is to propose two hypotheses, one the null (or default) hypothesis, H_0, the other the alternative hypothesis, H_1. The procedure is to compute a statistic to challenge the statistical implications of the null hypothesis, producing a probability that the sample data is consistent with H_0. If the probability is lower than some conventional threshold (usually 0.05 or 0.1), H_0 is rejected and H_1 accepted, in a sort of probabilistic proof by contradiction. Erroneously rejecting H_0 is referred to as a type 1 error (false positive); erroneously accepting H_0 is referred to as a type 2 error (false negative).

14.4.1. *Z-test.* Many statistical tests exist, and perhaps the simplest is the Z-test, which can be used to decide whether a set of n samples conforms to a given distribution, by calculating the probability of the sample mean. The calculations are similar to creating a confidence interval, as it involves computing the standard error and (implying the true mean and standard deviation be known) the Z-score,

$$Z = \frac{\hat{x} - \mu_0}{\sigma/\sqrt{n}}.$$

The value of Z is looked up in standard normal tables, yielding a probability. The probability (or p-value) is compared with a conventional threshold (typically 0.05 or 0.1) and the null hypothesis accepted or rejected depending on where it falls. We may interpret the Z-score as asking the question, 'can this data reasonably be believed to have been drawn from our distribution?'

14.4.2. *Student's t-test.* In the absence of population parameters, one may perform Student's t-test and answer slightly different questions. In the one sample case, the t-statistic,

$$t = \frac{\hat{x} - \mu_0}{s/\sqrt{n}},$$

(distance in standard error units around a null mean) is normally distributed by virtue of the central limit theorem, even if the population distribution is not. The calculation involves the sample mean, implying $N-1$ degrees of freedom. In a *paired* two-sample case, where data comes in before/after, dependent pairs, the one-sample test is performed on the mean of the difference, $\mu_D = \mu_X - \mu_Y$, with $\mu_0 = 0$ (the null hypothesis postulating no *a posteriori* effect. When the samples are independent (treated/untreated), the formulation changes slightly, and the number of degrees of freedom is doubled (twice the data, twice the constraints). The formulation further changes when the independent sample sizes are uneven.

14.4.3. *Pearson's Chi-Squared Test.* This test checks for bias in categorical variables. The null hypothesis is that the n categories of a variable are uniformly distributed. If, for example, we have a die producing a set of sample data from N throws, we can compute the difference between the observed total of each category, O_i, $i = 1, \ldots, 6$, and the expected total for a fair die, $E_i = \frac{1}{6} \times N$. This difference is assumed to be a continuous, normally

distributed random variable. The normalised squared error for each variable is $(O_i - E_i)^2/E_i$. We can therefore sum these squared errors and use a chi-squared distribution with $k = 5$ degrees of freedom[34] to give us a probability for the null hypothesis.

15. The Central Limit Theorem

15.1. **Moment-Generating Functions.** The moment-generating function (MGF) of a random variable, X, is defined as,

$$\begin{aligned} M_X(t) = \mathbb{E}[e^{tX}] &= \mathbb{E}\left[\sum_{k=0}^{\infty} \frac{(tX)^k}{k!}\right] \\ &= \sum_{k=0}^{\infty} \frac{\mathbb{E}[X^k]t^k}{k!}. \end{aligned}$$

Such a function is an alternative specification of a distribution to a pdf. By Lévy's continuity theorem, convergence in MGF implies convergence in distribution, a result that underpins the proof of the central limit theorem. The function is 'moment-generating' in that the coefficients of the series expansion are the central moments of the distribution. The MGF of a *sum* of N i.i.d variables, $T = X_1 + X_2 + \cdots + X_N$ is,

$$\begin{aligned} M_T(t) = \mathbb{E}[e^{tX_1+tX_2+\cdots+tX_N}] &= \mathbb{E}[e^{tX_1}]\mathbb{E}[e^{tX_2}]\cdots\mathbb{E}[e^{tX_N}] \\ &= M_X(t)^N. \end{aligned}$$

The MGF of a Gaussian distribution[35] is,

$$M_z(t) = \mathbb{E}[e^{tZ}] = \int_{-\infty}^{+\infty} e^{zt}\frac{1}{\sqrt{2\pi}}e^{-z^2/2}\,dz\,.$$

If we consider the exponents, we have $e^{zt-z^2/2}$. Completing the square gives $e^{-z^2/2+zt-t^2/2+t^2/2} = e^{(z-t)^2/2}\cdot e^{t^2/2}$. Therefore we have,

$$\begin{aligned} M_z(t) &= e^{t^2/2}\int_{-\infty}^{+\infty} \frac{1}{\sqrt{2\pi}}e^{(z-t)^2/2}\,dz \\ &= e^{t^2/2}\cdot 1, \end{aligned}$$

since the integrated expression defines $\mathcal{N}(z;t,1)$.

[34] In such problems, the degrees of freedom is $n-1$, where n is the number of categories. This is due to the fact that once the first $n-1$ variables are given, the final one is determined because the sum is constant.

[35] Here standardised for simplicity.

15.2. **Proof of the Central Limit Theorem.** The central limit theorem (CLT) is the crowning jewel of statistics–a profound result, with far-reaching applications. It explains, at least in part, the uncanny ubiquitousness of the bell curve in nature. Its proof is therefore the demonstration of a universal truth. This is where Laplace's statement that 'probability is common sense reduced to calculation' resonates most strongly. The theorem states that the sum of n random variables, whatever their individual distributions, converges to a Gaussian distribution as $n \to \infty$. If we then suppose natural phenomena to be aggregations of many random events, the central limit theorem explains how the bell curve appears with such regularity[36].

Given $X_1, X_2, ...X_n$ are i.i.d. random variables with mean 0, variance σ^2, and moment-generating function, $M_x(t)$, denote their standardised sum, $Z = (X_1 + X_2 + \cdots X_n)/\sqrt{n\sigma^2}$. Therefore,

$$M_Z(t) = \left(M_x\left(\frac{t}{\sqrt{n\sigma^2}} \right) \right)^n.$$

A Taylor approximation for M_x is,

$$M_x(s) = M_x(0) + sM_x(0) + \frac{1}{2}s^2 M_x''(0) + o(s^2),$$

where $o(s^2)$ indicates a function that shrinks faster than a quadratic function as $s \to 0$. By definition, $M_x(0) = 0$, $M_x'(0) = 1$, and $M_x''(0) = \sigma^2$. Combining these results gives,

$$\begin{aligned} M_Z(t) &= \left(1 + \frac{t^2/2}{n} + o\left(\frac{t^2}{n\sigma^2} \right) \right)^n \\ &\to e^{t^2/2}, \ n \to \infty, \end{aligned}$$

that is, the MGF of a Normal distribution. Thus, the MGF, $M_Z(t)$, converges to the MGF of a Normal distribution. Hence, by the Lévy continuity theorem, it converges in distribution to a Gaussian.

16. The Law of Large Numbers

In this section we prove one of the fundamental laws of statistics–the law of large numbers–first deriving the Markov and Chebyshev inequalities that are used in the proof.

16.1. **Markov Inequality.** The Markov inequality expresses a general property of probability distributions. An elegant proof exists, but it may be derived simply by the following observation: it is not possible for $(1/n)$th of a population to be greater than n times the average value. Expressed mathematically this is,

[36] A wonderful mechanical demonstration of the central limit theorem is the *Galton board*. These can often be found in science museums.

$$\Pr(X \geq a) \leq \frac{\mathbb{E}[X]}{a}$$

16.2. **Chebyshev Inequality.** The Chebyshev inequality is another general result for probabilities, expressing a bound on the probability of a random variable, X, straying from its mean, μ, by more than k standard deviations. To derive this, we first define a random variable, $Y = (X - \mu)^2$, and constant, $a = (k\sigma)^2$. Then, by the Markov inequality,

$$\Pr((X - \mu)^2 \geq (k\sigma)^2) \leq \frac{\mathbb{E}[(X - \mu)^2]}{(k\sigma)^2},$$

which we may rewrite as,

$$\begin{aligned} \Pr(|X - \mu| \geq k\sigma) &\leq \frac{\mathrm{Var}(X)}{k^2\sigma^2} \\ &= \frac{1}{k^2} \end{aligned}$$

16.3. **Law of Large Numbers.** The law of large numbers states that given a sequence $X_1, X_2, \ldots, X_N$ of i.i.d random variables, the sample average, $\bar{X} = \frac{1}{n}(X_1 + X_2 + \cdots + X_n)$ converges to the true mean, μ, as n grows. That is, $\bar{X}_n \to \mu$ as $n \to \infty$.

Note first that,

$$\begin{aligned} \mathrm{Var}(\bar{X}_n) &= \mathrm{Var}\left(\frac{X_1 + X_2 + \cdots + X_n}{n}\right) \\ &= \frac{1}{n^2}\big(\mathrm{Var}(X_1) + \mathrm{Var}(X_2) + \cdots + \mathrm{Var}(X_n)\big) \\ &= \frac{\sigma^2}{n} \end{aligned}$$

Now, by the Chebyshev inequality,

$$\Pr(|\bar{X}_n - \mu| \geq k\frac{\sigma}{\sqrt{n}}) \leq \frac{1}{k^2}.$$

Choosing $k = \frac{\sqrt{n}}{\sigma}\epsilon$ for any arbitrary choice of ϵ,

$$\Pr(|\bar{X}_n - \mu| \geq \epsilon) \leq \frac{\sigma^2}{n\epsilon^2},$$

that is,

$$\Pr(|\bar{X}_n - \mu| < \epsilon) \geq 1 - \frac{\sigma^2}{n\epsilon^2},$$

which converges to 1 for a sufficiently large choice of n.

17. Information Theory

This section presents the basics of information theory.

17.1. **Entropy.** The entropy of a random variable, X, is a measure of its uncertainty. It is a core tenet of information theory, the science underpinning coding and signal processing. Entropy measures the average number of bits required to encode an alphabet, and is the lower bound on compression (coding). Entropy is defined as,

$$\mathcal{H}(X) = \sum_{x \in X} p(x) \log \frac{1}{p(x)}.$$

Entropy clearly must be positive, and is minimised for a distribution where a single event has probability 1, that is, a deterministic variable. In this case, the entropy of the variable is 0, that is, minimally uncertain. Hence, $\mathcal{H}(X) \geq 0$. When we have binary events, $x \in \{0, 1\}$, we have the binary entropy function, which simplifies to,

$$h_2(p) = -p \log p - (1-p) \log(1-p).$$

17.2. **Kullback-Liebler Divergence.** Kullback-Liebler (KL) divergence is a measure of the disparity between two probability distributions, p and q, written,

$$\mathcal{D}(p||q) = \sum_{k=1}^{K} p_k \log \frac{p_k}{q_k}$$

KL divergence can therefore be interpreted as the average number of *additional* bits required to encode an alphabet with chosen distribution q, given true distribution p. Like entropy, divergence must be greater than or equal to zero. To see this, write,

$$\begin{aligned} -\mathcal{D}(p||q) &= -\sum_{k=1}^{K} p_k \log \frac{p_k}{q_k} \\ &= \sum_{k=1}^{K} p_k \log \frac{q_k}{p_k} \\ &\leq \log \left(\sum_{k=1}^{K} p_k \frac{q_k}{p_k} \right) \qquad (11) \\ &= \log 1 = 0, \end{aligned}$$

where the inequality in (11) comes from applying Jensen's inequality, since log is a concave function. A corollary to this is that the distribution maximising entropy for discrete variables is the uniform distribution, since,

$$\begin{aligned} 0 \leq \mathcal{D}(p||u) &= \sum_{k=1}^{K} p_k \log \frac{p_k}{u_k} \\ &= \sum_{k=1}^{K} p_k \log p_k - \sum_{k=1}^{K} p_k \log \frac{1}{K} \\ &= -\mathcal{H}(p) + \log K \end{aligned}$$

17.3. **Mutual Information.** Mutual information uses the KL divergence to measure the difference in the entropy of a random variable, X, before and after a second variable, Y, (on which X may be dependent) is introduced. The formula for mutual information is,

$$\begin{aligned} I(X;Y) \triangleq \mathcal{D}(p(X,Y)||p(X)p(Y)) &= \sum_x \sum_y p(x,y) \log \frac{p(x,y)}{p(x)p(y)} \\ &= \mathcal{H}(Y) - \mathcal{H}(Y|X) \\ &= \mathcal{H}(X) - \mathcal{H}(X|Y), \end{aligned}$$

and so it quantifies how much knowing about one variable tells us about the other (note the formula is symmetric). That is, how much uncertainty is lifted from X by learning about Y, and vice versa. In other words, it measures *the extent to which one random variable becomes deterministic once we learn about another*. For this reason, mutual information can be used as a more sophisticated correlation measure, as it may also reveal non-linear dependencies.

18. Convex Optimisation

18.1. **Convexity.** Convexity is a useful property in optimisation because it guarantees that any local optimum of a function is a *global* optimum.

18.1.1. *Convex Sets.* A set of points, S, in a vector space is convex if for any two points x_1 and x_2 in S, all points in-between, that is, all points on the straight line connecting x_1 and x_2 are also in S. In a sense, it is a region that does not have any 'holes'. For example, a circular region is convex, but a doughnut is not, nor is a crescent. A line passing through points x_1 and x_2 can be written, $x_1 + c(x_2 - x_1)$, for some constant c. Formally, a set S is convex if,

$$S = \Big\{ x_1, x_2 \in S \implies (1-\lambda)x_1 + \lambda x_2 \in S \Big\},$$

for all $\lambda \in [0, 1]$, that is, all points along the chord between the two points are also part of the convex set. A convex hull of a set of points is the smallest convex region enclosing those points.

18.1.2. *Convex Functions.* A function, $f(x)$, is convex on an interval if,

$$f((1-\lambda)x_1 + \lambda x_2) \leq (1-\lambda)f(x_1) + \lambda f(x_2),$$

for all $\lambda \in [0,1]$. That is, the curve lies below a line drawn between x_1 and x_2. A function with more than one turning point in a domain cannot be convex. Examples of convex functions are e^x, x^2, etc. *Strict* convexity occurs when there is a strict inequality. *Concavity* is the same property with the inequality reversed. Hence, $f(x)$ convex $\iff$ $-f(x)$ concave. A linear function is both concave and convex.

18.1.3. *Jensen's Inequality.* Jensen's inequality essentially extrapolates the definition of convexity to n dimensions, stating,

$$f\left(\sum_{i=1}^{N} \lambda_i \mathbf{x}_i\right) \leq \sum_{i=1}^{N} \lambda_i f(\mathbf{x}_i),$$

for convex function, f, $\mathbf{x} \in \mathcal{R}^N$, and $\sum_{i=1}^{N} \lambda_i = 1$. For $N = 2$, we have,

$$f(\lambda_1 x_1 + \lambda_2 x_2) \leq \lambda_1 f(x_1) + \lambda_2 f(x_2),$$

for any λ_1, λ_2 such that $\lambda_1 + \lambda_2 = 1$ (definition of convexity). Then suppose, $f\big(\sum_{i=1}^{k} \lambda_i x_i\big) \leq \sum_{i=1}^{k} \lambda_i f(x_i)$. We define,

$$\lambda_j = \begin{cases} \lambda_i & j = 1:k-1 \\ \lambda_k + \lambda_{k+1} & j = k \end{cases} \text{ and } x_j = \begin{cases} x_i & j = 1:k-1 \\ \frac{\lambda_k}{\lambda_k+\lambda_{k+1}} x_k + \frac{\lambda_{k+1}}{\lambda_k+\lambda_{k+1}} x_{k+1} & j = k \end{cases},$$

then,

$$\begin{aligned} f\left(\sum_{i=1}^{k+1} \lambda_i x_i\right) &= f\left(\sum_{j=1}^{k} \lambda_j x_j\right) \\ &\leq \sum_{j=1}^{k} \lambda_j f(x_j) \\ &= \sum_{i=1}^{k-1} \lambda_i f(x_i) + (\lambda_k + \lambda_{k+1}) f\left(\frac{\lambda_k}{\lambda_k + \lambda_{k+1}} x_k + \frac{\lambda_{k+1}}{\lambda_k + \lambda_{k+1}} x_{k+1}\right) \\ &\leq \sum_{i=1}^{k-1} \lambda_i f(x_i) + (\lambda_k + \lambda_{k+1}) \frac{\lambda_k}{\lambda_k + \lambda_{k+1}} f(x_k) + \frac{\lambda_{k+1}}{\lambda_k + \lambda_{k+1}} f(x_{k+1}) \\ &= \sum_{i=1}^{k+1} \lambda_i f(x_i), \end{aligned}$$

and Jensen's inequality follows from the principle of mathematical induction.

18.1.4. *Sums of Convex Functions.* An important result is that if we sum two convex functions, the result is convex also. To see this in two dimensions, define convex functions $f(x)$ and $g(x)$, and their sum $h(x)$. Then,

$$\begin{aligned} h((1-\lambda)x_1 + \lambda x_2) &= f((1-\lambda)x_1 + \lambda x_2) + g((1-\lambda)x_1 + \lambda x_2) \\ &\leq (1-\lambda)f(x_1) + \lambda f(x_2) + (1-\lambda)g(x_1) + \lambda g(x_2) \\ &= (1-\lambda)(f(x_1) + g(x_1)) + \lambda(f(x_2) + g(x_2)) \\ &= (1-\lambda)h(x_1) + \lambda h(x_2). \end{aligned}$$

If one of the functions, say $f(x)$, is linear, it is sufficient to note that $f((1-\lambda)x_1+\lambda x_2) = (1-\lambda)f(x_1) + \lambda f(x_2) \leq (1-\lambda)f(x_1) + \lambda f(x_2)$, and the proof holds as before. In fact, a linear function satisfies the definition for convexity *and* concavity.

18.2. **Necessary and Sufficient Conditions for Optimality.** A basis for designing optimisation algorithms are the sets of conditions that tell us when we have found a local or global optimum. Necessity and sufficiency are formal logical terms. If a condition S implies a condition N, we write $S \implies N$, meaning if S is true, then N is also. Thus, we say S is a *sufficient* condition for N, even though N may be true without S. It also means that if N is not true, S cannot be true, even though N may be true independently when S is false. Thus, we say N is a *necessary* condition for S. When the implication runs both ways (if and only if), we write $S \iff N$, and say that S is a *necessary and sufficient* condition for N, and vice versa.

18.3. **Unconstrained Problems.** When we have a differentiable objective function and no constraints, the optimality conditions are simplified. Consider the second derivative test for one-dimensional problems. From first principles, the second derivative at a stationary point ($f'(x^*) = 0$) is,

$$f''(x^*) = \lim_{h\to 0} \frac{f'(x^*+h) - \cancelto{0}{f'(x^*)}}{h} = \lim_{h\to 0} \frac{f'(x^*+h)}{h}.$$

If this quantity is positive, we must be at a minimum, as the function is increasing to the right of x^*. If negative, x^* is a maximum. Similar conditions exist in the multi-dimensional setting. First, we must have a zero gradient,

$$\nabla_{\mathbf{x}} f(\mathbf{x}^*) = \mathbf{0},$$

and we must have a positive semi-definite Hessian matrix, that is,

$$\mathbf{v}^T \cdot \nabla_{\mathbf{xx}} f(\mathbf{x}^*) \cdot \mathbf{v} \geq 0, \forall\ \mathbf{v} \in \mathbb{R}^N,$$

the intuition being that in every direction outward from $\mathbf{x}^*$, the gradient will be positive, indicating a minimum–just as it was in the one-dimensional case. For a maximum, we instead require a negative semi-definite Hessian. These are necessary and sufficient conditions for optimality.

18.4. **Equality Constraints.** Note that geometrically, the gradient at a point is normal to the *level curve* or contour. The directional derivative of *differentiable* f in the direction of some vector, $\mathbf{v}$ is,

$$\begin{aligned}\nabla_{\mathbf{v}}\mathbf{f}(\mathbf{x}) &= \lim_{h\to 0}\frac{f(\mathbf{x}+h\mathbf{v})-f(\mathbf{x})}{h} \\ &= \nabla\mathbf{f}(\mathbf{x})\cdot\mathbf{v} = \Delta x_1\frac{\partial f}{\partial x_1}+\Delta x_2\frac{\partial f}{\partial x_2}+\cdots+\Delta x_n\frac{\partial f}{\partial x_n}.\end{aligned} \tag{12}$$

If we choose $\mathbf{v}$ in the direction of a contour, then by definition $f(\mathbf{x}+h\mathbf{v}) = f(\mathbf{x})$, hence the directional derivative is 0, and so $\nabla\mathbf{f}(\mathbf{x})\cdot\mathbf{v} = 0$, hence the gradient is normal to the contour. Now consider an optimisation problem with a single equality constraint,

$$\begin{array}{rl}\min\limits_{\mathbf{x}\in\mathcal{X}} & f(\mathbf{x}) \\ \text{subject to} & h(\mathbf{x}) = 0\end{array},$$

noting it is always possible to ensure an equality constraint equals 0 by subtracting the right-hand side terms. Given a feasible point (satisfying the constraint), $\mathbf{x}_F$, we can decrease the objective function by taking a step, $\delta\mathbf{x}$, such that the directional derivative, $\delta\mathbf{x}\cdot(-\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x}_F)) > 0$. From (12) we know this implies $f(\mathbf{x}_F+\delta\mathbf{x}) < f(\mathbf{x}_F)$. Now, in order to move to a new feasible point, we must choose $\delta\mathbf{x}$ such that we move along the constraint surface, that is, parallel to the level curve at 0 on the surface, $h(\mathbf{x})$. We know that a vector running parallel to a level curve will be normal to the gradient. Thus, we need $\delta\mathbf{x}\cdot\nabla_{\mathbf{x}}\mathbf{h}(\mathbf{x}) = 0$. Combining these ideas, consider where $-\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x}_F) = \lambda\nabla_{\mathbf{x}}\mathbf{h}(\mathbf{x}_F)$ for some scalar λ. Multiplying by $\delta\mathbf{x}$, we get the stopping condition,

$$\delta\mathbf{x}\cdot(-\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x}^*)) = \delta\mathbf{x}\cdot\lambda\nabla_{\mathbf{x}}\mathbf{h}(\mathbf{x}^*) = 0,$$

that is, where the gradient of $f(\mathbf{x})$ runs parallel to the gradient of the level curve. At such a point $\mathbf{x}^*$, we have it that the next feasible point (normal to the constraint surface) will not further minimise the objective function. This stopping condition is incorporated in the method of Lagrange multipliers.

18.5. **The Method of Lagrange Multipliers.** The method of Lagrange multipliers, discovered by Italian-French mathematician Joseph Louis Lagrange (1736-1813), takes a constrained problem and creates a new, *unconstrained* objective function that incorporates these conditions, by introducing auxiliary variables called the Lagrange multipliers. Solving it will solve the original, constrained problem. We define the *Lagrangian*,

$$\mathcal{L}(\mathbf{x},\lambda) = f(\mathbf{x}) + \lambda h(\mathbf{x}).$$

Clearly, at an optimum, $\mathcal{L}(\mathbf{x}^*,\lambda^*) = f(\mathbf{x}^*)$, since $\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^*,\lambda^*) = \mathbf{0} \implies -\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x}^*) = \lambda\nabla_{\mathbf{x}}\mathbf{h}(\mathbf{x}^*)$, the optimality condition from above, and $\nabla_{\lambda}\mathcal{L}(\mathbf{x}^*,\lambda^*) = 0 \implies h(\mathbf{x}) = 0$, the equality constraint. We finally require $\mathbf{v}^T\cdot\nabla_{\mathbf{xx}}f(\mathbf{x}^*)\cdot\mathbf{v} \geq 0, \forall\ \mathbf{v} : \nabla_{\mathbf{x}}\mathbf{h}(\mathbf{x}^*)^T\mathbf{v} = 0$, that is, positive semi-definiteness for the Hessian (though we need only consider vectors

v along the constraint contour). To illustrate, consider maximising the Shannon entropy equation, $f(p_1, \ldots, p_n) = -\sum_i p_i \log p_i$, for some probability distribution, **p**, subject to the constraint $\sum_i p_i = 1$. Introducing a Lagrange multiplier for the constraint gives us,

$$\mathcal{L}(\mathbf{x}, \lambda) = -\sum_i p_i \log p_i + \lambda\bigg(\sum_i p_i - 1\bigg),$$

which may be differentiated, giving us an equation for each p_i that show (as before) that the maximising distribution is the uniform distribution, that is, $p_i = 1/n$. It is easy to extend this method to multiple equality constraints. In this case we can write the Lagrangian,

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \boldsymbol{\lambda}^T \mathbf{h}(\mathbf{x}),$$

and the optimality condition $\nabla_\lambda \mathcal{L}(\mathbf{x}^*, \boldsymbol{\lambda}^*) = \mathbf{0} \implies h_i(\mathbf{x}^*) = 0$ for each constraint i.

18.6. **Inequality Constraints.** Consider an optimisation problem with a single inequality constraint,

$$\begin{array}{ll} \min\limits_{\mathbf{x}\in\mathcal{X}} & f(\mathbf{x}) \\ \text{subject to} & g(\mathbf{x}) \leq 0 \end{array}.$$

Such a constraint may be either *active* or *inactive*. When it is inactive, the local optimum falls within the feasible region, and the problem is effectively unconstrained. In this case, the optimality conditions are identical to the unconstrained case. Note, however, our conditions do not tell us how to predetermine whether a constraint is active, nor do they (as previously noted) tell us how to find the optimum. At most, the optimality conditions inform the design of optimisation algorithms by telling us what to look for. When the constraint is *active*, the local optimum lies outside of the feasible region. We note first that the constrained optimum must lie on the constraint surface, as this is closest to the local optimum. If this were not the case, it would imply the existence of another (closer) local optimum inside the feasible region. So, in this case we effectively have an equality constraint as before. The optimality condition is therefore,

$$-\nabla_{\mathbf{x}} \mathbf{f}(\mathbf{x}) = \mu \nabla_{\mathbf{x}} \mathbf{h}(\mathbf{x}), \tag{13}$$

for $\mu > 0$. The scalar must be positive as it ensures the gradient is facing 'outwards' of the feasible region, indicating that we are at the extreme point closest to the local optimum. The Karush-Kuhn-Tucker (KKT) conditions capture these two cases in a general set of conditions.

18.7. **KKT Conditions.** The Karush-Kuhn-Tucker (KKT) conditions are a general set of necessary and sufficient conditions for non-linear optimisation problems constrained by multiple equality and inequality constraints. For optimisation problem,

$$\begin{array}{ll} \min\limits_{\mathbf{x}\in\mathcal{X}} & f(\mathbf{x}) \\ \text{subject to} & g(\mathbf{x}) \leq 0 \end{array},$$

we form the Lagrangian,

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \mu g(\mathbf{x}),$$

and necessary and sufficient conditions for optimality are,

(1) $\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^*, \mu^*) = \mathbf{0}$. Thus, when the inequality constraint is inactive, $\mu^* = 0$ and $\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x}) = \mathbf{0}$. When the constraint is active, $\mu^* > 0$ and $-\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x}) = \mu\nabla_{\mathbf{x}}\mathbf{h}(\mathbf{x})$ as in equation (13).
(2) $\mu^* \geq 0$, with equality for an inactive constraint.
(3) $g(\mathbf{x}^*) \leq 0$ with equality for an active constraint.
(4) $\mu^* g(\mathbf{x}^*) = 0 \implies g(\mathbf{x}^*) = 0$ when $\mu^* > 0$ for an active constraint.
(5) Positive semi-definite constraints on $\nabla_{\mathbf{xx}}\mathcal{L}(\mathbf{x}^*, \mu^*)$.

Clearly, extending this to multiple inequality constraints, and further incorporating equality constraints is a trivial matter, giving a Lagrangian function,

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \boldsymbol{\mu}^T\mathbf{g}(\mathbf{x}) + \boldsymbol{\lambda}^T\mathbf{h}(\mathbf{x}).$$

19. Computability and Complexity

19.1. **Decision Problems.** Decision problems, as studied in computability and computational complexity theory, are characterised by a question posed on some arbitrary inputs that has a boolean (`TRUE-FALSE`) answer. The question itself is the decision *problem*, which when paired with a parameter set is called a problem *instance*. For example, a decision *problem* might be to determine if some number x is prime or not. A particular problem *instance* might be very trivial, for example, 'is the number 2 prime?'. Exact methods of primality testing do exist, however, making primality testing a *decidable* problem. In other words, primality is *computable* for any input, even if in practice it may take a long time to compute. On the other hand, there exist decision problems that are *undecidable*. An example of this is the *halting problem*. It was in demonstrating the undecidability of the halting problem that the great English mathematician, Alan Turing (1912-1954), devised an abstract model of the computer, the Turing machine.

19.2. **The Halting Problem.** The halting problem was one of the first problems shown to be undecidable, that is, no algorithm exists to solve it for all possible inputs. The halting problem is concerned with determining whether an arbitrary algorithm (in the form of some description), provided with arbitrary inputs, will terminate (halt) or not. For example, a trivial input algorithm might be 'divide the input by 2'. This single-instruction algorithm can conceivably be analysed by any number of halting 'analysers' and successfully be shown to terminate for any input. However, an algorithm solving the halting problem will need to decide termination for *any* algorithm. It is tempting to suggest simply simulating the input

algorithm and declaring a halt when it terminates. The problem here is that the algorithm may never terminate, and it is impossible to know whether it will *not* terminate in a finite amount of time (as it might just be mistaken for a very long-running–yet finite–algorithm). Waiting an infinitely long time to determine the halting problem is against the rules.

Algorithm 1 Pseudocode illustrating a paradox arising from a halting algorithm.

```
1: procedure HALT(algorithm, parameters)
2:     if /*algorithm halts with parameters*/ then
3:         return TRUE
4:     else
5:         return FALSE
6: procedure HALT+(algorithm)
7:     if HALT(algorithm, algorithm) then
8:         LoopForever()
9:     else
10:        Terminate()
11: HALT+(HALT+) /*undecidable*/
```

Turing showed the halting problem to be undecidable with the following proof by contradiction. Suppose there is an algorithm HALT that takes as input some complete description of an algorithm along with some compatible parameters (Algorithm 1) and solves the halting problem, that is, determines in finite time whether this arbitrary algorithm terminates given the set of inputs. Imagine another algorithm HALT+ that first runs this HALT algorithm, and if the result is `TRUE` (halts), it loops forever (does not halt), otherwise, if `FALSE` (does not halt), it halts. Finally, consider calling HALT+ on itself. Now we see the contradiction–if the HALT subroutine returns `TRUE` for HALT+(HALT+), then HALT+(HALT+) loops forever and does not halt. If HALT returns `FALSE`, then HALT+(HALT+) instead terminates. So, HALT+ always negates whatever decision HALT makes, making its behaviour undecidable by HALT, and therefore HALT does not solve the halting problem, making the problem undecidable.

19.3. **Turing Machines.** To complete his proof on the halting problem, Turing required an abstract model of computation on which to base his formal reasoning. Clearly, it would be infeasible to make assertions about the behaviours of algorithms without a clear definition of what an algorithm actually is. Turing therefore came up with the Turing machine, an abstract computing model generalising the finite state machine (FSM) or automaton[37], sufficiently general to simulate any computer program or algorithm. A Turing machine consists of an infinitely (arbitrarily) long tape divided into cells. A read-write head focuses on one cell at a time, and depending on the cell contents (symbol) and the current

[37]A finite state machine is an abstract model of computation consisting of a finite number of states, representing actions. Movement between states is dictated by the conjunction of an input value and the identity of the current state. Finite state machines are usually visualised with a diagram.

internal state of the machine, the machine writes a symbol, moves the tape left or right, and transitions to a new internal state. Formally, the transition function, δ, of a Turing machine can be written,

$$\delta : S \times Q \to S \times \{\text{left}, \text{right}\} \times Q,$$

that is, given a current read symbol $\in S$ and internal state $\in Q$, the transition function defines a write symbol, left or right movement of the tape, and a new internal state. Any symbol set may be used (though binary code is regularly used to illustrate) without changing the computing power of the machine. Intuitively, the tape is providing the machine with memory, and the transition function defines conditional instruction sets to be performed. It may still be a stretch, however, to fathom that this simple design is capable of performing any algorithm, procedure, or computation imaginable.

19.3.1. *Universal Turing Machines.* If we take a Turing machine to be a model of computation for some function, a *universal Turing machine* (UTM) is a Turing machine whose instruction set is configured to be able to run any Turing machine (whose own instruction sets can be encoded as strings in the UTM symbol set) and any arbitrary input. Thus, we would have an interpreter for any Turing machine, and with it a model for a stored-program *computer*, rather than the more abstract model of *computation* that Turing machines represent. This model is the original concept behind all computer code (software) as executed on computer hardware. In fact, almost all modern computers can be described as universal Turing machines. A programming language is said to be *Turing complete* if it is capable of simulating any Turing machine, making it a universal Turing machine. All imperative programming languages (Python, C++, Java) have this property–all that is required is the ability to execute instructions conditionally (for example with 'if' statements) and to allocate an arbitrary amount of memory. It is strongly argued that the universal Turing machine prefaced John von Neumann's design for the stored-program computer, the von Neumann architecture, the basis of all modern computers.

19.3.2. *Non-deterministic Turing Machines.* A non-deterministic Turing machine (NTM) is a Turing machine for which multiple operations may exist for each input-state pair. The NTM can pursue each of these alternatives simultaneously and at no extra cost. Thus, an operation with exponentially growing alternative paths, such as a tree search, can be computed in linear time (in the depth of the search). It is crucial to note the direct equivalence between how a NTM finds a solution and how a solution can be verified. By pursuing each alternative, the NTM is *guessing* its way through the solution space, without incurring the cost of erring. If a specific solution is provided, a deterministic Turing machine can verify it in the same time (by *guessing* its way through a single solution). Thus, NTM problem solving is exactly equivalent to TM problem verification. Further note that any computation completed on an NTM can also be computed on a deterministic Turing machine, though perhaps not as efficiently. NTMs have no real-world implementation (unlike Turing machines), and remain a purely theoretical idea. Rather, they exist as a model of *solution verification*, a key concept for classifying algorithm complexities.

19.3.3. *Other Turing Machines.* Many other Turing machines exist, such as Quantum Turing machines, which model the behaviour of a quantum computer. An important unresolved question in physics asks whether such a computing model could efficiently simulate any physical system.

19.4. **P versus NP.** P versus NP is one of the most well-known unresolved problems in computer science, and is in fact one of the seven Clay Institute Millennium problems. It refers to whether all problems that may be verified in polynomial time may be solved in polynomial time, and thus whether complexity class P is a subset of NP ($P \neq NP$) or that they are the same class ($P = NP$). The former case is widely considered to be more likely, although theorists as eminent as Donald Knuth (1938-) side with the latter, albeit with the catch that the degree of polynomial may be astronomical.

19.5. **Complexity Classes.** Turing machines also provide the basis for discussing computational complexity classes. Complexity classes are most commonly discussed for decision problems, but equivalents exist for functional problems (for example the FP class), counting problems (#P class), and others.

19.5.1. *Reducibility.* We say that one problem, A, may be reduced to another problem, B, if an algorithm solving B could be used as a subroutine in an algorithm solving A. For example, if we had such an algorithm for B, we could write our algorithm for A in terms of a sequence of executions of an *oracle* (black box) executing the algorithm for B. If a polynomial number of executions were required, we could say A is polynomially reducible to B, formally $A \leq_p B$. This is an important property for comparing algorithms of different complexity classes.

19.5.2. *P class.* The P class is the set of decision problems decidable in polynomial time on a deterministic Turing machine. Polynomial time is a desirable, *tractable* complexity that, as a rule, indicates algorithm run time does not increase *too quickly* (think exponentially) with the size of the problem[38].

19.5.3. *NP class.* A common misconception is that NP stands for 'non-polynomial' time. Ironically, that would be to beg the question of P versus NP[39]. In actual fact, NP stands for *non-deterministic polynomial time*, that is, *NP is the class of algorithms that run in polynomial time on a non-deterministic Turing machine.* However, as noted above, algorithms that run polynomially on an NTM are algorithms whose solution may be verified in polynomial time on a deterministic Turing machine. Hence, the NP class can be alternatively stated as *algorithms that can be verified in polynomial time by a deterministic Turing machine.* Note that all P time algorithms are in the NP class (whatever can be done polynomially on a deterministic Turing machine can be done polynomially on a NTM) though not necessarily vice versa. This disjunction is the subject of one of the most famous

[38]A problem's size is usually expressed in terms of the number of variables.

[39]It is true that known algorithms for NP problems run in super-polynomial (exponential) time on deterministic Turing machines, but the whole point of P versus NP is that it is not yet known if faster algorithms exist.

unsolved problems in mathematics. A problem can be shown to be in the NP class if it can be shown that all candidate solutions can be verified in polynomial time.

19.5.4. *NP-hard class.* NP-hard (non-deterministic polynomial time *hard*) problems are those problems that are *at least as hard as the hardest problems in NP*. In formal terms, this means an NP-hard problem can be reduced to any problem in NP in polynomial time. Note that a problem may be purely NP-hard (not in NP), whereas problems in the intersection between NP and NP-hard are known as NP-complete.

19.5.5. *NP-complete class.* The NP-complete class is the intersection of the NP and NP-hard classes, that is, it contains problems that are reducible to all NP problems, and are additionally verifiable in P time. To prove the NP-completeness of a problem, it is necessary to,

(1) Show it is in NP, by showing its solutions may be efficiently verified
(2) Show it is NP-hard, by showing it can be efficiently reduced to another known NP-complete problem.

The first decision problem proved to be NP-complete is the boolean satisfiability problem or SAT (see the Cook-Levin theorem, 1971), which aims to find a combination of values for literals in a boolean expression such that the overall expression evaluates `TRUE`. The expression is taken to be in conjunctive normal form (CNF)[40]. There are some special cases of SAT where computation is easy, for example if each variable features in exactly one clause, but in general, no efficient algorithm is known to exist. From a purely intuitive perspective, the task lends itself to an exponential runtime, as there are 2^N possible readings of an expression with N variables. Until P versus NP is resolved, it is not known whether there is a reliable shortcut. Richard Karp used this result to find and categorise 21 NP-complete problems, beginning with the 3SAT problem, a variant of SAT in which all clauses contain three literals. It is easy to transform a CNF expression to the 3SAT form, with the introduction of dummy variables. The expression loses logical equivalency, but retains equisatisfiability, meaning that solutions in the 3SAT form are necessary and sufficient for solutions in the original form. The length of the expression triples, hence the reduction step is polynomial. Karp's 21 NP-complete problems further include problems from graph theory (such as clique selection) and 0-1 integer programming. Note that if a problem in the NP-hard or NP-complete classes could be shown to have an efficient, polynomial-time solution algorithm, it would imply all problems in NP have efficient solutions.

20. Discrete Optimisation

20.1. **Integer Programming Problems.** Integer programming is a branch of discrete optimisation. It is closely related to combinatorial optimisation. When we talk about integer programming, we usually refer to *integer linear programming* (ILP), a variant of linear programming with the restriction that decision variables must be integers. Zero-one integer linear programming is a special case where variables take on binary values, and there is no objective function to optimise, only constraints to satisfy. Zero-one ILP is

[40] A conjunction of disjunctions or an `AND` of `OR`s), for example, $(X_1 \vee X_2 \vee \cdots \vee X_N) \wedge (\neg X_1 \vee \dots) \wedge \dots$

NP-complete, on the other hand ILP in general is NP-hard[41]. An integer programming problem may be formulated as,

$$\begin{array}{ll} \text{maximise} & z = \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} = \mathbf{b}, x_i \geq 0, x_i \in \mathbb{Z} \end{array} . \tag{14}$$

There are many famous problems that may be modelled as an integer program.

20.1.1. *The assignment problem.* The assignment problem models the allocation of machines to tasks such that each machine performs exactly one task and each task is performed, minimising a function subject to costs incurred from job-machine pairings. Unlike most ILPs, the assignment problem is in the P class, the famous Hungarian algorithm solving it in cubic time.

20.1.2. *The knapsack problem.* The knapsack problem is an integer programming model for maximising the summed benefit of a range of items, subject to a weight constraint. Each item has an associated value and weight parameter. The decision variables represent the number of each item selected.

20.1.3. *The travelling salesman problem.* The travelling salesman problem (TSP) models a route plan between a set of destinations, each of which must be visited once and only once. The destinations may be represented as vertices on a fully connected graph whose arcs provide the costs of moving between each pair. As an integer programming problem, the objective function minimises the total route cost and the boolean decision variables x_{ij} indicate if destination j is visited from destination i. There are *sum to one* constraints to ensure each vertex is arrived at and departed from exactly once.

20.2. **Solution Methods.** Of the exact solution methods for integer programming problems, the branch and bound algorithm is the most prevalent. However, as integer programming is NP-hard, computing exact solutions is often infeasible, and heuristic techniques are used to approximate optimal solutions efficiently. A variety of other approaches exist, including dynamic programming.

20.2.1. *Branch and bound.* The branch and bound method is an exact algorithm, improving upon an exhaustive search. For example, in an assignment minimisation problem, it is easy to establish a rough lower bound on the optimal solution, simply by ignoring the constraints and taking the minimising value for each variable. Thus, an initial iteration will consider all possible assignments for a first variable, and calculate the (possibly infeasible) lower bound for solutions arising from that. The algorithm continues to consider all possible alternatives for a second variable, and a third, and so on. Once an exact feasible solution is found, it can be used to eliminate all branches with a lower bound exceeding that solution, as even in the best (possibly infeasible) case a solution will be inferior. There are some similarities

[41]The decision version of ILP will also be NP-complete. For example, 'is there a feasible value of $\mathbf{x}$ such that $z \geq k$?' for some value k. Clearly, this can be verified in polynomial time.

between branch and bound and the alpha-beta pruning algorithm used in computer chess, and others.

20.2.2. *Simulated annealing.* Simulated annealing is a popular meta-heuristic algorithm. The aim is to reach a *near-optimal* solution efficiently. Exact solutions may be acquired by chance, though this becomes increasingly unlikely for larger problems. Simulated annealing facilitates both incremental improvement and exploration, according to a schedule of *temperatures* (annealing is a metallurgical term) that decrease linearly to zero round by round. In each round, update steps are taken on the objective function. Improvements are always retained, but updates that reduce the objective function are admitted according to some probability function (such as softmax), for which the probability of accepting inferior updates reduces as the schedule plays out. As a meta-heuristic, it describes an exploration strategy for non-convex objective functions without specifying the update step. It is therefore compatible with many problems.

20.2.3. *Tabu Search.* Tabu (taboo[42]) search is a meta-heuristic algorithm for discrete optimisation problems. From an initial solution, the algorithm considers all possible updates in the solution's neighbourhood. It chooses the best amongst them, even if it is inferior (though it tracks the best overall solution). This update is memorised and from it, a new neighbourhood is generated. Any update that would undo the previous update is regarded as *tabu*, except if it beats the best overall solution. The algorithm proceeds thus, with continued local aspiration searching, until some stopping condition is met. Various stopping conditions are possible, for example if no improvement is made for a streak of iterations.

20.2.4. *Genetic Algorithms.* Genetic algorithms (see also evolutionary algorithms) are yet another *guided random* search technique. The algorithm follows an iterative program whereby each iteration represents a generation of an evolving population. Members of the population are candidate solutions, expressed in some vectorised form, imitating gene sequences. From a randomly generated initial population, the best solutions (according to fitness measure of the objective function) are selected randomly, with the probability of selection proportional to the fitness of the candidate. This allow for the same exploratory behaviour as seen in simulated annealing and tabu search. Next, surviving members are paired off and combined to create a pair of replacements. Thus, each iteration brings about a completely new population, the previous generation having reproduced and died off. The rules of combination of two 'parent' solutions mimic the realities of biologic reproduction, with a randomly chosen crossover point in the vectorised solutions creating a split, with the remainder of the sequences swapped, creating two unique 'children'. Finally, each element of each vector in the new generation is subject to a random perturbation of low probability, to capture the genetic mutation seen in nature. The round completed, the overall fitness of the population is improved, and the algorithm continues until convergence.

[42]From the Tongan word for *prohibited*.

21. Graph Theory

Graphs are mathematical objects with many interesting properties, as well as a vast range of applications. A graph, G = (V, E), consists of a set of vertices (or nodes, or points), V, and a set of edges (or arcs, or lines), E, that connect them. Edges consist of distinct start and end vertices. In general, a graph's properties do not depend on the *placement* of its vertices, nor on the curvature of its edges. As a result, it does not matter how a graph might be drawn. Its *structure* does not vary as long as all edges and vertices are correctly connected. It can be of mathematical interest, however, to compare the ways a graph can be drawn, for example: given a graph, G, can it be drawn without intersecting edges?

There are many structural properties of graphs that may be of interest. A graph is *connected* if every node is reachable from every other node via some path of one or more edges. A disconnected graph may have several *islands* of vertices with no edges to bridge them. A graph of N vertices is *complete* when each of its nodes share an edge with every other node, thus amounting to $1+2+3+\cdots+N = N(N-1)/2$ edges[43]. The density of a graph reflects the degree to which it is complete. Density is therefore given by the number of edges divided by the above formula (the maximum). A graph is *bipartite* if its vertices are in two disjoint sets, with every node of each set connected to at least one node in the other set, but with no arcs between nodes of the same set. A graph is *complete* bipartite if each node connects to every node in the opposite set. A graph is *planar* if it can be drawn in two dimensions such that its edges intersect only at vertices, that is, no criss-crossing. It may therefore be *embedded* on a two-dimensional plane. Graphs are taxonomised in various other ways according to the number of their vertices and the manner by which they are connected.

The edges of a graph may embed information such as direction and weight. If a graph has directed edges (commonly depicted with an arrowhead), it is known as a *directed* graph or *digraph*, otherwise, undirected. If some arcs are bidirectional, the graph is known as a directed multigraph, and, for example, a weight may be stored in each direction (upstream and downstream). A directed acyclic graph (DAG) is a directed graph with no cycles. For example, a tree is always directed and acyclic. If a graph has edge weights, it is called a *weighted* graph, and may model things such as route distances and network flows for which paths of cumulative length are of interest.

21.1. **The Seven Bridges of Königsberg.** A celebrated early result in graph theory is by Euler in 1736, when he solved the *Seven Bridges of Königsberg* problem. He demonstrated no walk could be made over the seven bridges connecting the islands of the town, Königsberg (then Prussia, now Kaliningrad, Russia), such that each bridge was crossed exactly once. Euler reduced the layout of the town from a detailed cartographic map to a graph diagram, with four vertices to represent the four land masses and seven arcs between them for the bridges. He then made the simple observation that apart from the first and

[43] The same formula used in the legend of the schoolboy Gauss, who summed the arithmetic series $1+2+3+\cdots+100$.

last land masses in a tour, each land mass must have an outbound arc for every inbound arc in order to meet the requirement of crossing every bridge exactly once. As every land mass had an odd number of bridges, the problem was logically unsolvable. Of course, Euler could have performed this same reasoning without this novel formalism, but a neat graph depiction brought about some mathematical rigour.

21.2. **The Three Cottages Problem.** The Three Cottages Problem is an old problem of unknown origin whose solution constitutes a fundamental result in graph theory. The problem statement reads:

Suppose we have three cottages in a small village, and three utility stations (for example, water, drainage, and electricity). Is it possible to connect each cottage with each utility with paths that do not cross?

Thus, the cottages and stations form a complete bipartite graph, with three nodes in each set. In the Kuratowski[44] notation, this is $K_{3,3}$, so the problem is asking whether $K_{3,3}$ is planar. A relatively simple solution exists. Note first that because the graph is bipartite, any face has at least four edges, as we cannot draw arcs between nodes of the same set. Thus, we may write $E^* \geq 4F$, where F is the number of faces and E^* is an upper bound on E, the number of edges, noting that an edge may be shared between multiple faces. In fact, we may observe also that each edge is in exactly two faces, as it shares a vertex with two other edges. Thus, $E^* = 2E$, giving $E \geq 2F$. Now, by the Euler characteristic[45], $V - E + F = 2$, where finally V is the number of vertices. By substitution, we obtain $E \leq 2V - 4$. Since in our problem $V = 6$ and $E = 9$, we have found a contradiction, hence the problem has no solution. Wagner's theorem is a fundamental result for planar graphs, stating that no planar graph may contain the *utility graph*, $K_{3,3}$, or K_5 (the fully-connected graph in fives node) as a minor[46].

21.3. **NP-complete Graph Problems.** Many graph-related problems are NP-complete. Here we briefly detail the clique problem and the minimum vertex cover problem, both members of Karp's 21 NP-complete problems.

21.3.1. *The Clique Problem.* A clique is a subset of the vertices of a graph whose subgraph is complete. In a social network, this could represent a group of people who all know each other. Finding a clique of a certain size is an NP-complete problem, as demonstrated graphically in Figure 5. For the 3SAT expression $(y \vee x \vee x) \wedge (\neg x \vee y \vee y) \wedge (\neg x \vee \neg y \vee \neg y)$, form the tripartite graph corresponding to the groupings of the three clauses, connecting nodes whose corresponding literals are logically compatible. If three literals are logically

[44]Kazimierz Kuratowski (1896-1980) was a Polish mathematician.

[45]The Euler characteristic states for any convex polyhedron (any shape in any dimensions that is outward-facing like a circle, triangle, cube, sphere, etc.), or planar graph, $V - E + F = 2$, where V is the number of vertices, E is the number of edges, and F is the number of faces. This equation is regularly voted as one of the most beautiful in mathematics, usually right behind Euler's identity.

[46]A graph minor is, roughly speaking, a graph formed from a subset of the vertices in a graph. Specifically, a minor can be formed from an initial graph by *deleting* some number of vertices and edges, or *contracting* edges, that is merging two vertices such that the edge between them disappears.

compatible, they solve the 3SAT problem, and form a clique. This implies that the clique problem is NP-complete.

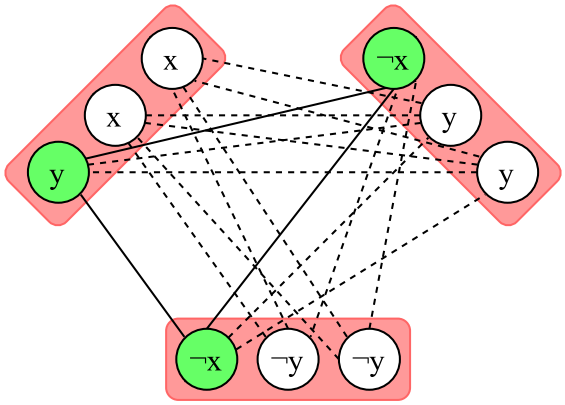


FIGURE 5. Graphically showing the equivalence between 3SAT and the clique problem, implying the clique problem is NP-complete.

21.3.2. *Minimum Vertex Cover.* The goal of the minimum vertex cover problem is to find a subset, V', of vertices of a graph such that every edge in the graph has at least one endpoint in V', and such that the sum of the costs of the chosen vertices is minimised. This can be formulated as an integer linear program,

$$\begin{aligned} &\text{minimise} && z = \textstyle\sum_v c(v)x_v \\ &\text{subject to} && x_u + x_v \geq 1, \forall \{u, v\} \in E, \\ &&& x_i \in \{0, 1\} \end{aligned} \tag{15}$$

where $c(v)$ denotes a cost function for a vertex v and x_v is set to one if the vertex v is selected for the cover.

21.4. **Graph Traversal.** The act of traversing a graph is to visit each of its nodes exactly once, according to some ordering. Two main strategies may be used: depth-first search (DFS) and breadth-first search (BFS). A DFS visits the descendants of a node before visiting its siblings, that is, traversing sub-branch by sub-branch. A BFS visits the siblings of a node before visiting its descendants, that is, traversing depth by depth. Pseudo-code for DFS and BFS is given in Algorithms 2 and 3 respectively. When the graph has a tree structure (only branching, no converging), there is a single path to each node from the root, hence marking nodes as visited is unnecessary. Traversal may, for example, be used to exhaustively locate a particular node in the graph. A BFS has the potential advantage of facilitating a depth cutoff, searching each depth exhaustively and one at a time. However, there is then a space requirement to queue descendants in some internal memory structure, and for a ballooning tree structure such as in a chess *game tree*, storing more than a few levels or *ply* is infeasible.

Algorithm 2 Pseudocode for a depth-first search (DFS).

```
1: procedure InitDFS(graph)
2:     DFS(graph→root)
3: procedure DFS(node)
4:     node→MarkAsRead()
5:     for child in node→children do
6:         if NOT child→AlreadyRead() then
7:             DFS(child)
```

Algorithm 3 Pseudocode for a breadth-first search (BFS).

```
1: procedure BFS(graph)
2:     queue ← [ ] /* initialise FIFO queue */
3:     for child in graph→root→children do
4:         queue→Enqueue(child)
5:     while NOT queue→IsEmpty() do
6:         for child in queue→front→children do
7:             if NOT child→AlreadyRead() then
8:                 queue→Enqueue(child)
9:         queue→Dequeue(queue→front)
```

21.5. **Shortest Path Problem.** A common problem to compute over a weighted graph is the shortest path between a source node and sink node within the graph. Perhaps the two most famous shortest path algorithms are Dijkstra's algorithm and the Bellman-Ford algorithm.

21.5.1. *Dijkstra's Algorithm.* Given a weighted graph, Dijkstra's algorithm, discovered by Dutch computer scientist, Edsger W. Dijkstra (1930-2002), finds the shortest path from a source node to all other nodes in a graph. In practice, the goal is to find the shortest path from a source node to a unique sink node, but Dijkstra's algorithm computes shortest paths to all nodes as a matter of course. The algorithm consists of visiting each node in order of closeness (initially the source node has zero distance and all others are infinite). Each neighbour of the current node is then inspected, and its shortest distances updated, if it so happens that a shorter path can reach it via the current node. After a node has been visited, it is discarded, and the algorithm terminates once all nodes have been visited. Aside from a shortest distance, nodes record the node from which its shortest path derives, such that at the end, the shortest path can be traced back from sink to source. The algorithm runs in $O(N \log N)$ time in the worst case, where N is the number of vertices.

22. Introduction to Machine Learning

Machine learning seeks to construct algorithms that create mathematical models or *machines*, predictive or descriptive, over a set of data. The field utilises an amalgamation

of applied mathematics techniques from probability theory, linear algebra, and optimisation theory. Here we specify a glossary of important terms in machine learning.

22.1. **Learning.** In machine learning, learning refers to optimising a model to generalise over a data set. A machine learning task will follow one of the following fundamental approaches, which depend on the nature of the data $\mathcal{D}$ to be learned from:

22.1.1. *Supervised Learning.* In this case, the dataset $\mathcal{D} = \{\mathbf{X}, \mathbf{y}\}$ is *labelled*, that is, each data sample, $\mathbf{x}_i$, has a corresponding output, y_i, which is real-valued for a regression task, and discrete for classification[47]. The task of supervised learning is therefore to learn the mapping that best approximates $\mathbf{y}$ from $\mathbf{X}$. This is by far the most common form of machine learning, which includes many well known techniques, such as regressions, decision trees, support vector machines, and neural networks.

22.1.2. *Unsupervised Learning.* In this case $\mathcal{D} = \{\mathbf{X}\}$ is unlabelled and the learning task is to extract patterns from the data in and of itself. It is sometimes also known as *knowledge discovery*. For example, we might want to find how to reasonably group the observations in $\mathcal{D}$, a task known as clustering. This is arguably the part of machine learning that is most like human learning in that rules are inferred without external confirmation. Unlike supervised learning, which sports a profusion of techniques, unsupervised learning techniques are less extensive and understood. Unsupervised learning is a sort of holy grail of machine learning.

22.1.3. *Semi-Supervised Learning.* In this case, the dataset is a mixture of both labelled and unlabelled data. In many practical problems, labelled data is hard to come by, but unlabelled data might be abundant. Semi-supervised techniques seek to combine the data in a reliable way. This approach to learning has shown surprisingly limited success so far.

22.1.4. *Self-Supervised Learning.* Self-supervised learning is a branch of unsupervised learning whereby some predictive task is nonetheless constructed, usually for the purposes of *representation learning*. After good representations have been learned, an auxiliary model can be trained on a smaller dataset using the self-supervised representations. Examples of self-supervised tasks are:

- imputation-based: mask some data (e.g. next word in a sentence) and task the model with predicting it.
- contrastive-based: apply class-invariant transformations (e.g. image augmentation) and deploy a loss function maximising representation similarity between original and augmented data.

22.1.5. *Transfer Learning.* Transfer learning refers to a broad set of techniques that use the solution of one problem in solving another. Arguably, supervised machine learning is subsumed by the transfer learning framework, but where the source and target domains are the same, and where training is performed as a single step.

[47]Note the output may further be scalar- or vector-valued, though we will focus on scalar-valued outputs.

22.1.6. *Reinforcement Learning.* Reinforcement learning involves maximising reward in a sequence of actions in a game-like environment.

22.1.7. *Deep Learning.* Deep learning refers to approaches that aim to model a hierarchy of representations in a dataset. One important example are deep convolutional neural networks from computer vision, which "stack" multiple convolutional hidden layers in sequence, each layer modeling finer features of its input images. These models have shown tremendous success in computer vision tasks in recent years, making vast improvements on the earlier state of the art. Another example are deep belief networks, which are restricted Boltzmann machines (RBM)[48] stacked one after the other.

22.1.8. *Discriminative and Generative Learning.* Discriminative models fit a posterior predictive distribution directly. In other words, they learn to directly discriminate between (i.e. classify) data. Generative models rather derive a posterior distribution over data by modeling likelihood and prior and combining them using Bayes' rule. This is often hard to do, especially when the input data is vector-valued, and is a limitation on generative models. In contrast, discriminative models allow for arbitrary basis function expansion and feature engineering. On the other hand, generative models are usually easier to fit, more robust to changes in the dataset (discriminative models must be retrained), and better facilitate semi-supervised learning. Another advantage of generative models is we have or can derive a model for $p(X)$. This means data synthesis is on the table. What are commonly referred to as generative models nowadays are concerned with directly or indirectly modeling the marginal distribution on X.

22.2. **Training and Prediction.** Here we define some key concepts surrounding the training and prediction of models.

22.2.1. *Feature Engineering.* Feature engineering is the exercise of designing features, according to some ontology about the model. This is an opportunity for domain knowledge to be imparted into a model, with the view of making more indicative features. This is an inherently soft science, but constitutes a large part of applied machine learning in practice. *Feature extraction* is a separate concept, which, for example, may use unsupervised learning to derive features for modelling.

22.2.2. *Training.* Training or learning is the act of inferring model parameters from a dataset through some sort of optimisation algorithm. It is vitally important to (randomly) set aside some data from the start, because there must be some independent data available to validate the trained model. If we use all data on training, it is impossible to know how it will perform on unseen data. The usual scheme is to first split the data into training and test datasets. The test data is strictly separated from the training process. A part of the training set is extracted as a validation set. These split sizes follow some loose convention, leaving the majority of data for training, for example a $50 - 25 - 25$ split. Models are

[48]RBMs are a generative neural networks factorising over a bipartite graph (hence, *restricted*). Similarly to autoencoders, they can be used in an unsupervised fashion to learn a feature representation in terms of *latent variables*.

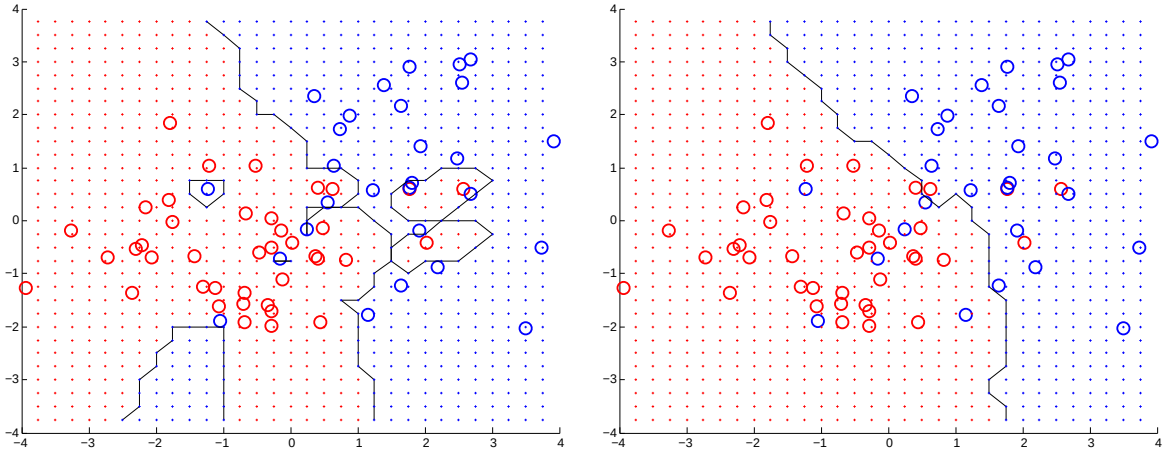
trained on a training set and its predictive performance is assessed on a validation set, providing a benchmark to compare choices of model and model hyper-parameters. This is a process known as *model selection*.

22.2.3. *Cross-Validation.* When data is scarce, we can simulate having a validation set in a procedure called cross-validation (CV). Cross-validation partitions the training data into K validation *folds*. A model is trained on the first $K-1$ folds, and validated on the final fold. This process is repeated K times, such that each fold is validated on, giving K samples for estimating error. If the fold size is chosen to be a single sample (that is, $K = N$), we have leave-one-out cross-validation (LOOCV). The chosen model will be the one that minimises the average CV error. A model is then trained on all the training data (including validation data), and the resulting model tested on the test data. The predictions made are compared with the true values, giving an estimation of generalisation error that can then be reported, in a process known as *model assessment*.

22.2.4. *Generalisation Error.* Generalisation error refers to the expected test error averaged over all possible datasets. As such complete coverage of data is in practice never available, an estimate can be found using cross-validation. As more data is added, the predictive power (generalisation error) of the model improves, as the approximation error (the discrepancy between the estimate and the best estimate given the choice of model) in the model parameters is minimised.

22.2.5. *Overfitting.* Overfitting is a constant danger when training models. This is the effect of *fitting the noise instead of the signal*. All data contains noise, and when the dataset is sufficiently small and the model sufficiently complex (e.g. a high degree polynomial), the model can fit the training data perfectly, only to be useless on test data. Take for example a k-nearest neighbours (kNN) classifier (Figure 6). For small k, outliers have unreasonable influence on local test data, creating discontinuous decision boundaries. There are various strategies to attenuate overfitting. The most ideal is to increase the amount of data available for training, but this is rarely an option. A more typical approach is *regularisation*. This usually takes the form of a penalty term, but really corresponds to imposing a prior distribution on the parameter set. A prior distribution assigns low probabilities to large parameter values. Complex models that overfit usually require large parameter values to make a fit, so regularisation curtails this tendency. Another, more drastic approach, is to change the form of the cost function (likelihood) of the model to insulate it from outliers in the data. An example of this is robust regression, where a Laplace likelihood replaces a Gaussian. A variant of overfitting is the zero count (sparse data) problem, where specific data that ought to be modelled has not appeared in the dataset and leads to a zero probability prediction. *Underfitting* occurs when the complexity of the model is insufficient to capture the trends of the data. This can be addressed by modifying the choice of basis function or kernel, otherwise by choosing a different technique.

22.2.6. *Ground Truth.* Ground truth is a somewhat nebulous term usually referring to a supervised data set, in particular, that which can be assumed to be correct, and which is to be assimilated by a model. It differs from those patterns that are inferred, for example



(A) $k = 1$ (B) $k = 7$

FIGURE 6. Visualisation of overfitting with k-nearest neighbours (kNN) (non-parametric model). A data point is classified as the majority class of the k geometrically closest data points. Note the steadier decision boundary formed for $k = 7$. This classifier can be very effective, but suffers from the curse of dimensionality. Data normalisation and dimensionality reduction usually help. However, in certain high-dimensional data sets where data is confined to a lower-dimensional manifold, for example in OCR data, kNN can be very effective. Unlike most classifiers, test time is much greater than training time. Approximations using memory tradeoffs mitigate this. Created with `knnDemo.m`.

in an unsupervised setting. The term comes from the field of earth sciences where it refers to data measured directly on the ground, or in the atmosphere, rather than indirect data samples collected from, for example, satellite imagery. The *gold standard* is a related term from statistics that refers to the convention for the best possible statistical test one can perform. What qualifies as the gold standard varies according to the context. In practice, it is shorthand for indicating that a statistical study conforms to the highest possible standard.

22.2.7. *The Curse of Dimensionality.* The curse of dimensionality refers to the difficulty of generalising over a small amount of high-dimensional data. This is because the observation state space grows exponentially as features are added. We therefore need to increase our data exponentially to maintain good coverage. Automatic techniques such as principal components analysis (PCA) exist for *dimensionality reduction* of data, in which the data is mapped to a lower-dimensional space representing the axes of highest variance (trends) of the data. Another option is to encourage model sparsity using l_1 regularisation. This tends to build models that ignore unimportant features. Various feature selection techniques also

exist, where an optimised subset of features is selected according to preprocessing (filtering), cross-validation (wrapping), or as part of the learning algorithm itself (embedding).

22.2.8. *Bias-Variance Decomposition.* The bias-variance tradeoff refers to a result illustrating the expected behaviour of a parameter estimate, with respect to the 'true' parameters it approximates. Bias represents *underfitting*, and variance represents *overfitting*. Underfitting/bias is observed in training error; overfitting/variance in test error.

Let $Y = f(x) + \epsilon$ be the ground truth value with $f(x)$ the true signal for a single test sample x, and noise, $\epsilon = \mathcal{N}(0, \sigma^2)$. Let our model estimate be denoted by $\hat{f}(x; \mathcal{D})$ for sample training data $\mathcal{D} = \{(x_1, y_2), \dots (x_n, y_n)\}$. Then, the test error under a MSE for all possible training samples (entailing all possible trained models for the class $\hat{f}$), measured over all possible test samples is,

$$\begin{aligned}
\mathbb{E}_x\left[\mathbb{E}_{\mathcal{D},\epsilon}\left[\big(Y - \hat{f}(x)\big)^2\right]\right] &= \mathbb{E}_x\left[\mathbb{E}_{\mathcal{D},\epsilon}\left[\Big(f(x) + \epsilon - \mathbb{E}\big[\hat{f}(x)\big] + \mathbb{E}\big[\hat{f}(x)\big] - \hat{f}(x)\Big)^2\right]\right] \\
&= \mathbb{E}_x\left[\sigma^2 + \Big(f(x) - \mathbb{E}_{\mathcal{D},\epsilon}\big[\hat{f}(x)\big]\Big)^2 + \mathbb{E}_{\mathcal{D},\epsilon}\left[\Big(\hat{f}(x) - \mathbb{E}_{\mathcal{D},\epsilon}\big[\hat{f}(x)\big]\Big)^2\right]\right] \\
&= \mathbb{E}_x[\text{noise} + \text{bias}^2 + \text{variance}],
\end{aligned}$$

where the first step introduces the terms $\pm\mathbb{E}\big[\hat{f}(x)\big]$ so as to complete the decomposition, and the second step relies on some tedious algebra. The decomposition shows therefore that generalisation error depends both on the variance of the trained model around its own mean (which increases with model complexity i.e. overfitting), and the bias, the gap between the mean of the estimator and the ground truth (which increases with model parsimony i.e. underfitting). Should the model class be incorrectly chosen, the resulting bias is known as 'structural error'. Even if the model class is correct, there is an inescapable error of noise that all models share called the *noise floor*, which, given all terms are positive, the represents an irreducible lower bound on generalisation error. The noise floor is the limit of the learning curve (the trend of improvement in generalisation error as data is added) for an unbiased estimate. An unbiased estimate is a model that converges (in probability) to a hypothetical true parameter set as data increases. A classic case of bias-variance tradeoff is seen between ridge regression and ordinary least squares.

22.2.9. *Gauss-Markov Theorem.* Orthogonal to the bias-variance tradeoff is the Gauss-Markov theorem. This shows that among unbiased linear models, the one with lowest variance is ordinary least squares. This is thus a statement on the choice of loss function, in contrast to bias-variance, which is a statement about regularisation. In effect, it is a fundamental result, though somewhat tautological: "the best way to minimise variance is to infer parameters that minimise mean square error, that is, variance".

22.2.10. *No free-lunch theorems.* No free lunch in machine learning refers to the fact that, averaged over all possible ground truths, no model is better than random guessing. This

illustrates the power of the implicit assumptions we make when modelling data, for example, that the ground truth function is smooth with Gaussian noise. If the data is uniformly random, there will be no signal to learn.

23. Bayes and the Beta-Binomial Model

We begin by examining the primordial Beta-Binomial model–that which was studied by English mathematician, Thomas Bayes (1701-1761), in *An Essay towards solving a Problem in the Doctrine of Chances*[49]. Here we consider inference over data produced from a sequence of coin tosses, such as,

$$0101001010111,$$

where 0 represents 'tails' and 1 'heads'. In fact, it is sufficient to know the number of heads and tails, which we denote N_1 and N_0 respectively. These are known as 'sufficient statistics'. Note that the beta-binomial model is not a classifier, as we are not engaging in *classification*, but rather we are predicting a single next outcome. The starting point is to define the likelihood. The probability distribution for such a random sequence is the discrete binomial distribution. As we are maximising for the rate parameter, we can drop the normalisation constant and we have,

$$p(\mathcal{D}|\theta) = \theta^{N_1} \cdot (1-\theta)^{N_0}.$$

Clearly, we can maximise this expression by taking the derivative to give,

$$N_1\theta^{N_1-1}(1-\theta)^{N_0} - N_0\theta^{N_1}(1-\theta)^{N_0-1} = 0$$
$$\implies \hat{\theta}_{MLE} = \frac{N_1}{N_0+N_1} = \frac{N_1}{N},$$

where $N = N_0 + N_1$. The prior distribution is chosen to be *conjugate* to the likelihood, that is, of like form. The continuous beta distribution is therefore a suitable choice, hence,

$$p(\theta) = \text{Beta}(\theta|a,b) \propto \theta^{a-1}(1-\theta)^{b-1},$$

where we drop the normalising beta function for the proportionality. The values of a and b are hyper-parameters that control the skew. When $a = b = 1$, we have a uniform prior and the posterior is just the same as the likelihood. We multiply the prior and likelihood to obtain the posterior,

$$p(\theta|\mathcal{D}) \propto \theta^{N_1+a} \cdot (1-\theta)^{N_0+b}.$$

Notice how the hyper-parameters behave as additional observations. For this reason they are known as *pseudo-counts*–it is as though we had some additional data other than $\mathcal{D}$ that we can use to influence or stabilise the behaviour of the posterior. This is the power of a prior distribution. Now, the beta distribution distributes the continuous θ, so this time we can simply look up the mode of a beta distribution to obtain the MAP estimate,

[49]This early term for probability theory comes from de Moivre's seminal textbook (1718).

$$\hat{\theta}_{MAP} = \frac{N_1 + a - 1}{N + a + b - 2}.$$

Notice that as $N \to \infty$, the MAP estimate approaches the ML estimate. This shows that a prior serves its role of stabiliser only for smaller data sets. As data is added, it is overwhelmed by the volume of data fitted by the likelihood. Now, the posterior predictive distribution for unseen data, $\hat{x}$, is either given directly by the ML or MAP estimates, or else from Bayes model averaging,

$$\begin{aligned} p(\hat{x} = 1|\mathcal{D}) &= \int_0^1 p(x = 1|\theta) p(\theta|\mathcal{D})\, dx \\ &= \int_0^1 \theta p(\theta|\mathcal{D})\, dx = \mathbb{E}[\theta|\mathcal{D}] = \frac{N_0 + a}{N + a + b}. \end{aligned}$$

The mean of a beta distribution differs from its mode, as it is asymmetric.

24. Generative Classifiers

Here we describe *generative classifiers*, by illustrating how their components are derived with the naive Bayes classifier (NBC). A classifier is simply a model that predicts the membership of an observation, x, in one of a discrete number of classes. Generative classifiers 'generate' the posterior distribution through inference, in contrast to *discriminative* classifiers, which model the posterior distribution directly. The parameters of a NBC are found by considering the likelihood for N data training samples, $\mathcal{D}$, of dimension D, which may be written,

$$\begin{aligned} \textbf{likelihood} = p(\mathcal{D}|\theta) = p(\mathbf{X}, \mathbf{y}|\boldsymbol{\theta}) &= p(\mathbf{y}|\boldsymbol{\theta}) p(\mathbf{X}|\mathbf{y}, \boldsymbol{\theta}) \\ &= \textbf{class prior} \times \textbf{class conditional density} \\ &= \prod_{i=1}^{N} p(y_i|\boldsymbol{\pi}) p(\mathbf{x}_i|y_i, \boldsymbol{\theta}), \qquad (16) \\ &= \prod_{i=1}^{N} p(y_i|\boldsymbol{\pi}) \prod_{j=1}^{D} p(x_{ij}|y_i, \boldsymbol{\theta}_j), \qquad (17) \end{aligned}$$

where we denote the set of parameters for both input and output variables, $\boldsymbol{\theta} = \{\boldsymbol{\pi}, \boldsymbol{\theta}\}$. Step (16) follows from the data being independent and identically distributed (a standard assumption), and step (17) from the naive Bayes assumption of feature independence when the class is given[50] We take logarithms to derive the log-likelihood, a common trick that makes optimisation easier without changing the optimal parameters,

[50]This is called a naive assumption because it precludes covariance between model features, which in practice is usually present.

$$\begin{aligned}\log p(\mathcal{D}|\boldsymbol{\theta}) &= \sum_{i=1}^{N} \log p(y_i|\boldsymbol{\pi}) + \sum_{i=1}^{N}\sum_{j=1}^{D} \log p(x_{ij}|y, \boldsymbol{\theta}_j) \\ &= \sum_{c} N_c \log \boldsymbol{\pi}_c + \sum_{c}\sum_{i:y_i=c}\sum_{j=1}^{D} \log p(x_{ij}|y=c, \boldsymbol{\theta}_{jc}),\end{aligned}$$

where N_c is the number of observations having class c. Now we have an expression that is easy to optimise, we can obtain the **maximum likelihood estimate** (MLE) by optimising each of the parts of the sum. This is equivalent to calculating the *mode* of each of the probability functions. For the multinomial class variable, the task is simple, $\hat{\pi}_c = N_c/N$. For the input parameters, the MLE depends upon the choice of distribution. One of the benefits of the naive independence assumption is it allows us to model any combination of distributions with ease–all we need do is substitute their probability functions. For illustration, we may simply opt for Bernoulli distributions on each of the features. Hence, the mean for feature j given class c is $\hat{\theta}_{jc} = N_{jc}/N_c$, where $N_{jc} = \sum_{i:y_i=c} x_{ij}$, noting the x_{ij} are binary. The maximum likelihood estimate is therefore, $\hat{\boldsymbol{\theta}}_{MLE} = \{\hat{\pi}_c, \hat{\theta}_{jc} : 1 \leq c \leq C, 1 \leq j \leq D\}$. This is now enough to make a prediction. The posterior predictive distribution derives from,

$$\begin{aligned}\textbf{posterior predictive} = p(y=c|\mathbf{x}, \boldsymbol{\theta}) &= \frac{p(y=c|\boldsymbol{\theta})p(\mathbf{x}|y=c, \boldsymbol{\theta})}{\sum_{c'} p(y=c'|\boldsymbol{\theta})p(\mathbf{x}|y=c', \boldsymbol{\theta})} \\ &\propto p(y=c|\boldsymbol{\theta})p(\mathbf{x}|y=c, \boldsymbol{\theta}),\end{aligned} \tag{18}$$

having the same form as (17). For a test observation, $\hat{\mathbf{x}}$, we can simply cycle through each of the classes, plug the appropriate parameters into (18), and compute a probability. We can then predict the classification of $\mathbf{x}$ to be the class c having maximum probability. This is called a **plug-in approximation**, and it is the first of two alternatives for prediction. We will come to the other shortly, but first there are several more things we can do with plug-in approximation. To begin with, we may wish to include *prior* information on the parameters. Thus, we form the prior distribution,

$$\textbf{prior} = p(\boldsymbol{\theta}) = p(\boldsymbol{\pi}) \prod_{c}\prod_{j=1}^{D} p(\theta_{jc}).$$

The prior imposes a distribution on the parameter set, which has a stabilising effect when the training data is limited, mitigating overfitting. We choose the prior distributions on the parameters to be *conjugate* to the likelihood distributions, that is, having a similar form. This is by no means compulsory, but it makes the derivations easier. Since we have a multinomial distribution on y and Bernoulli distributions on $x|y = c$, we choose Dirichlet and beta distributions respectively for the prior parameters. Thus, $\boldsymbol{\pi} \sim \text{Dir}(\boldsymbol{\alpha})$

and $\boldsymbol{\theta}_j \sim \text{Beta}(\beta_0, \beta_1)$. Choosing $\boldsymbol{\alpha} = \mathbf{1}$ and $\boldsymbol{\beta} = \mathbf{1}$ corresponds to a common practice known as *add one* smoothing.

If we combine the prior with the likelihood, we get the posterior distribution, which strikes a balance between what the data tells us and what we expect in advance,

$$\begin{aligned}\textbf{posterior} = p(\boldsymbol{\theta}|\mathcal{D}) &\propto \textbf{likelihood} \times \textbf{prior} \\ &= \prod_{n=1}^{N} \text{Cat}(y_i|\boldsymbol{\pi})\text{Dir}(\boldsymbol{\pi};\boldsymbol{\alpha}) \prod_{c} \prod_{i:y_i=c} \prod_{j=1}^{D} \text{Ber}(x_{ij}|\theta_{jc})\text{Beta}(\theta_{jc};\beta_0,\beta_1), \\ &= p(\boldsymbol{\pi}|\mathcal{D}) \prod_{j=1}^{D} p(\boldsymbol{\theta}_j|\mathcal{D}),\end{aligned}$$

where $p(\boldsymbol{\pi}|\mathcal{D}) = \text{Dir}(N_1 + \alpha_1, \ldots, N_C + \alpha_C)$ is the posterior on $\boldsymbol{\pi}$, and $p(\boldsymbol{\theta}_j|\mathcal{D}) = \text{Beta}((N_c - N_{jc}) + \beta_0, N_{jc} + \beta_1)$ is the posterior on $\boldsymbol{\theta}_j$. Note that because we are optimising for $\boldsymbol{\theta}$, it does not matter that the expression is only proportional to the posterior. Also note that if we have a uniform prior, the probabilities are constant, and the posterior is equivalent to the likelihood. Now we have two options: firstly, we can compute the mode of this posterior distribution giving us the **maximum a posteriori estimate**, $\hat{\boldsymbol{\theta}}_{MAP}$. This can then be plugged into the posterior predictive, just as with the MLE. The alternative is to use **Bayes model averaging**. This calculates the posterior as a weighted average of the distribution. In this scheme, neither the ML or MAP estimates are computed directly, rather, we sum or integrate over the unknown parameters, marginalising them,

$$\begin{aligned}p(y = c|\hat{\mathbf{x}}, \mathcal{D}) &= \frac{p(\hat{\mathbf{x}}|y = c, \mathcal{D})p(y = c|\mathcal{D})}{\sum_{c'} p(\hat{\mathbf{x}}|y = c', \mathcal{D})p(y = c'|\mathcal{D})} \\ &\propto p(\hat{\mathbf{x}}|y = c, \mathcal{D})p(y = c|\mathcal{D}) \\ &= \int p(\hat{\mathbf{x}}, \boldsymbol{\theta}|y = c, \mathcal{D})\, d\boldsymbol{\theta} \int p(y = c, \boldsymbol{\pi}|\mathcal{D})\, d\boldsymbol{\pi} \qquad (19) \\ &= \int p(\hat{\mathbf{x}}|y = c, \boldsymbol{\theta}, \cancel{\mathcal{D}})p(\boldsymbol{\theta}|\cancel{y = c,}\mathcal{D})\, d\boldsymbol{\theta} \int p(y = c|\boldsymbol{\pi}, \cancel{\mathcal{D}})p(\boldsymbol{\pi}|\cancel{y = c,}\mathcal{D})\, d\boldsymbol{\pi}, \qquad (20)\end{aligned}$$

where step (19) comes form the law of total probability and the cancellations in step (20) occur because the prediction is independent of the data when conditioned on the parameters. Thus we are averaging by the posterior distribution. This may be solved analytically for the choice of distributions we have made. Now, a retrospective connection to plug-in approximation may be seen. Plug-in approximation works on the assumption that the distribution on the parameters, $p(\theta|\mathcal{D}) \to \delta_{\hat{\theta}_{MAP}}(\theta)$ as $|\mathcal{D}| \to \infty$. This follows the intuition that increasing data increases our certainty about the parameters. If we replace the posteriors in the integral with the limit, then by the sifting property of the Dirac delta function, we obtain the class conditional and class prior distributions, conditioned on the

MAP estimate–the plug-in approximation. A simple implementation of the naive Bayes classifier is given in `nbcDemo.m`.

25. Multivariate Normal Distribution

The multivariate Normal (MVN) or multivariate Gaussian distribution is the multidimensional generalisation of the univariate Gaussian distribution. The form of the probability density function for a D-dimensional Gaussian is defined to be,

$$\mathcal{N}(\mathbf{x};\mu,\Sigma) \triangleq \frac{1}{(2\pi)^{D/2}|\mathbf{\Sigma}|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\mathbf{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right]$$

where the vector $\boldsymbol{\mu}$ is the vector of means and $\mathbf{\Sigma}$ is the covariance matrix. The exponent expresses the Mahalanobis distance (named after Indian statistician Prasanta Chandra Mahalanobis (1893-1972)), which captures the behaviour of mean squared error in D dimensions. The covariance matrix is by definition symmetric positive semi-definite, and may be diagonalised with a diagonal matrix of eigenvalues, $\mathbf{\Lambda}$, and orthonormal eigenvectors, $\mathbf{U}$, as $\mathbf{\Sigma} = \mathbf{U}\Lambda\mathbf{U}^T$ where orthonormality implies $\mathbf{U}^{-1} = \mathbf{U}^T$. Consequently,

$$\mathbf{\Sigma}^{-1} = \mathbf{U}^T\Lambda^{-1}\mathbf{U} = \sum_{d=1}^{D}\frac{1}{\lambda_d}\mathbf{u}_d\mathbf{u}_d^T,$$

where $\mathbf{u}_d$ are the columns of $\mathbf{U}$. Therefore, the Mahalanobis distance can be expressed as,

$$(\mathbf{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu}) = \sum_{d=1}^{D}\frac{y_d^2}{\lambda_d^2},$$

where $y_i = \mathbf{u}_i^T(\mathbf{x}-\boldsymbol{\mu})$, from which it may be seen that events of equal probability lie along elliptical contours, for which the eigenvectors form the axes and eigenvalues control the distortion. Fitting a MVN from data $\mathcal{D}$ of size N can be done with maximum log-likelihood estimation. With a bit of vector calculus, the MLE is,

$$\hat{\boldsymbol{\mu}}_{MLE} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n,$$

that is, the sample mean and,

$$\hat{\mathbf{\Sigma}}_{MLE} = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_i-\boldsymbol{\mu})(\mathbf{x}_i-\boldsymbol{\mu})^T,$$

the sample covariance matrix.

25.1. **Gaussian Discriminant Analysis.** Gaussian discriminant analysis is a family of classifiers that fit Gaussian distributions to each of K classes. That is, the class-conditional densities are defined to be,

$$p(\mathbf{x}|y=c) = \mathcal{N}(\mathbf{x}; \hat{\mu}_c, \hat{\Sigma}_c),$$

where the parameters come from the maximum likelihood for multivariate normal distributions (MVN). Where the covariance matrix is diagonal, the features are independent, and this is equivalent to the naive Bayes classifier. This sort of modelling, where we learn K Gaussian densities and predict membership of a test observation is effectively a generalisation of a nearest centroids classifier[51].

25.1.1. *Quadratic Discriminant Analysis.* The posterior predictive distribution can then be defined as,

$$p(y=c|\mathbf{x},\theta) = \frac{\pi_c(2\pi)^{-D/2}|\mathbf{\Sigma}|^{-1/2}\exp\left[-\frac{1}{2}(\mathbf{x}-\mu_c)^T\mathbf{\Sigma}_c^{-1}(\mathbf{x}-\boldsymbol{\mu}_c)\right]}{\sum_{c'}\pi_{c'}(2\pi)^{-D/2}|\mathbf{\Sigma}|^{-1/2}\exp\left[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_{c'})^T\mathbf{\Sigma}_{c'}^{-1}(\mathbf{x}-\boldsymbol{\mu}_{c'})\right]},$$

where π_c is the class prior. It may be shown (though we will not do so here) that a decision rule[52] creates a quadratic boundary between the centroids in Euclidean space. Notice finally that we are effectively doing a plug-in approximation, in the absence of Bayes model averaging.

25.1.2. *Linear Discriminant Analysis.* Linear discriminant analysis (LDA) arises from QDA when the simplifying assumption is made that the covariance matrices are all equal. In this instance we have,

$$p(y=c|\mathbf{x},\theta) \propto \exp\left[\boldsymbol{\mu}_c^T\Sigma^{-1}\mathbf{x} - \frac{1}{2}\boldsymbol{\mu}_c^T\mathbf{\Sigma}^{-1}\boldsymbol{\mu}_c + \log\pi_c\right]\exp\left[-\frac{1}{2}\mathbf{x}^T\mathbf{\Sigma}^{-1}\mathbf{x}\right],$$

and the quadratic term cancels out over the sum, leaving an expression that is linear in $\mathbf{x}$. It can be shown easily that this produces linear decision boundaries. There are some interesting connections between LDA and other parts of machine learning. If we define $\gamma_c = -\frac{1}{2}\mu_c^T\Sigma^{-1}\mu_c + \log\pi_c$ and $\beta_c = \Sigma^{-1}\mu_c$, we can write,

$$p(y=c|\mathbf{x},\theta) = \frac{e^{\beta_c^T\mathbf{x}+\gamma_c}}{\sum_{c'}e^{\beta_{c'}^T\mathbf{x}+\gamma_{c'}}} = \mathcal{S}(\eta)_c,$$

[51] A nearest centroids classifier is one of the simplest classifiers imaginable. Training is performed by computing centroids for points in each class of training data, $\boldsymbol{\mu}_c = 1/|c|\sum_{i:y_i=c}\mathbf{x}_i$. A prediction for unseen data $\hat{\mathbf{x}}$ is then made as $\hat{y} = \min_c||\hat{\mathbf{x}} - \boldsymbol{\mu}_c||$, that is, the class of the nearest centroid.

[52] A decision rule is a probability threshold for classification, usually equal to 0.5 in binary classification.

where $\eta_c = [\beta_1^T \mathbf{x} + \gamma_1, \beta_2^T \mathbf{x}\gamma_2, \ldots, \beta_C^T \mathbf{x} + \gamma_C]$, and $\mathcal{S}$ is the *softmax* function[53]. The marginal likelihood is known as the partition function, denoted Z for 'Zustandssumme'– the German expression for 'sum over states'. This form of LDA is very similar to logistic regression, differing only in the fact that LDA is generative and logistic regression is discriminative. If we normalise a naive Bayes classifier with Gaussian features, it is equivalent to LDA–hence naive Bayes and logistic regression form what is known as a generative-discriminative pair.

26. Linear Regression

A linear regression is used to make predictions for a continuous variable.

26.1. **Constructing a Loss Function.** The starting point for building a regression model is a dataset, $\mathcal{D}$, where,

$$\mathcal{D} = \{(\mathbf{x}_n, y_n)\}_{n=1}^N,$$

that is, a set of N data points, where $\mathbf{x}_i \in \mathbb{R}^D$ is a D-dimensional *input* vector, and $y_i \in \mathbb{R}$ are the corresponding scalar outputs. The dataset, $\mathcal{D}$, is known as our *training set*. The objective of the regression algorithm is to infer the linear function that best fits the data. More precisely, we want to find the vector of coefficients, $\boldsymbol{\beta} = [\beta_1, \beta_2, \ldots, \beta_D]$, that gives the best approximation for,

$$\begin{aligned} y_n &\approx \beta_1 x_{n1} + \beta_2 x_{n2} + \cdots + \beta_D x_{nD} \\ &= \mathbf{x}_n^T \boldsymbol{\beta}, \end{aligned}$$

for all $n = 1, \ldots, N$. Each of the D dimensions of $\mathbf{x_n}$ is known as an indicator. The greater the corresponding coefficient in the parameter vector, $\boldsymbol{\beta}$, the more weight this dimension has in determining the output, y_n. We can formulate the above as an optimisation problem,

$$\min_{\boldsymbol{\beta}} \mathcal{L}(\boldsymbol{\beta}) = \frac{1}{2N}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) = \frac{1}{2N}\sum_{n=1}^{N}(y_n - \mathbf{x_n}^T\boldsymbol{\beta})^2,$$

where $\mathcal{L}(\boldsymbol{\beta})$ is the 'loss' function, expressing a *mean square error* (MSE) for choosing parameter vector $\boldsymbol{\beta}$, that is, the average[54] squared distance of the approximation from the true value. Note the similarity to the variance statistic. It is the job of the learning algorithm to minimise this function. It is also typical to write MSE = RSS/N, where RSS stands for *residual sum of squares*, the non-weighted sum. Note we could have chosen any

[53]The softmax function is so called because at low 'temperatures', that is, if we divide all the exponents by T, the probability of the most likely class goes to 1 as $T \to 0$. This terminology comes from statistical physics–in fact the softmax function has the same form as the Boltzmann distribution (Ludwig Boltzmann (1844-1906) was a German physicist credited with the development of statistical mechanics).

[54]This is empirical risk minimisation: minimising the expected loss. It is a fundamental principle in statistical learning.

sort of loss function, but our reasons for choosing mean squared error will be revealed in the following section.

26.2. **Method of Least Squares.** The method of least squares (MLS) is a closed-form solution to a linear regression. As previously stated, it was first published in the same paper by Gauss that also first formalised the Normal distribution. We will discover the link in the following section. Using the chain rule, we may write the derivative of the loss function,

$$\frac{\delta \mathcal{L}}{\delta \boldsymbol{\beta}} = -\frac{1}{N} X^T (\mathbf{y} - X\boldsymbol{\beta}),$$

where $\mathbf{X} \in \mathbb{R}^{N \times D}$ is the matrix whose ith row is the ith D-dimensional vector, $\mathbf{x}_i$, and $\mathbf{y} \in \mathbb{R}^D$ is the vector whose ith element is the ith scalar, y_i. We can therefore minimise $\mathcal{L}(\boldsymbol{\beta})$ by setting its derivative to 0, giving the normal equation,

$$\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) = 0,$$

and finally, the optimal parameter vector,

$$\boldsymbol{\beta}^* = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}.$$

MLS also has a nice geometric interpretation. According to the normal equation, the error, $\mathbf{X}\boldsymbol{\beta} - \mathbf{y}$, is minimised when it is perpendicular to the hyperplane defined by $\mathbf{X}^T$. That is,

$$\mathbf{X}^T\mathbf{e} = \mathbf{0}$$
$$\implies \mathbf{e}^T\mathbf{X}\mathbf{a} = 0, \forall \mathbf{a},$$

where $\mathbf{e} = \mathbf{X}\boldsymbol{\beta} - \mathbf{y}$ is the error vector. Thus, the optimal parameters occur when the error is orthogonal to all vectors in the span of $\mathbf{X}$, that is, when it is normal to the plane. This makes intuitive sense, because it means the choice of prediction, $\mathbf{X}\boldsymbol{\beta}$, that minimises the distance to the true value, y, is the one directly 'below' it on the hyperplane, $\mathbf{X}^T$. This solution is known as the ordinary least squares (OLS) solution.

26.3. **The Gram Matrix.** For the closed-form method of least squares, we require the *gram* matrix $\mathbf{X}^T\mathbf{X}$ to be invertible. If $\mathbf{X}$ is full rank then $\mathbf{X}\mathbf{a} = \mathbf{0}$ iff $\mathbf{a} = \mathbf{0}$[55]. Therefore, $\mathbf{a}^T\mathbf{X}^T\mathbf{X}\mathbf{a} \geq 0$, implying the gram matrix is positive-definite, and therefore invertible. Note that when $N < D$ and we have a 'fat' $\mathbf{X}$, $\mathbf{X}$ is often rank deficient. Even when the gram matrix is invertible, it may still be ill-conditioned. This arises when a high degree of *multicolinearity* is present in the data. This refers to high degrees of correlation between features. Ill-conditioning is a concept common in numerical analysis, and in linear algebra it is captured by the *condition number* of a matrix. This provides an upper bound on the relative error between solutions for changes in the data. A high condition number

[55]A matrix is full rank if its columns are linearly independent. By the rank-nullity theorem, full rank implies a zero-dimensional null space or kernel.

indicates parameters may vary significantly for small changes in the data. This can lead to problems in practice, when floating point arithmetic introduces small inaccuracies. It can be shown that the condition number, $\kappa(\mathbf{A}) = ||\mathbf{A}|| \cdot ||\mathbf{A}^{-1}|| = \boldsymbol{\sigma}_{max}/\boldsymbol{\sigma}_{min}$, that is, the ratio of the largest and smallest eigenvalues of $\mathbf{A}$. Geometrically, this makes sense. Consider the way a matrix transforms a unit circle of vectors in two dimensions. When one eigenvalue dominates the other, vectors are mostly 'stretched' in the direction of the first eigenvector. We would find that a small change in the direction of the second eigenvector corresponds to a much larger change in the direction of the first. 'Lifting' the eigenvalues (thereby reducing the condition number) is a convenient outcome of ridge regression. This is easily seen by considering the *orthonormal*[56] eigendecomposition of the gram matrix, then $\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{U}^T + \lambda\mathbf{I} = \mathbf{U}(\boldsymbol{\Sigma} + \lambda\mathbf{I})\mathbf{U}^T$. It can now be seen that a ridge regression improving its condition number through regularisation is the means by which it reduces variance in the bias-variance tradeoff. The equivalence between penalty term, prior distribution, and numerical stabiliser can also be seen. Note that ridge regression is a special case of Tikhonov regularisation for ill-posed problems: a ridge regression imposes a spherical Gaussian prior on the model parameters. Tikhonov regularisation permits any covariance matrix, however.

26.4. **Gradient Descent.** As an alternative to closed-form solutions, there are always sure-fire numerical methods. The most fundamental of these is the method of gradient descent. The convenient form of our loss function makes it a differentiable function that is furthermore *convex*. It therefore has a unique global minimum. We can approach this optimal point iteratively from an arbitrary starting point by taking steps in the direction of the gradient (Figure 7). The algorithm is defined as,

$$\boldsymbol{\beta}^{k+1} = \boldsymbol{\beta}^k - \alpha\nabla\mathcal{L}(\boldsymbol{\beta}^k),$$

where α is the step size parameter and $\nabla\mathcal{L}(\boldsymbol{\beta}^k) = \frac{\delta\mathcal{L}(\boldsymbol{\beta}_k)}{\delta\boldsymbol{\beta}}$. Gradient descent can be enhanced with a line search to find the optimal step size for each search direction. When used, the algorithm exhibits a zig-zag path to the optimal point, as each step is perpendicular to the last. Gradient descent also has more sophisticated variants such as Newton's method and quasi-Newton methods, whose application may be warranted for larger data sets. One alternative to gradient descent is the conjugate gradient method, an iterative technique related to the direct method of Cholesky decomposition for symmetric positive-definite matrices, in which the solution vector is built up iteratively as a linear combination of a set of mutually conjugate vectors. This approach may be advantageous in sparse linear systems.

26.4.1. *Proof of Convergence.* Firstly, suppose we have a convex function f whose gradient f' is Lipschitz continuous with constant M. As a result, the second derivative is bounded by M. Consider the Taylor expansion of f at a point y, in one dimension for brevity:

[56]Note it is always possible to make an eigendecomposition orthonormal by dividing the eigenvectors by their norms and scaling the eigenvalues accordingly.

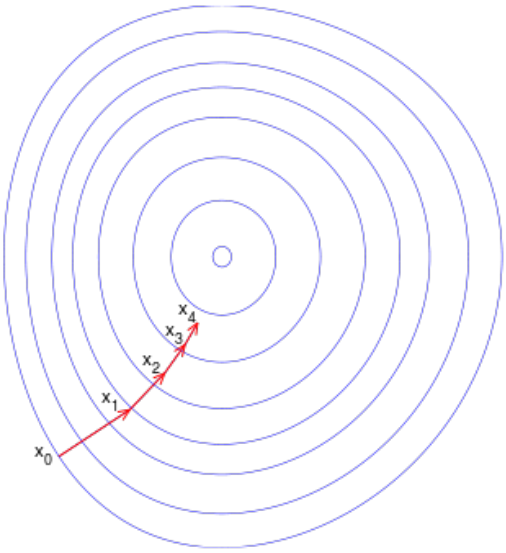


FIGURE 7. The method of gradient descent iteratively approaches a global minimum on a convex function (https://commons.wikimedia.org/wiki/File:Gradient_descent.svg).

$$f(y) = f(x) + f'(x)(y-x) + R_1(y)$$

where

$$R_1(y) = \int_x^y f''(t)(y-t)dt \leq \int_x^y M(y-t)dt = \frac{M}{2}(y-x)^2$$

Note that in higher dimensions, M is chosen such that $M\mathbf{I} \succeq \nabla^2 f(\mathbf{x})$, that is, $M\mathbf{I} - \nabla^2 f(\mathbf{x})$ is negative semi-definite[57]. Thus, in general we have,

$$f(\mathbf{y}) \leq f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y}-\mathbf{x}) + \frac{M}{2}||\mathbf{y}-\mathbf{x}||_2^2$$

Consider a gradient step: $\mathbf{x}^+ = \mathbf{x} - t\nabla f(\mathbf{x})$. By substitution, we have,

[57]The symbol $\succ$ reads "succeeds", and refers to order. In linear algebra, $\mathbf{A} > 0$ means that $\mathbf{A}$ is positive-definite

$$\begin{aligned} f(\mathbf{x}^+) &\leq f(\mathbf{x}) + f'(\mathbf{x})(\mathbf{x}^+ - \mathbf{x}) + \frac{M}{2}||\mathbf{x}^+ - \mathbf{x}||_2^2 \\ &= f(\mathbf{x}) - t\nabla f(\mathbf{x})^T \nabla f(\mathbf{x}) + \frac{Mt^2}{2} f(\mathbf{x})^T \nabla f(\mathbf{x}) \\ &= f(\mathbf{x}) - t \cdot \left(1 - \frac{Mt}{2}\right) \cdot ||\nabla f(\mathbf{x})||_2^2 \end{aligned}$$

Note that if we choose $0 < t \leq 1/L$, we can guarantee convergence since then,

$$f(\mathbf{x}^+) \leq f(\mathbf{x}) - \frac{t}{2}||\nabla f(\mathbf{x})||_2^2$$

Next consider the optimal point x^*. Since f is convex, we can establish the relation,

$$f(\mathbf{x}) \leq f(\mathbf{x}^*) + \nabla f(\mathbf{x})^T(\mathbf{x}^* - \mathbf{x})$$

and the objective function decreases after each iteration. Combining these inequalities gives,

$$f(\mathbf{x}^+) \leq f(\mathbf{x}^*) + \nabla f(\mathbf{x})^T(\mathbf{x}^* - \mathbf{x}) - \frac{t}{2}||\nabla f(\mathbf{x})||_2^2$$

Then, we have,

$$\begin{aligned} f(\mathbf{x}^+) - f(\mathbf{x}^*) &\leq \frac{1}{2t}\big(2t\nabla f(\mathbf{x})^T(\mathbf{x} - \mathbf{x}^*) - t^2||\nabla f(\mathbf{x})||_2^2\big) \\ &= \frac{1}{2t}\Big(2t\nabla f(\mathbf{x})^T(\mathbf{x} - \mathbf{x}^*) - t^2||\nabla f(\mathbf{x})||_2^2 - ||\mathbf{x} - \mathbf{x}^*||_2^2 + ||\mathbf{x} - \mathbf{x}^*||_2^2\Big) \\ &= \frac{1}{2t}\Big(||\mathbf{x} - \mathbf{x}^*||_2^2 - ||\mathbf{x} - t\nabla f(\mathbf{x})^T - \mathbf{x}^*||_2^2\Big) \\ &= \frac{1}{2t}\Big(||\mathbf{x} - \mathbf{x}^*||_2^2 - ||\mathbf{x}^+ - \mathbf{x}^*||_2^2\Big) \end{aligned}$$

Now, summing to iteration k gives us,

$$\begin{aligned} \sum_{i=1}^{k} f(\mathbf{x}^{(i)}) - f(\mathbf{x}^*) &\leq \sum_{i=1}^{k} \frac{1}{2t}\Big(||\mathbf{x}^{(i-1)} - \mathbf{x}^*||_2^2 - ||\mathbf{x}^{(i)} - \mathbf{x}^*||_2^2\Big) \\ &= \frac{1}{2t}\Big(||\mathbf{x}^{(0)} - \mathbf{x}^*||_2^2 - \cancel{||\mathbf{x}^{(1)} - \mathbf{x}^*||_2^2 + ||\mathbf{x}^{(1)} - \mathbf{x}^*||_2^2} + \dots\Big) \\ &= \frac{1}{2t}\Big(||\mathbf{x}^{(0)} - \mathbf{x}^*||_2^2 - ||\mathbf{x}^{(k)} - \mathbf{x}^*||_2^2\Big) \\ &\leq \frac{1}{2t}||\mathbf{x}^{(0)} - \mathbf{x}^*||_2^2 \end{aligned}$$

$\cancel{} \to 0$

We know from before that $f(x^{(k)})$ is non-increasing. Therefore,

$$f(\mathbf{x}^{(k)}) - f(\mathbf{x}^*) \leq \frac{1}{k}\sum_{i=1}^{k} f(\mathbf{x}^{(i)}) - f(\mathbf{x}^*) \leq \frac{||\mathbf{x}^{(0)} - \mathbf{x}^*||_2^2}{2tk}$$

which concludes the proof. We therefore see there is a convergence rate of $\mathcal{O}(1/k)$.

26.4.2. *Coordinate Descent.* Another plausible optimisation algorithm is coordinate descent, where each descent step optimises with respect to a single variable. For OLS regression we have,

$$\frac{\delta\mathcal{L}(\boldsymbol{\beta}_k)}{\delta\boldsymbol{\beta}_k} = \frac{1}{N}\sum_{n=1}^{N} x_{ik}(y_i - \mathbf{x}_i^T\boldsymbol{\beta}) \implies \beta_k^* = \frac{\sum_n x_{nk}y_n - \sum_n\sum_{j\neq k} x_{nk}x_{nj}\beta_j}{\sum_n x_{nk}^2}.$$

26.5. **Model Prediction.** Once we have computed our optimal model parameters, $\boldsymbol{\beta}^*$, we can start making predictions for unseen data, $\hat{\mathbf{x}}$. Our prediction is then,

$$\hat{y} = \hat{\mathbf{x}}^T\boldsymbol{\beta}^*.$$

26.6. **Maximum Likelihood Estimation.** Now we will derive our regression model from a probabilistic angle. Probabilistic constructions are preferred in machine learning because they make the models more comprehensible[58]. We start our derivation with the following assumptions: firstly, that the data points, $(\mathbf{x}_n, y_n)$ are independent and identically distributed (i.i.d); second, that the output variable is modelled by a linear function with Gaussian noise, that is, $\mathbf{y} = X\boldsymbol{\beta} + \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon}$ is a vector of normally distributed random variables, $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$, for which σ^2 is the variance representing the random noise. We are thereby making an assumption about how the output variable is distributed around our line of best fit, $\mathbf{x}_n^T\boldsymbol{\beta}$. Now we may write a probability distribution for y_n,

$$p(y_n|\mathbf{x}_n, \boldsymbol{\beta}) = \mathcal{N}(\mathbf{x}_n^T\boldsymbol{\beta}, \sigma^2),$$

and since the data points are independent, the distribution over all y_n is,

$$p(\mathbf{y}|X, \boldsymbol{\beta}) = \prod_{n=1}^{N} p(y_n|\mathbf{x}_n, \boldsymbol{\beta}).$$

As we will soon see, maximising this probability is equivalent to minimising the square loss function! Because of our assumption of Gaussian noise, the optimal parameters will be those that construct a hyperplane such that the data points around it are distributed according to a Normal distribution, that is, maximising the likelihood of the data around the plane. This is equivalent to minimising the square loss. As we shall see, it is convenient to work with the log of the likelihood. Since a log function is a monotonically increasing function, maximising the log-likelihood is no different to maximising the likelihood itself. Therefore we have,

[58]For some techniques, however, such as neural networks, things are not so straightforward.

$$\begin{aligned}
\max_{\boldsymbol{\beta}} \mathcal{L}_{lik}(\boldsymbol{\beta}) &= \max_{\boldsymbol{\beta}} \Big\{ \log \prod_{n=1}^{N} \mathcal{N}(\mathbf{x}_n^T \boldsymbol{\beta}, \sigma^2) \Big\} \\
&= \max_{\boldsymbol{\beta}} \Big\{ \sum_{n=1}^{N} \log \frac{1}{\sigma\sqrt{2\pi}} e^{-(y-\mathbf{x}_n^T\boldsymbol{\beta})^2/2\sigma^2} \Big\} \\
&= \max_{\boldsymbol{\beta}} \Big\{ -\frac{1}{2\sigma^2} \sum_{n=1}^{N} (y - \mathbf{x}_n^T \boldsymbol{\beta})^2 + \log \frac{N}{\sigma\sqrt{2\pi}} \Big\} \\
&= \max_{\boldsymbol{\beta}} \Big\{ -k_1 \mathcal{L}(\boldsymbol{\beta}) + k_2 \Big\} \\
&= \min_{\boldsymbol{\beta}} \mathcal{L}(\boldsymbol{\beta})
\end{aligned}$$

where k_1 and k_2 are positive constants, and so they have no bearing on the optimal $\boldsymbol{\beta}^*$. Thus, we see that maximising our likelihood function is equivalent to minimising our original mean square error loss function. This is equivalently known as the negative log-likelihood (NLL).

26.7. **Non-linear Fits.** If we wish to fit a non-linear function to our data, we can simply transform the data. For example, if we want to fit a quadratic function to our data (Figure 8), we need only square the values in our dataset and introduce it as a new dimension in our data. It is important to note that this is still a linear regression, as we still have a linear combination of parameters, albeit with a non-linear basis function. These more complex models are best created in the framework of a *ridge regression*, where we introduce a penalty term in the loss function.

26.8. **Bayesian Linear Regression.** Bayes model averaging may be performed for linear regression to obtain a more robust posterior predictive distribution, just as we saw for generative classifiers. Such a result can for example be analysed for its variance, unlike a mere point estimate such as ML or MAP estimates. The derivation involves multiplying and integrating Gaussians and belongs in a textbook, rather than in Last Chance Stats.

26.9. **Robust Linear Regression.** One drawback of using a Gaussian likelihood is that it is sensitive to outliers. This makes it likely to overfit in practice. There are several ways of mitigating this effect. The most crude is known as 'early stopping', where training is terminated before convergence, and before the model has a chance to overfit. The most ideal option is to train on more data, taking density away from outliers, but data is usually hard to come by. The most common approach is to *regularise* the cost function by introducing a prior distribution on the model parameters. This gives us the posterior distribution, just as it was with the beta-binomial model, and the optimal parameters are the MAP estimate, $\boldsymbol{\theta}_{MAP} = \text{argmax}_{\boldsymbol{\theta}} \text{NLL}(\boldsymbol{\theta})$, where NLL is the negative log-likelihood. If we choose another Gaussian for the prior, we are using l_2 regularisation, and we get what is known as a *ridge regression*,

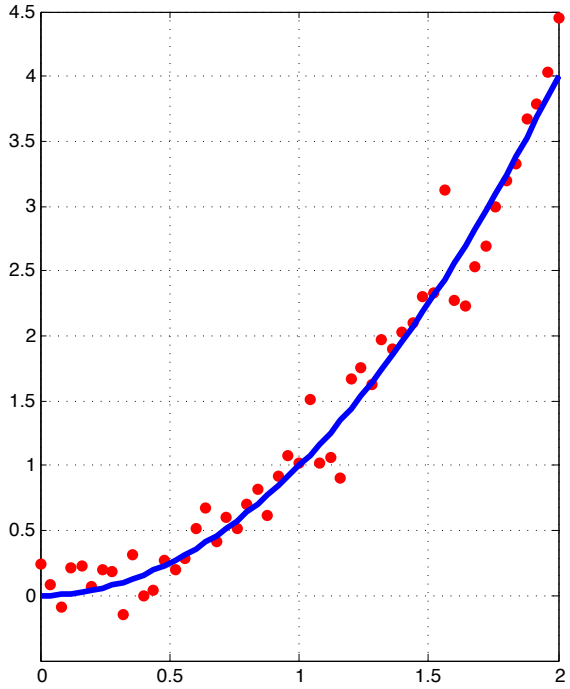


FIGURE 8. A non-linear fit may be found by transforming the data with a non-linear basis function.

$$\text{NLL} = \frac{1}{2}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) + \frac{\lambda}{2}\boldsymbol{\beta}^T\boldsymbol{\beta} = \frac{1}{2}\sum_{n=1}^{N}(y_n - \boldsymbol{\beta}^T\mathbf{x}_n)^2 + \frac{\lambda}{2}\sum_{d=1}^{D}\beta_d^2.$$

Clearly, this is a linear regression with an extra cost over the parameter values. This is known as a penalty term in the cost function. High parameter values, which correspond to more complex fits (overfitting), are punished with a quadratic cost. The parameter λ is a scaling constant which may be optimised through cross-validation. Ridge regression is very popular, and has the added bonus of producing a more numerically stable normal equation, because the regularisation term increases the eigenvalues of the gram matrix, $\mathbf{X^T X}$, in the normal equations, thereby reducing the ratio between the highest and lowest eigenvalues, and thereby improving its condition number. Another option is a Laplace prior. This is known as l_1 regularisation, and when applied to regression is known as a LASSO[59] regression. One of the effects of l_1 regularisation is to encourage *sparse* models. Sparsity refers to models whose parameters are non-zero for only a small subset of the dimensions. Encouraging sparsity is a form of *feature selection*. This is useful for high-dimensional data with many noisy indicators, otherwise known as the 'small N, large D' problem. The

[59]Least absolute shrinkage and selection operator

intuition behind this phenomenon of encouraging sparsity is that the contours of the prior distribution are linear, forming a diamond shape in the parameter space, whose vertices lie along each of the axes. In contrast, the Gaussian l_2 prior has ellipsoidal contours. Thus, in l_1 regularisation sparse parameters on the axes have greater magnitude than parameters along the edges of the diamond. These sparse parameters with greater magnitude (allowing for better fits) may be chosen at no extra penalty. Unfortunately, the cost function is no longer differentiable everywhere, due to the absolute value function, and our solution algorithm changes–the most common approaches are coordinate descent algorithms[60] such as the 'shooting' algorithm.

A final, more drastic option for robust regression is to modify the form of the likelihood to a distribution with heavier tails, making it more robust to outliers, thereby creating to *robust linear regression*. Typical choices are the Student or Laplace distributions. In the latter case, the function is convex but no longer differentiable, but with a reformulation trick can be made into a linear programming problem. For example, given a Laplace likelihood,

$$\text{NLL}(\mathbf{w}) \propto -\log \prod_n \exp\left(-|y_n - \mathbf{w}^T\mathbf{x}_n|\right) = \sum_n |y_n - \mathbf{w}^T\mathbf{x}_n|.$$

We can remove the absolute value sign using the *split variable trick*, that is, by introducing artificial positive and negative variables, r_n^+ and r_n^-, such that,

$$\text{NLL}(\mathbf{w}) = \sum_n r_n^+ + r_n^-,$$

where $r_n^+ - r_n^- = y_n - \mathbf{w}^T\mathbf{x}_n$ with the *type* constraint that $r_n^+ r_n^- = 0$ (ensuring only one variable is activated at a time), and $r_n^+, r_n^- \geq 0$. The type constraint can be omitted, as at optimality it will be satisfied as a matter of course. Now we have a constrained convex optimisation problem, which can be optimised with any linear solver, for example, the simplex algorithm.

27. The Simplex Algorithm

The simplex algorithm was designed by American mathematician George Dantzig[61] (1914-2005). It is an optimisation algorithm for linear programming problems, that is, a problem of the form,

[60] Coordinate descent algorithms are similar to gradient descent, but where at each step the function is optimised with respect to one variable.

[61] There is a famous anecdote from Dantzig's time as a doctoral student at Berkeley. Arriving late to class, Dantzig copied down two important unsolved statistics problems the professor had earlier written on the blackboard, taking them to be set for homework. With some effort, he managed to solve both problems, and submitted them to the professor for review. It was not until some weeks later that the professor looked at the solutions, to his amazement. Dantzig later had the rare privilege of submitting his homework as his doctoral thesis.

$$\begin{array}{rl} \text{maximise} & z = \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} = \mathbf{b}, x_i \geq 0 \end{array} \tag{21}$$

where z is known as the objective function, and the linear system specifies a set of linear constraints on the values of the input variables, x_i, which are further non-negative. The program specifies an N-dimensional convex solid[62], and the algorithm works by moving along the edges of this shape from corner to corner. As any optimal solution must be on the surface, we can discard all points interior to the solid. This reduces the feasible solution space to a finite set. Furthermore, the convexity of the solid guarantees that as long as we repeatedly take upward steps along the surface edges, we will arrive at a maximum in a finite number of steps. In practice the algorithm is very efficient, though there are a special class of problems for which the algorithm will run in exponential time. Now, take for example the simple program,

$$\begin{array}{rl} \text{maximise} & z = 2x_1 + x_2 \\ \text{subject to} & x_1 + x_2 \leq 5 \\ & 5x_1 + 2x_2 \leq 10 \\ & x_1, x_2 \geq 0 \end{array},$$

depicted in Figure 9. The first phase of optimisation involves reformulating the linear program into the *standard* form expressed in (21). First, any unrestricted variables are eliminated from the program. Then, any variable with a non-zero lower bound is replaced with a variable specifying an offset. Finally, *slack* variables are introduced to change any inequality constraint to an equality constraint. In our problem, two slack variables are required giving,

$$\begin{array}{rl} \text{maximise} & z = 2x_1 + x_2 \\ \text{subject to} & x_1 + x_2 + s_1 = 5 \\ & 5x_1 + 2x_2 + s_2 = 10 \\ & x_1, x_2, s_1, s_2 \geq 0 \end{array}.$$

The convenient way to apply the algorithm is to write the problem into a *tableau*. This can be written,

x_1	x_2	s_2	s_2	z	c
1	1	1	0	0	5
5	2	0	1	0	10
-2	-1	0	0	1	0

The tableau form permits the execution of the algorithm to follow a sequence of simple row reductions. We begin by considering the final line of the tableau. This indicates the rate of increase of the objective function with respect to a change in each variable. As a

[62]If we take a spherical fruit such as an apple to be our unconstrained convex function, adding a linear constraint corresponds to slicing it at some position at a fixed angle. Intuitively, the fruit remains convex after the cut, despite losing its smoothness (roundness).

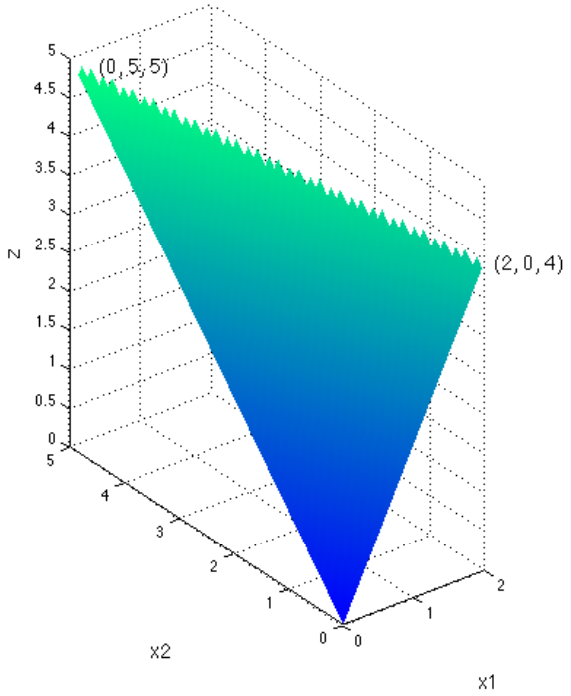


FIGURE 9. The convex surface of a simple linear programming problem.

simple heuristic, we select the variable yielding the greatest increase: x_1. We then select the constraint that most restricts increasing this variable–here, the second. The choice of row and column defines the *pivot* variable for the current iteration. The task now is to render the x_1 column an elementary vector with the unit in the pivot position. This can be achieved first by scaling the second row by $1/5$ ($R_2^* \leftarrow \frac{1}{5}R_1$), then subtracting it from the first row ($R_1^* \leftarrow R_1 - R_2^*$), and adding twice that to the final row ($R_3^* \leftarrow R_3 + 2R_2^*$), yielding,

x_1	x_2	s_2	s_2	z	c
0	**3/5**	1	$-1/5$	0	3
1	$2/5$	0	$1/5$	0	2
0	$-1/5$	0	$2/5$	0	4

Now we are at a point on the surface $(2, 0, 4)$ at which increases in x_1 yield 0 improvement to z. Hence, we are at a corner on the surface. The presence of negative coefficients in the final row indicates further improvement can be made. Identifying the next pivot element and repeating the process gives,

x_1	x_2	s_2	s_2	z	c
0	1	5/3	−1/3	0	5
1	0	−2/3	1/3	0	0
0	0	1/3	−1/15	0	5

Now we see the final row contains non-negatives for x_1 and x_2, therefore no improvement can be made by varying either variable, and hence we have arrived at an optimal solution. Because the state of R_3 (the objective function) has been arrived at as a combination of its previous state and the new first constraint, R_1, any solution which satisfies the under-determined R_1 satisfies its contribution to R_3. Hence, we choose the trivial solution, $(x_1, 5, 0, 0, 0)$. This may not always be the only optimal solution (though in our problem it is), but it is indeed optimal. Because of the linearly independent basis vectors, this leaves the trivial solution available in the second constraint, R_2, that is, $(0, x_2, 0, 0, 0)$. Combining these yields the optimal solution, $x_1 = 0, x_2 = 5, s_1 = 0, s_2 = 0$, as seen in Figure 9. Thus, in general, our method brings us to a state wherein the optimal solution can be read directly from the final tableau as the basic columns, with other variables equal to 0. As noted, this method verifiably gives us an optimal solution, though others may exist.

28. Logistic Regression

Logistic regression is a class of classification models. Learning a logistic regression can be thought of as fitting a multivariate Bernoulli distribution. There is also a strong connection between logistic regression and linear discriminant analysis. In fact, logistic regression models are the *discriminative* form of the same model. Logistic regressions are a core part of machine learning with ties to more sophisticated techniques such as neural networks. Studying logistic regressions notably presents the student with the quintessential optimisation techniques of machine learning. Linear and logistic regressions are part of a wider family of models called *generalised linear models* (GLM). This is a family of models modeling exponential densities over output variables, and where the mean is a linear combination of the parameters, in some cases passed through a non-linear function. A GLM can be characterised by a choice of probability distribution and a choice of *link function*. In OLS regression, this link function is the identity; in logistic regression it is the logistic function. In a *poisson regression*, the poisson distribution is used with a log link function. In this case, applying maximum likelihood gives a convex differentiable function and the regression may be performed with gradient descent. Recall a poisson distribution models the number of occurrences of an event with a given occurrence rate over a given time period. In the same way a logistic regression is more suitable for classification than OLS, a poisson regression is more suitable for modeling *rates*.

28.1. Binary Logistic Regression.

A binary logistic regression models,

$$p(y|\mathbf{x}, \boldsymbol{\beta}) = \text{Ber}(y|\sigma(\boldsymbol{\beta}^T\mathbf{x})),$$

where,

$$\sigma(\boldsymbol{\beta}^T\mathbf{x}) = \frac{\exp(\boldsymbol{\beta}^T\mathbf{x})}{1+\exp(\boldsymbol{\beta}^T\mathbf{x})},$$

is the sigmoid or logistic function[63]. Thus, in training a logistic regression model we are fitting a multivariate Bernoulli distribution. In this binary case, we encode the class variable as a binary variable, that is, $y_i \in \{0,1\}$, and we have $p(y=1) = \sigma(\boldsymbol{\beta}^T\mathbf{x})$ and $p(y=0) = 1-\sigma(\boldsymbol{\beta}^T\mathbf{x})$. The negative log likelihood for our model is therefore,

$$\begin{aligned}\mathrm{NLL}(\boldsymbol{\beta}) &= -\log\prod_i \sigma(\boldsymbol{\beta}^T\mathbf{x}_i)^{y_i}(1-\sigma(\boldsymbol{\beta}^T\mathbf{x}_i))^{(1-y_i)} \\ &= \sum_i \log(1+\exp(\boldsymbol{\beta}^T\mathbf{x}_i)) - y_i\boldsymbol{\beta}^T\mathbf{x}_i.\end{aligned}$$

Due to the logistic function, it is no longer possible to obtain a least-squares solution. We therefore have recourse to numerical techniques.

28.1.1. *Newton's method.* Because the function is convex, vanilla gradient descent will do the trick. However, more powerful techniques exist. Newton's method[64] is a similar algorithm incorporating second order information into the descent step. It comes from considering the second order Taylor polynomial for the NLL function, around a point $\boldsymbol{\beta}_k$,

$$NLL(\boldsymbol{\beta}) \approx NLL(\boldsymbol{\beta}_k) + g_k^T(\boldsymbol{\beta}-\boldsymbol{\beta}_k) + \frac{1}{2}(\boldsymbol{\beta}-\boldsymbol{\beta}_k)\mathbf{H}_k(\boldsymbol{\beta}-\boldsymbol{\beta}_k).$$

This quadratic expression is minimised for $\boldsymbol{\beta} = \boldsymbol{\beta}_k - \mathbf{H}^{-1}g_k$. This constitutes the Newton descent step. Now we will derive expressions for the gradient and Hessian of our logistic regression model. The gradient is the vector of partial derivatives,

$$g(\boldsymbol{\beta}) = \begin{bmatrix}\frac{\partial}{\partial\beta_0}\mathrm{NLL}(\boldsymbol{\beta}) \\ \frac{\partial}{\partial\beta_1}\mathrm{NLL}(\boldsymbol{\beta}) \\ \vdots \\ \frac{\partial}{\partial\beta_D}\mathrm{NLL}(\boldsymbol{\beta})\end{bmatrix} = \sum_i \begin{bmatrix}(\sigma(\boldsymbol{\beta}\mathbf{x}_i)-y_i)x_{i1} \\ (\sigma(\boldsymbol{\beta}\mathbf{x}_i)-y_i)x_{i2} \\ \vdots \\ (\sigma(\boldsymbol{\beta}\mathbf{x}_i)-y_i)x_{iD}\end{bmatrix} = \sum_i(\sigma_i-y_i)\mathbf{x}_i = \mathbf{X}^T(\boldsymbol{\sigma}-\mathbf{y}),$$

The Hessian is defined as $\frac{\partial}{\partial\boldsymbol{\beta}}g(\boldsymbol{\beta})^T$,

[63]This may be substituted by any function mapping $(-\infty,+\infty)\to[0,1]$, for example the Gaussian CDF, giving what is known as a *probit* regression. Recall these functions have roughly the same shape, but a Gaussian CDF may also be tuned by its variance.

[64]This is a multivariate generalisation of the Newton root-finding method. Here, however, we are finding the "root" of the gradient.

$$\mathbf{H} = \begin{bmatrix} \frac{\partial}{\partial\beta_0} g(\boldsymbol{\beta}) \\ \frac{\partial}{\partial\beta_1} g(\boldsymbol{\beta}) \\ \vdots \\ \frac{\partial}{\partial\beta_D} g(\boldsymbol{\beta}) \end{bmatrix} = \sum_i \begin{bmatrix} \sigma(\boldsymbol{\beta}\mathbf{x}_i)(1-\sigma(\boldsymbol{\beta}\mathbf{x}_i))x_{i1}\mathbf{x}_i^T \\ \sigma(\boldsymbol{\beta}\mathbf{x}_i)(1-\sigma(\boldsymbol{\beta}\mathbf{x}_i))x_{i2}\mathbf{x}_i^T \\ \vdots \\ \sigma(\boldsymbol{\beta}\mathbf{x}_i)(1-\sigma(\boldsymbol{\beta}\mathbf{x}_i))x_{iD}\mathbf{x}_i^T \end{bmatrix} = \sum_i \sigma_i(1-\sigma_i)\mathbf{x}_i\mathbf{x}_i^T = \mathbf{X}^T\mathbf{S}\mathbf{X},$$

where $S = \text{diag}([\sigma(\boldsymbol{\beta}^T\mathbf{x}_1), \sigma(\boldsymbol{\beta}^T\mathbf{x}_2), \ldots, \sigma(\boldsymbol{\beta}^T\mathbf{x}_N)])$. The Newton step may be rewritten as a least squares problem. In such a formulation, the optimisation algorithm is known as iterative reweighted least squares (IRLS).

28.1.2. *BFGS.* Due to the computational complexity of computing the Hessian, quasi-Newton techniques exist that create an approximation iteratively. The most common quasi-Newton method is the BFGS algorithm (Broyden, Fletcher, Goldfarb and Shanno). It is quasi-Newton because it relies on the secant equation[65],

$$\mathbf{H_{k+1}}(\mathbf{x_{k+1}} - \mathbf{x_k}) = \mathbf{g}_{k+1} - \mathbf{g}_k,$$

which is true in the limit, but only approximately true for a step of $\mathbf{x_{k+1}} - \mathbf{x_k}$. BFGS makes an update to the Hessian at each iteration,

$$H_{k+1} = H_k + \alpha\mathbf{u}\mathbf{u}^T + \beta\mathbf{v}\mathbf{v}^T,$$

equating to two rank-one updates (hence a rank-two update overall). The vectors $\mathbf{u}$ and $\mathbf{v}$ are chosen such that the secant equation is satisfied. This approximation makes the Hessian cheaper to compute and to invert. Particularly when the Hessian is sparse, BFGS can greatly speed up Newton's method.

The $\mathcal{O}(D^2)$ space complexity of the Hessian gives rise to a further approximation in the limited memory or L-BFGS algorithm. This is the usual weapon of choice for modeling high-dimensional data. Just as with linear regression, l_2 regularisation may be used. Unlike linear regression, however, the full posterior distribution is not directly computable due to the absence of a conjugate prior. Approximation techniques, such as Markov chain Monte Carlo (MCMC), are therefore used. An object-oriented logistic regression model is solved with both gradient descent and Newton's method in `logRegDemo.m`.

28.2. **Multinomial Logistic Regression.** Going beyond binary logistic regression requires some adjustments–the sigma function is replaced by the generalised softmax function,

$$p(y_i = c|\mathbf{x}_i, \mathbf{W}) = \frac{\exp(\mathbf{w}_c^T\mathbf{x})}{\sum_{c'=1}^{C}\exp(\mathbf{w}_{c'}^T\mathbf{x})},$$

[65]Similar to the fact that Newton's method is a generalisation of the Newton root-finding method, BFGS is approximates a multivariate generalisation of the secant root-finding method.

just as with linear discriminant analysis[66]. We now have the generalised multinomial or softmax regression. The parameters now make up a matrix $\mathbf{W}$ whose C columns correspond to each of the C classes[67]. Note that the matrix structure is only for notational convenience, the parameters still effectively constitute a vector of unknowns, and the gradient still has a vector structure. The binary encoding on the classes is dropped in favour of a 'one-of-C encoding' binary vector of length C. Thus, the negative log likelihood is,

$$\text{NLL}(\mathbf{W}) = -\sum_i \left[\left(\sum_{c=1}^{C} y_{ic} \mathbf{w}_c^T \mathbf{x}_i \right) - \log \left(\sum_{c'=1}^{C} \exp(\mathbf{w}_{c'}^T \mathbf{x}_i) \right) \right].$$

The gradient and Hessian may be derived as before, but it useful to introduce a specialised *Kronecker*[68] *tensor product* notation for specifying block matrices,

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1m}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2m}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}\mathbf{B} & a_{n2}\mathbf{B} & \cdots & a_{nm}\mathbf{B} \end{bmatrix}.$$

Now, we may derive the gradient as before, adhering to the first principle of forming the vector of partial derivatives,

$$g(\mathbf{W}) = \begin{bmatrix} \frac{\partial}{\partial \mathbf{w}_{11}} \text{NLL}(\mathbf{W}) \\ \vdots \\ \frac{\partial}{\partial \mathbf{w}_{D,C}} \text{NLL}(\mathbf{W}) \end{bmatrix} = \sum_i \begin{bmatrix} (\sigma(\mathbf{w}_1 \mathbf{x}_i) - y_{i1}) x_{i1} \\ \vdots \\ (\sigma(\mathbf{w}_c \mathbf{x}_i) - y_{ic}) x_{iD} \end{bmatrix} = \sum_i (\boldsymbol{\sigma}_i - \mathbf{y}_i) \otimes \mathbf{x}_i.$$

The Hessian is seemingly messy to derive, but it can be done on paper without too much trouble[69], yielding,

$$\mathbf{H}(\mathbf{W}) = \begin{bmatrix} \frac{\partial}{\partial \mathbf{w}_{11}} g(\mathbf{W}) \\ \vdots \\ \frac{\partial}{\partial \mathbf{w}_{D,C}} g(\mathbf{W}) \end{bmatrix} = \sum_i (\text{diag}(\boldsymbol{\sigma}_i) - \boldsymbol{\sigma}_i \boldsymbol{\sigma}_i^T) \otimes (\mathbf{x}_i \mathbf{x}_i^T),$$

giving a $DC \times DC$ matrix. The model can likewise incorporate a regularisation term. An example of multinomial logistic regression is given in `logRegDemo.m`.

[66] Clearly, the sigma function is the simplified binary form of the softmax function.

[67] This, incidentally leads to an identifiability issue. Identifiability is a property of statistical models whereby a model (the distribution) is defined by a unique parameter set, that is, there is a one-to-one mapping between parameters and models. It is important for precise statistical inference. In this instance the model is clearly not identifiable, as adding any constant vector to each of the parameter vectors gives the same model probabilities. To address this, the parameters are usually offset to eliminate the ambiguity.

[68] Leopold Kronecker (1823-1891) was a German mathematician, well known for his Kronecker delta–a shorthand for the indicator function for identity. This is not to be confused with the Dirac delta function, a Gaussian distribution with infinitesimal variance.

[69] It is simplest to try the 2-dimensional case with two classes to spot the pattern.

29. Online Learning and Perceptrons

29.1. **Online learning.** Online learning is an alternative to the more customary offline learning. In online learning, data is streamed rather than processed in a batch. That is, data samples arrive one at a time, and the parameters are adjusted at each iteration. The loss is compared to the batch performance in a function known as the *regret*. This leads to a technique known as online gradient descent. It is further possible to simulate online learning as an alternative to batch gradient descent in an algorithm known as *stochastic* gradient descent (SGD). Here, the gradient at a point is constructed on some subset of the available data, possibly a single data sample alone, rather than the full dataset. The algorithm cycles through the dataset in these smaller chunks, according to some random permutation. A complete cycle is known as an *epoch*, and multiple epochs may be required before convergence is reached. There are various problems where it may be beneficial to do this, for example when fitting neural networks whose cost function is not convex. In this case, the randomness of stochastic gradient descent can help escape local minima during the descent. Moreover, the algorithm may simply outperform batch gradient descent, as calculating the full gradient is $\mathcal{O}(N^2)$, but only $\mathcal{O}(N)$ when a single sample is used. It is likely that a small subset of the data can give a good approximation to the true gradient, in particular when duplication is present in the data. When applied to least squares, the algorithm is known as least mean squares (LMS). A comparison of batch and stochastic gradient descent is given in `regressionDemo.m`.

29.1.1. *The Perceptron.* The perceptron is an historically important algorithm. It is one of the earliest machine learning algorithms, invented by American psychologist, Frank Rosenblatt (1928-1971), in 1956. The perceptron learns a binary classifier in an online manner from supervised data, in a process that closely resembles stochastic gradient descent[70]. The algorithm is guaranteed to converge if the data is linearly separable, and there is a nice proof of this. The algorithm is furthermore guaranteed *not* to converge when the data is not linearly separable, and various enhancements exist to mitigate this problem. Notable examples of the perceptron's limitations are in learning logical functions. A perceptron can learn the `OR`, `AND`, and `NAND` functions, but is unable to learn `XOR`[71]. The multi-layer perceptron, also known as an artificial feed-forward neural network, arises from stacking several perceptrons in layers. These neural networks are much more flexible than the basic perceptron. The perceptron is also the origin of logistic regression, and maximum margin methods such as support vector machines (SVM). An example of a perceptron fitting the logical `NAND` function is given in `perceptronDemo.m`.

30. Neural Networks

A good starting point for studying the structure of a neural network is to consider a softmax regression model,

[70]The perceptron algorithm replaces the probability, $\mu_i = p(y_i = 1|\mathbf{x}_i)$, with the prediction, $\hat{y}_i$ (effectively rounding the probability), but is otherwise identical to SGD.

[71]Consider the four points arising from `XOR` in two variables–they are clearly not linearly separable.

$$\mathbf{f}(\mathbf{x}) = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

where $\mathbf{W} \in \mathbb{R}^{K \times D}$ are the weights and $\mathbf{b} \in \mathbb{R}^{K \times 1}$ (sometimes incorporated into into the weights via the bias trick) and $\text{softmax}(x) = \frac{\exp(x)}{\sum_{x'} \exp(x)}$ generalises the sigma logistic function. The softmax gives us a vector of K probabilities $\mathbf{p}_i \in \mathbb{R}^{K \times 1}$ for each input sample $\mathbf{x}_i$. The loss function $L = \frac{1}{N} \sum_{i=1}^{N} L_i$ where $L_i = D_{KL}(\mathbf{p}_i || p(y_i))$. Given the ground truth y_i puts all probability mass on one value, the divergence term reduces to $L_i = -\log(p_{y_i})$. Now, for a given *score*, $s_k = \mathbf{w}_k^T \mathbf{x} + b_k$, its gradient is given by, $\frac{\partial L_i}{\partial s_k} = -\frac{1}{p_{y_i}} \cdot \frac{\partial p_{y_i}}{\partial s_k}$. For $k = y_i$,

$$\frac{\partial p_{y_i}}{\partial s_k} = \frac{\partial}{\partial s_k} \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} = p_{y_i} - p_{y_i}^2 \implies \frac{\partial L_i}{\partial s_k} = p_{y_i} - 1$$

For $k \neq y_i$,

$$\frac{\partial p_k}{\partial s_k} = -p_k p_{y_i} \implies \frac{\partial L_i}{\partial s_k} = p_k$$

Hence, in general,

$$\frac{\partial L_i}{\partial s_k} = p_k - 1(k = y_i)$$

Now, to construct a true neural network, we introduce an intermediate *hidden* layer separating the input from the output,

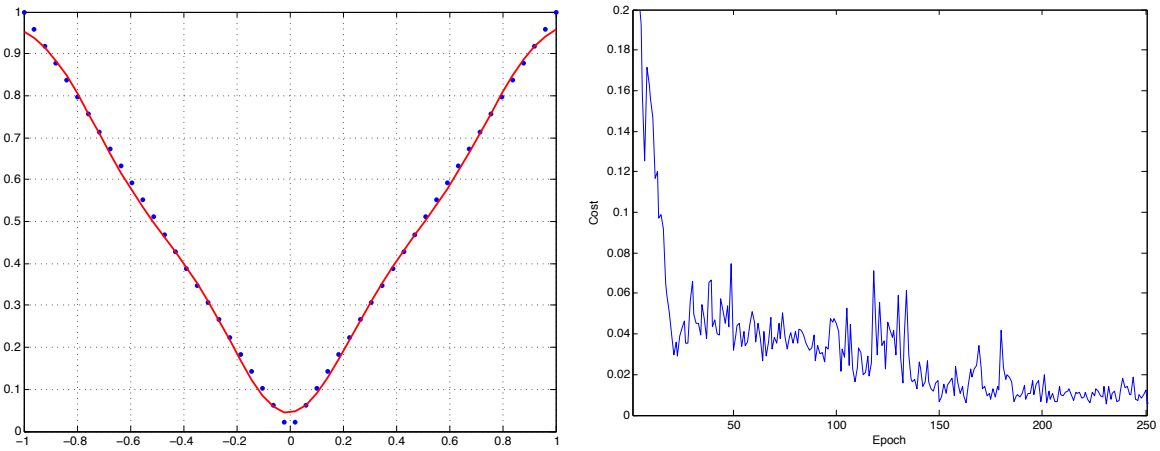
$$\mathbf{h}(\mathbf{x}) = \sigma(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \qquad \text{(hidden layer)}$$
$$\mathbf{f}(\mathbf{x}) = \text{softmax}(\mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}) \qquad \text{(output layer)}$$

This hidden layer creates an intermediate representation of the input data. The essence of *deep learning* is to model a hierarchy of multiple representations. Neural networks are therefore apt as a deep learning framework as, indeed, we may extend to a multi-layer network simply by stacking a desired layers,

$$\mathbf{f}(\mathbf{x}) = \sigma(\mathbf{W}^{(M)}\sigma(\mathbf{W}^{(M-1)}(\cdots \sigma(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \cdots) + \mathbf{b}^{(M-1)}) + \mathbf{b}^{(M)})$$

A neural network may be used for classification and regression, the output passed through mean-squared error in the latter case. Despite the complexity, this model can be trained with standard gradient descent techniques. Despite an elongated history, feed-forward neural networks and their variants have seen a resurgence of popularity in recent years, with the advent of deep learning. Neural networks are very powerful non-linear models, which are in fact capable of fitting any smooth function given enough layers, making them *universal approximators*. A simple example of a neural network fitting an absolute value function is given in Figure 10.

Note that we endow each hidden layer with its own set of weights. Though a neural network in non-linear and not convex, it remains differentiable and therefore trainable with gradient descent. The gradient of a model weight tells us locally how the loss function will increase for a unit increase to the weight. We therefore take a small (safe) step in



(A) (B)

FIGURE 10. A simple example of a neural network fitting an absolute value function (A) and the corresponding learning curve for stochastic gradient descent (B). Generated with `mlpDemo.m`.

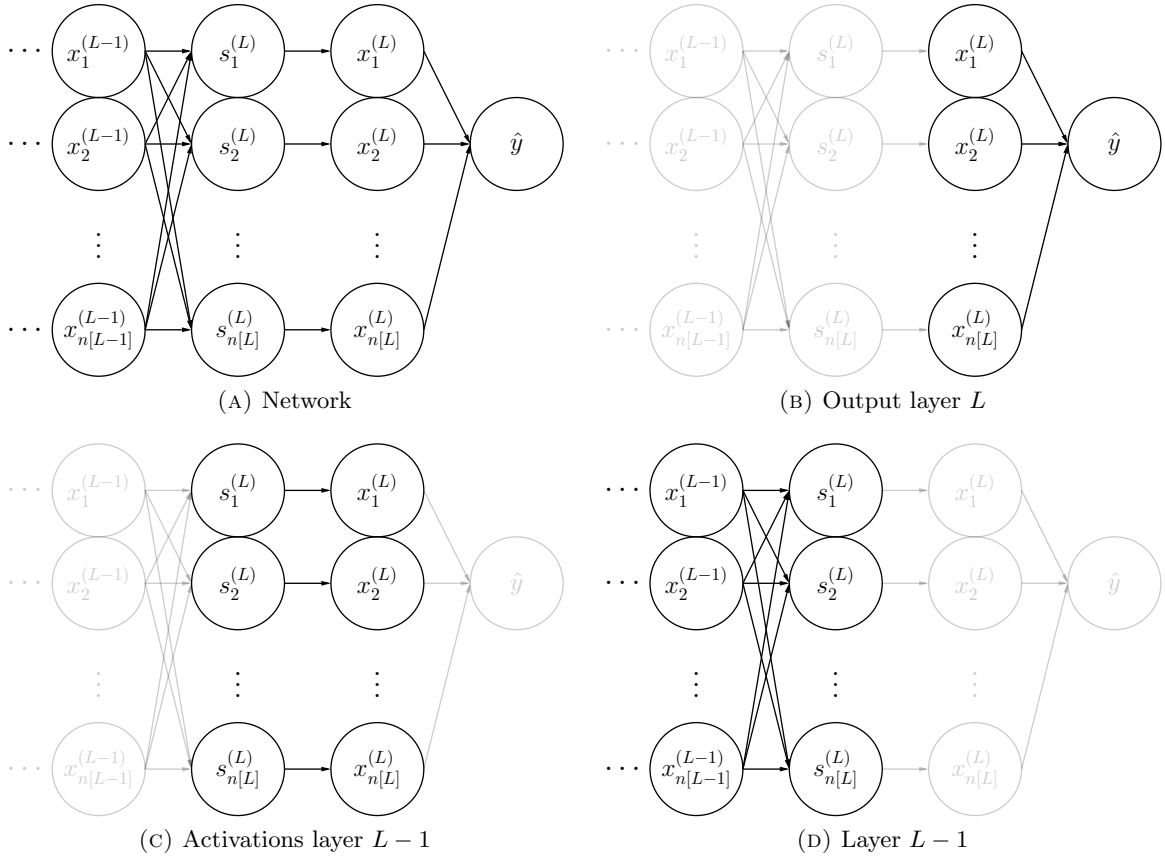
the opposite direction, that is, we reduce the weight by a proportional amount. The procedure for computing the gradients at each iteration of a gradient descent is called backpropagation. Backpropagation is an application of the chain rule to the graph structure of a neural network.

Consider a network with L layers and $n[l]$ nodes in layer l as in Figure 11. The forward computation at layer l is,

$$\begin{aligned} \mathbf{s}^{(l)} &= W^{(l)}\mathbf{x}^{(l-1)} \\ \mathbf{x}^{(l)} &= \sigma(\mathbf{s}^{(l)}) \end{aligned} \tag{22}$$

Backpropagation computes the gradients $\frac{\partial E}{\partial w_{ij}^{(l)}}$ for all weights indexed i, j, for all layers l.

To illustrate the backpropagation algorithm, first consider the network's output layer L, with a mean squared error loss, for a single data sample,



(A) Network (B) Output layer L

(C) Activations layer $L-1$ (D) Layer $L-1$

FIGURE 11. Backpropagation step by step.

$$E = (\hat{y} - y)^2 \tag{23}$$
$$= \left((\sum_{i=1}^{n[L]} w_i^{(L)} x_i^{(L)}) - y \right)^2$$

By the chain rule we have,

$$\frac{\partial E}{\partial w_i^{(L)}} = 2x_i^{(L)}(\hat{y} - y)$$

Now, what about the general case? To derive a rule, we consider the multivariable chain rule. This states that, given a function of functions, $E(x) = f(g_1(x), \ldots, g_m(x))$ for differentiable functions f and g_j then,

$$\frac{\partial E}{\partial x} = \sum_{j=1}^{m} \frac{\partial E}{\partial g_j} \cdot \frac{\partial g_j}{\partial x}$$

To show this, one first forms the first-order Taylor polynomial,

$$f(g_1(x), \ldots, g_m(x)) = f(g_1(x_0), \ldots, g_m(x_0)) + \sum_{j=1}^{m} \frac{\partial f}{\partial g_j}(z_j(x) - z_j(x_0)) + O(\boldsymbol{\delta}\mathbf{x}^2)$$

Subtracting $f(x_0)$ and divide through by $x - x_0$ then take the limit as $(x - x_0) \to 0$ to obtain the derivatives. It remains to show that the error goes to zero in the limit.

This result suggests that if we take the g_j to be the subnetwork from layer $l-1$ to layer l, and f to be the final $L - l$ layers of the network and apply the multivariable chain rule to obtain the gradients. If we compute the gradients in reverse order of layer, we can efficiently compute all network gradients, propagating gradients backwards to each layer in turn. Returning to our network, we can apply the multivariable chain rule. Let us assume the non-linearity is the logistic function such that $\sigma(x) = \frac{1}{1+e^{-x}}$ then,

$$\begin{aligned} \frac{\partial E}{\partial s_i^{(l)}} &= \sum_{j=1}^{m} \frac{\partial E}{\partial x_j^{(l)}} \cdot \frac{\partial x_j^{(l)}}{\partial s_i^{(l)}} \\ &= \frac{\partial E}{\partial x_i^{(l)}} \cdot \sigma(s_i^{(l)})(1 - \sigma(s_i^{(l)}) \end{aligned} \tag{24}$$

The core formula of backpropagation is then,

$$\frac{\partial E}{\partial x_i^{(l-1)}} = \sum_{j=1}^{m} \frac{\partial E}{\partial s_j^{(l)}} \cdot w_{ji}^{(l-1)},$$

that is, the gradient of a neuron is the sum of the gradients of all neurons emanating from that neuron, weighted by the weights of their connections. It is these error gradients that are backpropagated to layer $l - 2$. Finally, the weight gradients can be computed, before sending them to a gradient descent algorithm,

$$\frac{\partial E}{\partial w_{ij}^{l-1}} = \frac{\partial E}{\partial s_i^{(l)}} \cdot x_j^{(l-1)}$$

These three formulas can be vectorised as,

$$\frac{\partial E}{\partial \mathbf{s}^{(l)}} = \frac{\partial E}{\partial \mathbf{x}^{(l)}} \cdot \sigma(\mathbf{s}^{(l)}) \cdot (1 - \sigma(\mathbf{s}^{(l)}))$$

$$\frac{\partial E}{\partial \mathbf{x}^{(l-1)}} = (W^{(l)})^T \frac{\partial E}{\partial \mathbf{s}^{(l)}}$$

$$\frac{\partial E}{\partial W^{(l-1)}} = \frac{\partial E}{\partial \mathbf{s}^{(l)}}(\mathbf{x}^{(l-1)})^T$$

30.1. **Convolutional Neural Networks.** In the field of image analysis, the mask (or filter, or kernel) is an important construct. A *convolution* is an operation involving an initial image and the mask. The operation is equivalent to flipping the mask both vertically and horizontally and then visually placing the it over each pixel in turn. The output is the sum over a pixel-wise product of the mask and the sub-image. Masks are usually symmetric, so flipping is unnecessary. Recall from signal processing, the *convolution* between two functions,

$$(f * g)(t) \triangleq \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau$$

In image processing, a convolution between an image $\mathbf{I}$ and *kernel* $\mathbf{K}$ of size $d \times d$ and centered at a given pixel (x, y) is defined as,

$$(\mathbf{I} * \mathbf{K})(x, y) = \sum_{i=1}^{d}\sum_{j=1}^{d} \mathbf{I}(x + i - d/2, y + j - d/2) \times \mathbf{K}(i, j)$$

Convolutional neural networks are a family of neural network architectures having at least one convolutional layer. *LeNet* is the original CNN network architecture, bearing the name of Yann Lecun, who proposed it in the 90s. Its architecture can be written,

$$\begin{aligned}
\mathbf{H}_1 &= \sigma(\mathbf{X} * \mathbf{K}^{(1)}) && \text{(first convolutional layer)}\\
\mathbf{P}_1 &= \text{maxpool}(\mathbf{H}_1) && \text{(first pooling layer)}\\
\mathbf{H}_2 &= \sigma(\mathbf{P}_1 * \mathbf{K}^{(2)}) && \text{(second convolutional layer)}\\
\mathbf{P}_2 &= \text{maxpool}(\mathbf{H}_2) && \text{(second pooling layer)}\\
\mathbf{F}_1 &= \sigma(\mathbf{W}^{(1)}\mathbf{P}_2 + \mathbf{b}^{(1)}) && \text{(first fully-connected layer)}\\
\mathbf{F}_2 &= \sigma(\mathbf{W}^{(2)}\mathbf{F}_1 + \mathbf{b}^{(2)}) && \text{(second fully-connected layer)}\\
\mathbf{f}(\mathbf{X}) &= \text{softmax}(\mathbf{W}^{(3)}\mathbf{F}_2 + \mathbf{b}^{(3)}) && \text{(output layer)}
\end{aligned}$$

Convolutional Neural Networks (CNNs) are the current state-of-the-art in many computer vision tasks. Their great success in image classification[72] heralded a resurgence of interest in neural networks and deep learning. CNNs make the specific assumption of receiving image inputs and emulate the process of feature extraction using image masks. It has been found that using a pre-trained CNN as a general-purpose feature extractor for a simple linear model can yield significant improvements over even the most meticulously hand-crafted feature engineering). Another possibility is to use the pre-trained weights as an initialisation for training on a new dataset. Both are examples of transfer learning.

[72]See ImageNet Large Scale Visual Recognition Competition (LSVRC), 2012.

The cornerstone of the CNN is the *convolutional layer*, a hidden layer where a square grid of weights is convolved with the input, just like an image mask (now, however, the mask is learned as part of an all-encompassing model). The output of the convolutional layer is akin to a convolved image. Next, the non-linear activation function, ReLu (REctified Linear Unit), is applied to zero-out any negative values. Next there is a *pooling* layer, emulating *downsampling*. Here, each group of (usually) four values (pixels) is replaced by the maximum (sometimes the mean) of the four, leaving a single most intense pixel. This pooling method is known as *max pooling*. This sequence of `CONV->RELU->POOL` layers may be repeated multiple times to create a deep architecture. Finally, a few fully-connected layers round off the architecture. Though it seems far more sophisticated than a MLP, it can be shown that a CNN can be represented as a classical fully-connected neural network. For example, a convolutional layer can be represented as a sparse fully-connected layer. Various techniques have been developed for training these vast models, for example momentum optimisers, weight initialisation, batch normalisation[73], and *dropout*[74] for regularisation.

30.1.1. *Backprop through Convolutional Layers.* But how do we compute the gradient for the convolutional kernel? Let us consider the following simple convolution,

$$\underbrace{\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}}_{\mathbf{X}} * \underbrace{\begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix}}_{\mathbf{K}} = \underbrace{\begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix}}_{\mathbf{S}}$$

This gives the set of equations,

$$\begin{aligned} s_{11} &= k_{11}x_{11} + k_{12}x_{12} + k_{21}x_{21} + k_{22}x_{22} \\ s_{12} &= k_{11}x_{12} + k_{12}x_{13} + k_{21}x_{22} + k_{22}x_{23} \\ s_{21} &= k_{11}x_{21} + k_{12}x_{22} + k_{21}x_{31} + k_{22}x_{32} \\ s_{22} &= k_{11}x_{22} + k_{12}x_{23} + k_{21}x_{32} + k_{22}x_{33} \end{aligned}$$

The partial derivatives of the kernel's elements are therefore,

[73]Batch normalisation mitigates *internal covariate shift*, the fluctuations to the internal distributions of activations that make convergence so arduous.

[74]Dropout involves deactivating (setting to 0) hidden units in fully-connected layers according to a Bernoulli probability. This is repeated for every forward-backward pass of the network, such that every training example is trained on a different sub-graph of the network.

$$\begin{aligned}
\frac{\partial L}{\partial k_{11}} &= \frac{\partial L}{\partial s_{11}}x_{11} + \frac{\partial L}{\partial s_{12}}x_{12} + \frac{\partial L}{\partial s_{21}}x_{21} + \frac{\partial L}{\partial s_{22}}x_{22} \\
\frac{\partial L}{\partial k_{12}} &= \frac{\partial L}{\partial s_{11}}x_{12} + \frac{\partial L}{\partial s_{12}}x_{13} + \frac{\partial L}{\partial s_{21}}x_{22} + \frac{\partial L}{\partial s_{22}}x_{23} \\
\frac{\partial L}{\partial k_{21}} &= \frac{\partial L}{\partial s_{11}}x_{21} + \frac{\partial L}{\partial s_{12}}x_{22} + \frac{\partial L}{\partial s_{21}}x_{31} + \frac{\partial L}{\partial s_{22}}x_{32} \\
\frac{\partial L}{\partial k_{22}} &= \frac{\partial L}{\partial s_{11}}x_{22} + \frac{\partial L}{\partial s_{12}}x_{23} + \frac{\partial L}{\partial s_{21}}x_{32} + \frac{\partial L}{\partial s_{22}}x_{33}
\end{aligned}$$

It is evident that the kernel derivative is simply a convolution between the score "image" $\mathbf{S}$ and the input,

$$\underbrace{\begin{bmatrix} \frac{\partial L}{\partial k_{11}} & \frac{\partial L}{\partial k_{12}} \\ \frac{\partial L}{\partial k_{21}} & \frac{\partial L}{\partial k_{22}} \end{bmatrix}}_{\frac{\partial L}{\partial \mathbf{K}}} = \underbrace{\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}}_{\mathbf{X}} * \underbrace{\begin{bmatrix} \frac{\partial L}{\partial s_{11}} & \frac{\partial L}{\partial s_{12}} \\ \frac{\partial L}{\partial s_{21}} & \frac{\partial L}{\partial s_{22}} \end{bmatrix}}_{\frac{\partial L}{\partial \mathbf{Y}}}$$

By following a similar procedure for the input gradient, it is easy to see that,

$$\frac{\partial L}{\partial \mathbf{X}^{(l)}} = \frac{\partial L}{\partial \mathbf{S}^{(l)}} * \mathrm{rot180}\big(\mathbf{K}\big),$$

that is, the input gradient is (almost shockingly) equal to a convolution between the (zero-padded) scores gradient and a 180° rotation of the kernel matrix. To elucidate this somewhat, consider again that the convolution operation may be represented as a matrix operation, by stacking a flattened kernel at offsets and flattening the input image to a vector. Thus, our kernel matrix becomes,

$$\underbrace{\begin{bmatrix} k_{11} & k_{12} & 0 & k_{21} & k_{22} & 0 & 0 & 0 & 0 \\ 0 & k_{11} & k_{12} & 0 & k_{21} & k_{22} & 0 & 0 & 0 \\ 0 & 0 & 0 & k_{11} & k_{12} & 0 & k_{21} & k_{22} & 0 \\ 0 & 0 & 0 & 0 & k_{11} & k_{12} & 0 & k_{21} & k_{22} \end{bmatrix}}_{\mathbf{K}} \underbrace{\begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \\ x_{21} \\ x_{22} \\ x_{23} \\ x_{31} \\ x_{32} \\ x_{33} \end{bmatrix}}_{\mathbf{X}} = \underbrace{\begin{bmatrix} s_{11} \\ s_{12} \\ s_{21} \\ s_{22} \end{bmatrix}}_{\mathbf{S}}$$

Now, in the feed-forward setting, the gradient of the input is computed as the matrix product of the gradient of the loss, and the transpose of the weight matrix. It can be seen easily that a transposition of this sparse kernel matrix amounts to the half rotation of the kernel in a convolution operation.

30.1.2. *Backprop through max pooling layers.* Finally, consider the gradient for max pooling. We treat this similar to differentiating the ReLU activation function. If we have, for example, a 2×2 block of neurons $\boldsymbol{\sigma} = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$, then for pooling function p, we have $\frac{\partial L}{\partial \sigma_i} = \frac{\partial L}{\partial p(\boldsymbol{\sigma})} \cdot \frac{\partial p(\boldsymbol{\sigma})}{\partial \sigma_i}$, where $\frac{\partial p(\boldsymbol{\sigma})}{\partial \sigma_i} = 1_{\{\sigma_1 = \max(\boldsymbol{\sigma})\}}$.

30.1.3. *Transposed convolutional layers.* Transposed convolutional layers (sometimes incorrectly called *deconvolutional layers*[75]) play an essential role in networks where the compressed hidden layers are eventually *decompressed*, usually to the original input sizes. Examples of such networks are convolutional autoencoders, fully-convolutional networks for segmentation, and the generator networks of GANs. A deconvolution computation is similar to that of the convolutional gradient–they are implemented as a "convolutional transpose"–specifically, a convolution between the layer input and the rotation of the kernel. The gradient of a transposed convolutional layer is therefore a normal convolution between the gradient of the scores and the kernel.

Recall that convolutional layers may be used for downsampling (rather than a pooling layer) when the stride $s > 1$. Likewise, the convolutional transpose will *upsample* its inputs, back towards the original input size. In this case we have what is called *fractional stride* $1/s$. To see this, take the convolution of stride 2,

$$\underbrace{\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix}}_{\mathbf{X}} * \underbrace{\begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix}}_{\mathbf{K}} = \underbrace{\begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix}}_{\mathbf{S}}$$

The corresponding sparse kernel matrix is,

$$\begin{bmatrix} k_{11} & k_{12} & 0 & 0 & k_{21} & k_{22} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & k_{11} & k_{12} & 0 & 0 & k_{21} & k_{22} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & k_{11} & k_{12} & 0 & 0 & k_{21} & k_{22} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & k_{11} & k_{12} & 0 & 0 & k_{21} & k_{22} \end{bmatrix}$$

It can be seen that the transpose of this matrix, multiplied by the flattened scores gradient, $\frac{\partial L}{\partial \mathbf{S}^{(l)}}$, is equivalent to the convolution,

$$\frac{\partial L}{\partial \mathbf{X}} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & s'_{11} & 0 & s'_{12} & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & s'_{21} & 0 & s'_{22} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\frac{\partial L}{\partial \mathbf{S}}} * \underbrace{\begin{bmatrix} k_{22} & k_{21} \\ k_{12} & k_{11} \end{bmatrix}}_{\text{rot180}(\mathbf{K})}$$

[75] Formally speaking, a deconvolution inverts a convolution. Transposed convolutional layers cannot be said to do this.

which we can refer to as having $s = 1/2$. It has been shown that the transposed convolutional layers found in GANs and other models tend to an undesirable "checkerboard" effect in the image output. A simple workaround appears to be upsampling followed by a standard convolution[76].

30.1.4. *Upsampling layers.* While transposed convolutional layers have weights making them *learnable* upsampling layers, static forms of upsampling are used in convolutional networks too. One example is *bilinear interpolation*. It can easily be seen on paper that, for a fixed, integral-sized interpolation, this is equivalent to a fixed convolution applied with fractional stride. For example, a bilinear upsampling by a factor of 2 is equivalent to convolving the input with the kernel,

$$\begin{bmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 1 & 1/2 \\ 1/4 & 1/2 & 1/4 \end{bmatrix}$$

with a stride of $1/2$. Likewise, a nearest neighbour upsampling can be achieved by running a 2×2 convolution with $1/2$ stride.

30.2. **Recurrent Neural Networks.** Recurrent neural networks model sequential data. Just as CNNs share weights across space, RNNs share weights across time, thus also embodying an *inductive bias* or *structural prior*, the key to the success of deep learning models. The basic design is a set of input nodes, representing a sequence of vectors. These are combined linearly with a (shared) set of weights. For all but the first node in the sequence, these are added to a similarly linearly transformed previous hidden state, also with a shared set of weights. These are then fed through a non-linear activation function. From these, outputs are generated as a softmax of a further dense layer. This is expressed by the relations,

$$\begin{aligned} \mathbf{h}^{(t)} &= \tanh\left(\mathbf{W}\mathbf{x}^{(t)} + \mathbf{U}\mathbf{h}^{(t-1)} + \mathbf{b}\right) \\ \mathbf{y}^{(t)} &= \operatorname{softmax}\left(\mathbf{V}\mathbf{h}^{(t-1)} + \mathbf{c}\right) \end{aligned}$$

The transition diagram of a basic RNN is given in Figure 12. From this perspective, the recurrence of the network is represented as an arrow reflexively pointing back an internal state node. This reflects the sharing of weights, and the dependency of states over time–the actual computational graph remains directed and acyclic. Note that this generalises other architectural designs: though an RNN may map a sequence to another sequence, we can account for losses only for the last (or last k) output in the sequence, turning the RNN into a categorical classifier. Also, the spatial depth may be increased simply by adding extra hidden layers, without changing the basic principles.

An RNN accommodates the varying lengths of input sequences in a batch, with the "unfolded" computational graph (Figure 12) implicitly extended to the required length.

[76] https://distill.pub/2016/deconv-checkerboard/

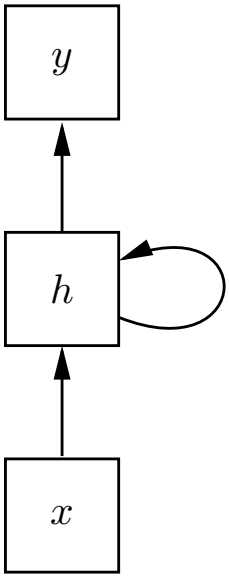


FIGURE 12. Transition diagram of an Elman network.

This is facilitated by the set of shared weights. The circuit is a dynamical system, as well as a universal Turing machine approximator.

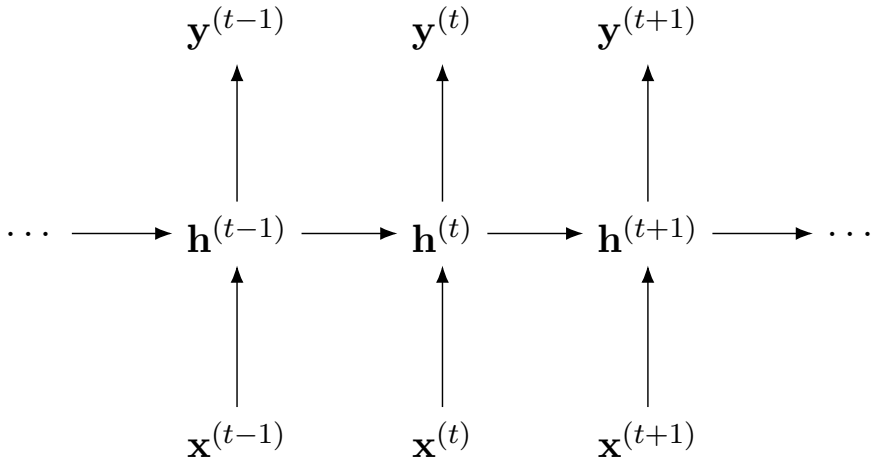


FIGURE 13. Computational graph resulting from "unfolding" an RNN's recurrent relation in time. Note the graph is directed and acyclic and trainable with backpropagation through time (BPTT).

RNNs are trained using *backpropagation through time* (BPTT), which simply consists of applying backpropagation on the unfolded computational graph. This is then no different to

backpropagation on feed-forward networks, with the gradients computed from the gradients of descendant nodes. However, the unfolded network requires storage of states over the whole sequence, incurring a greater demand on memory. If we replace *hidden-to-hidden* connections with *output-to-hidden* connections, the computation is much simplified, and the network is trained instead with a parallelisable backprop called *teacher forcing*. Such a network is, however, no longer a universal approximator.

30.2.1. *Specialised architecures.* RNNs have many application-specific architectures, where the computational graph is configured to serve a problem. An example is that of speech synthesis, where *bidirectional RNNs* are use to model paths running forwards and backwards along the input, to inform each hidden unit about past and future states (note this is done without violating the acyclic graph property). Another example is that of *encoder-decoder RNNs*, which have seen great success in machine translation. These model an input network producing a compressed *context* sequence[77]. This is then fed into a decoder network that produces an output sequence. The result is a network capable of decoding to a different length than the input, an essential capacity for translating phrases between languages. In *automatic captioning*, a CNN is first trained to classify images. The penultimate layer, a *CNN code*, is fed into an RNN as an initial state to generate a sentence describing the image, one of the more uncanny applications of deep learning. Another variant, *Recursive neural networks* are a less common variant that organise the input into a tree hierarchy.

30.2.2. *Training RNNs.* As in other forms of deep learning, the basic RNN architectures are decades old and mainstream research is devoted to the betterment of model training. Vanilla RNNs have historically been especially difficult to train, as the repeated application of the weights, both in the forward and backward directions, tends to lead to *vanishing* or *exploding gradients*, when the Jacobian's spectral radius is less than or greater than 1 respectively (this despite the interspersed non-linearities). This makes it particularly hard to model long-term dependencies in RNNs, an essential capacity for state-of-the-art sequence modeling.

An early attempt to enforce stability in RNNs were *leaky units* (Figure 14), which augmented the classical architecture with a recurrent self-connection to the hidden unit. The recurrence relation thus becomes,

$$\mathbf{h}'^{(t)} = (1-\alpha)\mathbf{h}^{(t-1)} + \alpha\mathbf{h}^{(t)}$$

This allows the network to accumulate knowledge over time in a stable way. Building upon the novelty of leaky units, *long short-term memory* (LSTM) constructs a *gated circuit*. This (remarkably) endows the RNN not only with the capacity to accumulate memory or knowledge, but to *choose to forget* (wipe its memory) over the course of a sequence. The advantage of such a capacity is that it allows the RNN to accumulate memory across *sub-sequences* within the greater sequence, as well as long-term dependencies. The LSTM circuit involves a set of gating mechanisms that take the input and feed it through a sigmoid

[77]What Geoffrey Hinton has described as a "thought".

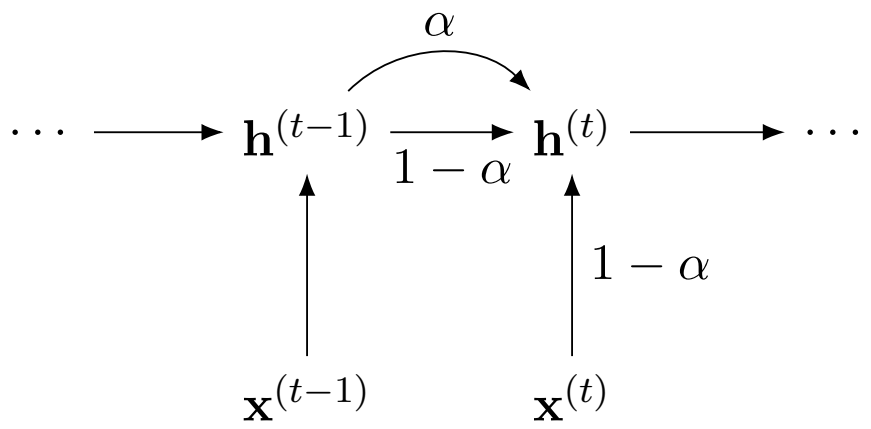


FIGURE 14. Leaky units are hidden units having an additional linear recurrent self-connection. These allow the RNN to accumulate memory over time. When the value of α is close to 1, the gradient is stable. The scalar α may be manually fixed or otherwise a learned parameter.

(0-1) non-linearity, allowing the network to turn on or off the flow of information through a hidden recurrent state whenever required. The LSTM equations extend the classical RNN with,

$$\begin{aligned}
\mathbf{f}^{(t)} &= \sigma\big(\mathbf{W}^f\mathbf{x}^{(t)} + \mathbf{U}^f\mathbf{h}^{(t-1)} + \mathbf{b}^f\big)\\
\mathbf{g}^{(t)} &= \sigma\big(\mathbf{W}^g\mathbf{x}^{(t)} + \mathbf{U}^g\mathbf{h}^{(t-1)} + \mathbf{b}^g\big)\\
\mathbf{s}^{(t)} &= \mathbf{f}^{(t)} * \mathbf{s}^{(t)} + \mathbf{g}^{(t)} * \sigma\big(\mathbf{W}\mathbf{x}^{(t)} + \mathbf{U}\mathbf{h}^{(t-1)} + \mathbf{b}\big)\\
\mathbf{q}^{(t)} &= \sigma\big(\mathbf{W}^o\mathbf{x}^{(t)} + \mathbf{U}^o\mathbf{h}^{(t-1)} + \mathbf{b}^o\big)\\
\mathbf{h}^{(t)} &= \mathbf{q}^{(t)} * \tanh\big(\mathbf{s}^{(t)}\big)
\end{aligned}$$

where the *forget gate* $\mathbf{f}$ controls the network's ability to wipe the hidden state, the gating unit $\mathbf{g}$ accumulates memory, the hidden state $\mathbf{s}$ models its own recurrent relation, and the output gate $\mathbf{q}$ combines to create the next recurrent hidden unit $\mathbf{h}$, as in classical RNNs. LSTM remains a differentiable, end-to-end trainable function. LSTM has more recently been generalised to *gated recurrent units* (GRU), which optimise the design of LSTM. The ideas of an implicit memory in RNNs has also been extended to an *explicit memory* in the recent *Neural Turing machines*, an end-to-end trainable network capable of reading and writing to *memory cells*. Despite the stabilising effects of gated units, exploding gradients may still arise and cause catastrophic damage to the training process. To avoid this, the relatively ad-hoc technique of *clipping gradients* was developed to rescale gradients whose norm exceed some nominal value.

30.2.3. *The constant error carousel.* To motivate LSTM, consider the trivial case of a single self-connected unit, and examine the conditions for it to have constant error flow. A constant error flow implies the gradients are equal in time,

$$\vartheta_j(t) = f_j'(net_j(t))\vartheta_j(t+1)w_{jj} \tag{25}$$

Setting $\vartheta_j(t) = \vartheta_j(t+1)$ yields,

$$f_j'(net_j(t))w_{jj} = 1.0. \tag{26}$$

Integrating this differential equation with respect to the neuron $net_j(t)$ yields,

$$f_j(net_j(t)) = \frac{net_j(t)}{w_{jj}} \tag{27}$$

Thus, f_j is linear (no activation) and,

$$y_j(t+1) = f_j(net_j(t+1)) = f_j(w_{jj} \cdot y^j(t)) = \frac{w_{jj}y^j(t)}{w_{jj}} = y^j(t). \tag{28}$$

i.e. the recurrent relationship is just a "copy-through" operation. This is what they call the *constant error carousel* (CEC). As will be seen, the CEC memory cell will remain a "copy-through" operation, ultimately acting like a skip layer that is accessed "on demand" (rather than at intervals as with a ResNet). Nevertheless, connecting with other units implies another sort of problem. Suppose backpropagation determines that copying an input to memory at some point in a training sequence is advantageous. This implies the weight should be increased to perform the copy. What then if elsewhere in the sequence another input is better to be ignored by the same weight? Now, the same weight should be decreased. Thus, the incoming weights (from input neurons) are obliged to both write relevant inputs and not write irrelevant inputs to the recurrent units, implying conflicting weight updates from different parts of the sequence. The same is true of the outputs copying from memory. Reading and writing should therefore be *context-sensitive*. LSTM does this by introducing gating functions,

$$y^{out_j}(t) = f_{out_j}(net_{out_j}(t)) = f_{out_j}\Big(\sum_u w_{out_j,u}y^u(t-1)\Big) \tag{29}$$

for the output gate and,

$$y^{in_j}(t) = f_{in_j}(net_{in_j}(t)) = f_{in_j}\Big(\sum_u w_{in_j,u}y^u(t-1)\Big) \tag{30}$$

for the input gate. The index u refers to any type of connection. In practice these will be a combination of input and hidden neurons (see below). These are usually annotated respectively as,

$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i), \tag{31}$$

and,

$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o), \tag{32}$$

The CEC state is updated according to,

$$s_{c_j}(t) = s_{c_j}(t-1) + y^{in_j}(t) g(net_{c_j}(t)) \tag{33}$$

$$= s_{c_j}(t-1) + y^{in_j}(t) g\Big(\sum_u w_{c_j,u} y^u(t-1)\Big), \tag{34}$$

that is, the application of the input gate. This encapsulates the following parts of the standard formulation,

$$\hat{c}_t = \sigma_c(W_c x_t + U_c h_{t-1} + b_c), \tag{35}$$

where $\hat{c}_t$ is like the "prospective" memory cell (subject to gating modifications) and,

$$c_t = c_{t-1} + i_t \circ \hat{c}_t. \tag{36}$$

Note that the "copy-through" operation preserves the constant error property with respect to the recurrent units. Thus, the new memory is a copy of the previous, with a context-sensitive residual. What about the forget gate? This wasn't introduced until 1999! The final part of LSTM is,

$$y^{c_j}(t) = y^{out_j}(t) h(s_{c_j}(t)), \tag{37}$$

that is, the application of the output gate.

$$h_t = o_t \circ \sigma_h(c_t). \tag{38}$$

Thus, the new hidden state is a copy of the new memory (itself a function of the old hidden state), with a context-sensitive filter for irrelevant information (the output gate). Error signals are said to be "trapped" in the memory cell. To see what is meant by this, consider the computational graph in Figure 15. The error for the last memory node c_4 is the product of its activation and the h_4 gradient. Now, the activation is linear, and furthermore, the recurrent weight is fixed at 1.0, so,

$$f'(net_{c_4}) = \frac{\partial}{\partial net_{c_4}} net_{c_4} / w_{c_4 c_4} = 1. \tag{39}$$

This is the essence of the constant error carousel. We obtain,

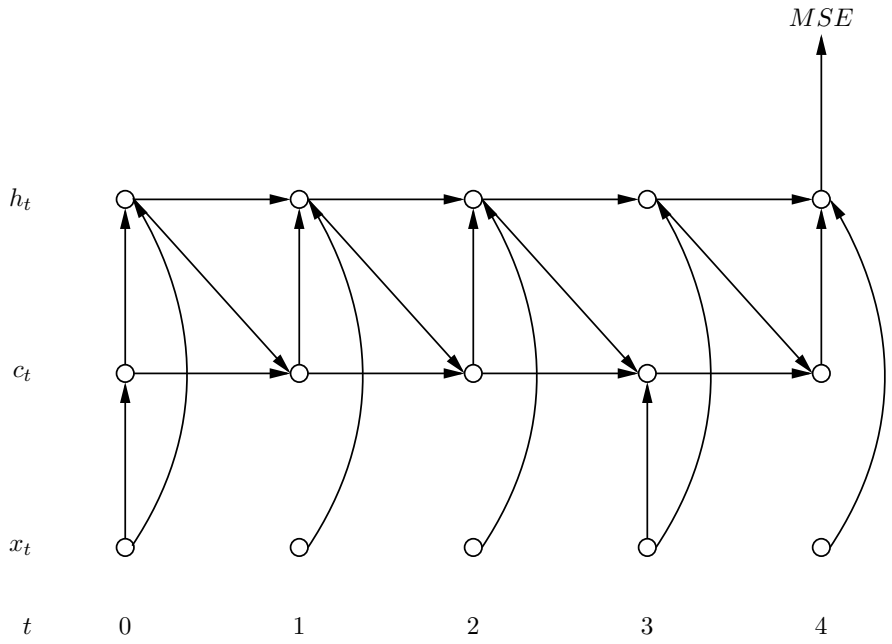


FIGURE 15. RNN unfolded in time for a single input, memory, and hidden unit, for 5 time steps. The missing arrows indicate a gated connection at that time point.

$$\vartheta_{c_4} = 1 \cdot w_{c_4 h_4} \vartheta_{h_4} \tag{40}$$

Following through, we have,

$$\vartheta_{c_3} = 1 \cdot 1 \cdot \vartheta_{c_4} = w_{c_4 h_4} \vartheta_{h_4}, \tag{41}$$

from which one can see how the error is "trapped". At the preceding node, another error is added:

$$\vartheta_{c_2} = 1 \cdot (\vartheta_{c_3} + w_{c_2 h_2} \vartheta_{h_2}) = w_{c_4 h_4} \vartheta_{h_4} + w_{c_2 h_2} \vartheta_{h_2}, \tag{42}$$

or rather "superimposed". Following through back to the initial node,

$$\vartheta_{c_0} = w_{c_4 h_4} \vartheta_{h_4} + w_{c_2 h_2} \vartheta_{h_2} + w_{c_1 h_1} \vartheta_{h_1} + w_{c_0 h_0} \vartheta_{h_0}, \tag{43}$$

and one can notice the errors have travelled back through time in through a stable shortcut, to better inform the weight updates of the input layer, gating functions, and earlier hidden states. Notice that h_0 will receive gradients that represent h_1, h_2, and h_4.

Note that, of course, the gates are soft and never entirely 0 or 1. It is to be assumed that this effect is negligible.

31. Support Vector Machines

Russian computer scientist, Vladimir Vapnik (1936-), invented the maximum-margin classifier in 1963, which later evolved into the kernelised support vector machine (SVM).

31.1. **Kernels.** Kernel functions are used to describe a similarity measure between objects. It is often convenient to work with a kernel of the data rather than the data itself. An example of this is tf-idf cosine similarity of a bag of words representation for document classification. Another example are Gaussian kernels, which give a multivariate Gaussian probability for the difference between two observations–the smaller the difference, the greater the probability. In linear models, it is possible to reformulate a model in terms of a kernel matrix $\mathbf{K} = \mathbf{\Phi}\mathbf{\Phi}^T$, where $\mathbf{\Phi}$ is some transformation of data $\mathbf{X}$ using basis function ϕ. This is known as the *kernel trick*, and it is particularly useful, for example, when we may apply a kernel such as the *radial basis function* whose corresponding basis function is infinite-dimensional, without having to compute the infinite-dimensional basis function itself. Models can be *kernelised* according to some reformulation, and replacing linear kernels (dot products) with another kernel function. In the case of kNN, this can be done quite readily. Kernelising ridge regression requires a reformulation trick known as the matrix inversion lemma. Support vector machines are kernelised by reformulating the problem into its dual problem, by application of the minimax theorem. The kernel trick requires a *Mercer* kernel be used, that is, such that the Gram matrix be symmetric positive-definite.

31.2. **Kernelised kNN.** The k nearest neighbours algorithm classifies a data point as the majority vote of the classes of the k closest training data points. By default, the distance metric is Euclidean distance, $||\mathbf{x}-\mathbf{x}'||_2^2 = \mathbf{x}^T\mathbf{x} + \mathbf{x}'^T\mathbf{x}' - 2\mathbf{x}^T\mathbf{x}'$. It is therefore clear we can replace the inner products with any kernel function of our choosing.

31.3. **Matrix Inversion Lemma.** We will here derive a result that we will use below, known as the Woodbury matrix identity, or the matrix inversion lemma. Given matrices $\mathbf{X} \in \mathbb{R}^{M\times N}$ and $\mathbf{Y} \in \mathbb{R}^{N\times M}$, consider,

$$\mathbf{XYX} + \mathbf{X} = \mathbf{X}(\mathbf{YX} + \mathbf{I}_N) \tag{44}$$

$$= (\mathbf{XY} + \mathbf{I}_M)\mathbf{X} \tag{45}$$

Equating (44) and (45), and multiplying by left and right inverses,

$$\begin{aligned}(\mathbf{XY} + \mathbf{I}_M)\mathbf{X} = \mathbf{X}(\mathbf{YX} + \mathbf{I}_N) &\implies \mathbf{X}^{-1}(\mathbf{XY} + \mathbf{I}_M)^{-1} = (\mathbf{YX} + \mathbf{I}_N)^{-1}\mathbf{X}^{-1} \\ &\implies (\mathbf{XY} + \mathbf{I}_M)^{-1}\mathbf{X} = \mathbf{X}(\mathbf{YX} + \mathbf{I}_N)^{-1},\end{aligned}$$

where the final step is achieved by multiplying both sides by $\mathbf{X}$ to the left and right.

31.4. **Reformulating Ridge Regression.** If we apply the matrix inversion lemma to the optimal parameters of a ridge regression, we obtain,

$$\boldsymbol{\beta}^* = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}_D)^{-1}\mathbf{X}^T\mathbf{y} = \mathbf{X}^T(\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I}_N)^{-1}\mathbf{y} = \mathbf{X}^T\boldsymbol{\alpha},$$

where $\boldsymbol{\alpha} = (\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I}_N)^{-1}\mathbf{y}$ is a vector of *dual* variables. This demonstrates that the optimal primal parameters $\boldsymbol{\beta}^*$ lie in the *row* space of $\mathbf{X}$. By contrast, the output $\mathbf{y} = \mathbf{X}\boldsymbol{\beta}$ lies in the *column* space of $\mathbf{X}$. There is a useful result called the *representer theorem* which states that the primal and dual solutions are always related in this way for models of like form, that is, models with a loss function and penalty term. This is useful, as it is easier to solve the dual problem for support vector machines.

31.5. **Decision Boundaries.** Consider a simple binary classifier,

$$f(\mathbf{x}) = \operatorname{sgn}\big(f(\mathbf{x})\big), \tag{46}$$

where $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$, that is, returning -1 if $f(\mathbf{x}) < 0$ and 1 if $f(\mathbf{x}) > 0$. The function is linear and has a linear decision boundary. It is useful to consider how the model parameters relate to the decision boundary. The decision boundary are those points $\mathbf{x}$ such that $f(\mathbf{x}) = 0$, hence on the boundary between the two classes. Consider the two-dimensional case where $f(\mathbf{x}) = w_1x_1 + w_2x_2 + b = 0$ (Figure 16A). For $\mathbf{x} = [x_1, x_2]$ on the decision boundary, consider $x_1 = 0 \implies x_2 = -b/w_2$, and $x_2 = 0 \implies x_1 = -b/w_1$. Then, the vector $\mathbf{x} = [0, -b/w_2] - [-b/w_1, 0] = [b/w_1, -b/w_2]$, runs parallel to the decision boundary. Clearly, $\mathbf{w}$ is orthogonal to $\mathbf{x}$, and this result can be shown to hold in higher dimensions also. A vector, $\mathbf{x}$, on the decision boundary and running parallel to $\mathbf{w}$, can be expressed as $\mathbf{x} = |\mathbf{x}| \cdot \mathbf{w}/|\mathbf{w}|$. Now, $f(\mathbf{x}) = \mathbf{w}^T|\mathbf{x}| \cdot \mathbf{w}/|\mathbf{w}| + b = 0$, hence $|\mathbf{x}| = -b/|\mathbf{w}|$. We thereby see that the bias term scales the distance of the decision boundary from the origin.

Among the infinite possible decision boundaries we could create, a promising heuristic for separating data would be to create a decision boundary such that the perpendicular distance between the nearest data points of each class is maximised[78]. Unlike the alternatives, this maximises the *space* left for unseen test cases that fall close to the boundary, likely increasing the performance of the classifier (Figure 16B). This principle is embodied in support vector machines (SVM). Therefore, SVMs are not based on statistical assumptions, rather on the rule of thumb that maximum margins perform better.

Now, any point $\mathbf{x}$ can be decomposed into the sum of its projection onto the decision boundary, $\hat{\mathbf{x}}$, and the residual, perpendicular to the boundary (and parallel to $\mathbf{w}$). Thus, $\mathbf{x} = \hat{\mathbf{x}} + r \cdot \mathbf{w}/|\mathbf{w}|$, where r is the length of the residual, hence the size of the margin of $\mathbf{x}$. Thus, $f(\mathbf{x}) = \mathbf{w}(\hat{\mathbf{x}} + r \cdot \mathbf{w}/|\mathbf{w}|) + b = |\mathbf{w}| \cdot r$, since $f(\hat{\mathbf{x}}) = 0$. Thus, we have $r = f(\mathbf{x})/|\mathbf{w}|$. Maximising the margin between the nearest points can therefore be expressed as the quadratic programming (QP) problem,

[78]Note that we are assuming temporarily that the data is linearly separable. This shortcoming will be addressed shortly.

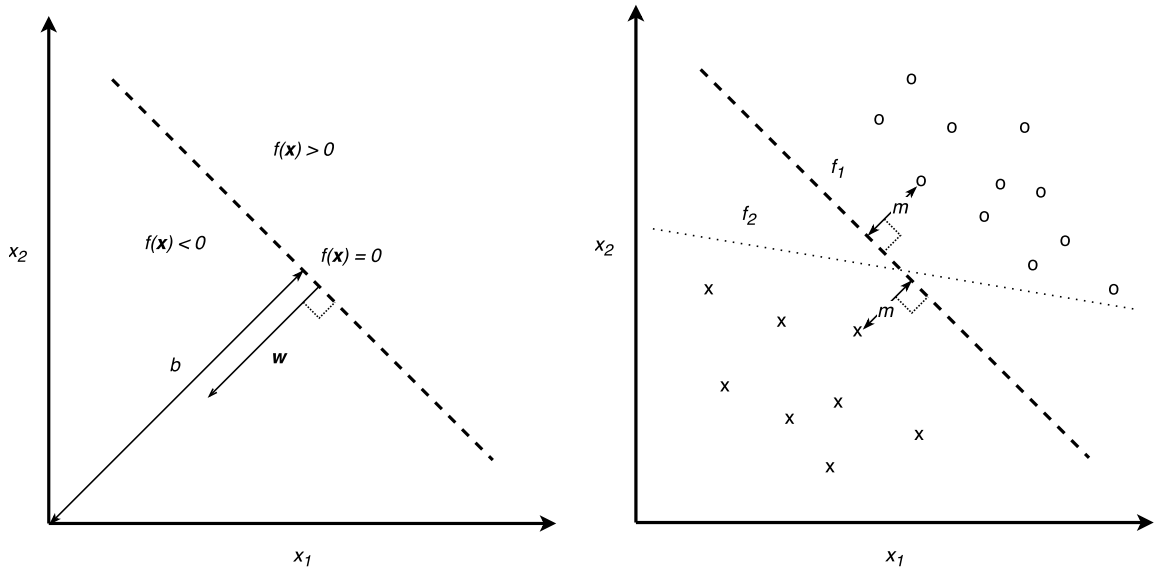


FIGURE 16. The relationship between a linear decision boundary and model parameters in two dimensions (A), and comparing a maximum margin, f_1, with an alternative, f_2 (B).

$$\begin{array}{rl} \min_{\mathbf{w},b} & \frac{1}{2}\mathbf{w}^T\mathbf{w} \\ \text{subject to} & y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \forall i \end{array}$$

where the constraints ensure each data point is correctly classified. Of course, the assumption of linear separability is in general untrue, so we introduce slack variables, ξ_i, to allow for data on the wrong side of the decision boundary, that is, for misclassifications. Thus, we replace each constraint with, $y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i$, leaving,

$$\begin{array}{rl} \min_{\boldsymbol{\xi},\mathbf{w},b} & \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{N}\xi_i \\ \text{subject to} & y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i, \forall i \\ & \xi_i \geq 0, \forall i \end{array}$$

where C is a weight for classification error. This problem is often expressed in terms of a hinge loss function, following the notation of other regression models. By the principle of convex duality, we can convert this problem into its dual form where the dual variables α_i are the Lagrange multipliers of the earlier constraints. The dual form is then the maximum of the *infimum*[79] of our current problem.

[79]The infimum of a set is its greatest lower-bound, the counterpart of the *supremum*, the least upper-bound. Some sets have these bounds without having a defined minimum or maximum. For example, the set $\{x : 0 < x < 1\}$ has no minimum or maximum, but its infimum is 0 and supremum is 1.

$$\begin{aligned} \min \quad & \mathcal{L}(\boldsymbol{\xi}, \mathbf{w}, b, \boldsymbol{\alpha}) = \tfrac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^N \xi_i - \sum_{i=1}^N \alpha_i(y_i(\mathbf{w}^T\mathbf{x}_i + b) + \xi_i - 1) \\ \text{subject to} \quad & \xi_i \geq 0, \forall i \end{aligned}$$

We can find the infimum of this expression by taking derivatives of each of the parameters, thus $\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i = 0 \implies \mathbf{w}^* = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i$ at optimality, which may be substituted into the objective function. For the bias term, $\frac{\partial \mathcal{L}}{\partial b} = \sum_{i=1}^N \alpha_i y_i = 0$ at optimality. For the slack variables, $\frac{\partial \mathcal{L}}{\partial \xi_i} = C - \alpha_i \leq 0$ (because of the slack constraints) $\implies \alpha_i \leq C$ at optimality. Thus, the parameters are eliminated by substitution, and the bias and slack variables correspond with constraints on the dual variables, giving the dual form,

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & \sum_{i=1}^N \alpha_i + \tfrac{1}{2}\sum_i^N \sum_j^N \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \\ \text{subject to} \quad & \sum_{i=1}^N \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C, \forall i \end{aligned} \tag{47}$$

thus a quadratic programming (QP) problem in the dual variables. Noting the relation between primal and dual parameters, we can rewrite the classifier (46) in terms of the dual variables as,

$$f(\mathbf{x}) = \operatorname{sgn}\left(\sum_{i=1}^N \alpha_i y_i \kappa(\mathbf{x}_i, \mathbf{x}) + b\right), \tag{48}$$

where $\kappa(\cdot, \cdot)$ is a Mercer kernel of our choosing, replacing the linear inner product from before. Apart from the dual form being conveniently kernelised, it affords an insightful alternative interpretation of the model. From (48), we can see how an observation, $\mathbf{x}$ is classified according to its similarity with the training vectors. Each training vector, $\mathbf{x}_i$, casts its vote, y_i, be it -1 or 1, with the influence regulated by a similarity measure provided by the kernel function, and the weight provided by the dual variable. Only the support vectors have a (non-zero) weighted vote. It is easy to see why this is such a powerful framework–we can choose between many similarity measures (kernels), though the mathematics restricts us to choosing Mercer kernels.

31.6. **Sequential Minimal Optimisation.** Our model (48) may be solved with any QP solver, which was formerly the standard approach. However, in 1998, American computer scientist, John Platt (1963-), invented sequential minimal optimisation (SMO), an algorithm for analytically optimising the dual variables two at a time. The superiority and simplicity of this algorithm helped bring support vector machines into the forefront of machine learning research. The algorithm works by iteratively optimising pairs of variables that violate the optimality conditions. These are,

$$\begin{aligned} \alpha_i = 0 &\implies y_i f(\mathbf{x}_i) \geq 1 \\ 0 < \alpha_i < C &\implies y_i f(\mathbf{x}_i) = 1 \\ \alpha_i = C &\implies y_i f(\mathbf{x}_i) \leq 1 \end{aligned} \tag{49}$$

Optimisation is done in pairs in order to maintain the equality constraint of the problem (47). Thus, for pair α_i and α_j, we make updates such that $y_i \Delta\alpha_i + y_j \Delta\alpha_j = 0$. Rearranging gives $\Delta\alpha_j = -\Delta\alpha_i y^{(i)}/y^{(j)} = -\Delta\alpha_i y^{(i)} y^{(j)}$, as since $y_i \in \{-1, 1\}$, division is the same as multiplication, and the cost function becomes,

$$\begin{aligned} W(\boldsymbol{\alpha}) = \sum_{k=1}^{N} \alpha_k + \Delta\alpha_i - \Delta\alpha_i y_i y_j - \frac{1}{2} \sum_{k=1}^{N} y_k \alpha_k (f(\mathbf{x}_k) - b) \\ - \Delta\alpha_i (f(\mathbf{x}_i) - f(\mathbf{x}_j)) - \Delta\alpha_i y_i (f(\mathbf{x}_i) - f(\mathbf{x}_j)) - \frac{1}{2} \Delta\alpha_i^2 (\mathbf{x}_i^T \mathbf{x}_i - 2\mathbf{x}_i^T \mathbf{x}_j + \mathbf{x}_j^T \mathbf{x}_j). \end{aligned}$$

Differentiating and solving gives,

$$\Delta\alpha_i^* = \frac{y_i \big(f(\mathbf{x}_j) - y_j - f(\mathbf{x}_i - y_i) \big)}{\mathbf{x}_i^T \mathbf{x}_i - 2\mathbf{x}_i^T \mathbf{x}_j + \mathbf{x}_j^T \mathbf{x}_j},$$

giving us our update step. The value of $\Delta\alpha_j^*$ can then be determined from the initial equality. Note that both variables remain constrained, hence their values must be clipped if they are less than 0 or greater than C. The bias term is then updated to account for the changes. For example, to ensure the classifier now emits y_i for $f(\mathbf{x}_i)$, we rearrange,

$$y_i = \sum_{k=1}^{N} \alpha_i y_i \mathbf{x}_k^T \mathbf{x}_i + b + \Delta b + \Delta\alpha y_i \mathbf{x}_i^T \mathbf{x}_i + \Delta\alpha y_j \mathbf{x}_i^T \mathbf{x}_j,$$

for Δb^*. The same is done for $f(\mathbf{x}_j)$, and the expressions are combined according to the convergence conditions (49). Thus, the optimisation steps are done analytically, rather than with linear algebra. The rest of the algorithm is mostly concerned with heuristics for choosing the best α_i, α_j pair. An implementation of support vector machines solved with the SMO algorithm is given in `supportVectorMachine.m`.

31.7. **Support Vector Machine Regression.** The ideas of support vector machines can be extended to regression models as well. This is achieved using a modification of the Huber loss function, creating a constrained quadratic programming (QP) problem. One of the shortcomings of SVMs is the difficulty in extending them to multi-class classification. This stems from the fact that they are not probabilistic models. However, schemes exist, for example, by training multiple classifiers in a hierarchy.

32. Decision Trees

Decision trees (also known as classification and regression trees–CART) are distinct from the previously discussed classifiers in that they are *non-parametric*. This means the model is not simply a mathematical function with a fixed form, and that the size of the model changes depending on the size of the data[80].

32.1. **Learning Decision Trees.** Just as with regression, we begin with a dataset, $\mathcal{D} = \{(\mathbf{x}_n, y_n)\}_{n=1}^N$. Now, however, the output variable, $y \in \{0, 1, \dots, K-1\}$, is a discrete variable taking a value in one of K categories. For simplicity, we will consider the case that $K = 2$, that is, binary classification. We therefore search for a way of splitting the data into two groups. We repeat this procedure for each of the subgroups recursively, until we satisfy some stopping condition. In so doing, we create a binary tree. Prediction can therefore be done by tracing a data point through the splitting conditions of the tree. Note this is an example of *embedded feature selection*.

The only question then is *how* we should split the data. For example, if we are classifying vehicles, and $y \in \{\text{car}, \text{motorbike}\}$, and our data consists of categories, $\mathbf{x} = [\text{number wheels}, \text{colour}, \text{milage}]$, the number of wheels is likely be a very effective indicator for sorting the cars from the bikes. In fact, it may be decisive, and rarely are indicators so informative in practice. The colour would likely tell us very little, but the milage might tell us something, as longer journeys are usually done by cars, so there may be a weak correlation. We therefore use an impurity measure to quantify the reduction in uncertainty upon making a given split. The best split is the one that minimises the uncertainty,

$$\begin{aligned}(k^*, \tau^*) &= \min_{k,\tau} I_{split}(X, k, \tau) \\ &= N_L I_{split}(p_L) + N_R I_{split}(p_R),\end{aligned}$$

where L and R denote the left and right splits of the data, N_L and N_R are the respective population sizes of the split data, and $p_L = \frac{\#(y=0)}{N_L}$ and $p_R = \frac{\#(y=0)}{N_R}$ are the sample probabilities of belonging to a given class, where in our fictitious data, $y = 0$ could indicate a car, and $y = 1$ a bike. Of course, when these probabilities go to 1, we are *certain* of the class of the remaining data. Note the split consists of two components: the variable of split x_k, and the decision rule τ. For example, with our fictitious data, the optimal split might be on $x_{k^*} = (\text{number wheels})$ and $\tau^* = (\text{number wheels} \leq 2)$. Clearly, this would more or less perfectly divide the bikes and the cars into distinct groups. In practice, we must systematically try all possible splits for each variable. If the variable is continuous, we must then discretise it in some way.

Several impurity measures are possible, but we know from information theory that uncertainty can be modelled with the *entropy* statistic. For binary classification, we have only two probabilities, $p(y = 0)$ and $p(y = 1) = 1 - p(y = 0)$. We therefore have the special case of binary entropy, $h_2(p)$, and our impurity measure becomes,

[80] Non-parametric methods could be said to "let the data do the talking".

$$I_{split}(p) = h_2(p) = -p\log p - (1-p)\log(1-p).$$

32.2. **Random Forest.** An extension of decision trees is the technique of *random forests*. Random forests construct multiple decision trees from subsets of the training data. Prediction is then done in an *ensemble* by aggregating the predictions of individual trees,

$$z(\mathbf{x}) = \frac{1}{M}\sum_{i=1}^{M} f_i(\mathbf{x}),$$

where the models f_i are individual decision trees trained on random subsets of the data. In such a scheme they are known as *weak learners*. Now, recall that,

$$\begin{aligned}\mathrm{Var}[X+Y] &= \mathbb{E}[(X+Y)^2] - \mathbb{E}[X+Y]^2 \\ &= \mathbb{E}[X^2 + 2XY + Y^2] - \mathbb{E}X^2 - 2\mathbb{E}[X]\mathbb{E}[Y] - \mathbb{E}[Y]^2 \\ &= \mathrm{Var}[X] + \mathrm{Var}[Y] + 2\mathrm{Cov}[X,Y]\end{aligned}$$

It is easy to then show that,

$$\mathrm{Var}\Big[\sum_i X_i\Big] = \sum_i \mathrm{Var}[X_i] + 2\sum_i\sum_{j\neq i}\mathrm{Cov}(X_i, X_j).$$

Consider then that in our random forest, each learner has a variance of σ^2 and $\mathrm{Cov}(f_i, f_j) = \rho\epsilon\forall i, j, i \neq j$. Then,

$$\mathrm{Var}\Big[\frac{1}{M}\sum_i f_i\Big] = \frac{\sigma^2}{M} + \rho\frac{M-1}{M}\sigma^2.$$

Thus, the ratio is reduced by, $M/(1+\rho(M-1))$. We therefore aim to decorrelate the individual trees. This is done in two ways. First, the training set is sampled (with replacement) to create M *bootstrapped* training sets of size N. Used in the ensemble, this is known as *bagging* (bootstrapping and aggregating). Secondly, train each learner on a random subset of the available features. The size of the subset is a model hyperparameter.

32.3. **Boosting methods.** Random forests are an example of a bagging (bootstrapped aggregation). A related, yet distinct, approach to learning, are boosting methods. These are meta algorithms to create ensembles of weak learners (often decision trees) by adding them incrementally into a strong learner as a weighted sum. For example, Adaboost trains a set of weak learners in an initial step. At each iteration, it selects for inclusion in the ensemble, the weak learner whose own misclassifications are best handled (least misclassified) by the existing ensemble. Its weight in the ensemble is then determined through a simple, analytical optimisation. Adaboost is a special case of the more general gradient boosting. The famous software XGBoost is an implementation of gradient boosting.

33. Dimensionality Reduction and PCA

The curse of dimensionality motivates techniques to transform high-dimensional data, given by an $N \times D$ matrix $\mathbf{X}$, to a lower rank approximation. This can be achieved by choosing lower dimensionality, $M < D$, and finding $D \times M$ orthonormal matrix $\mathbf{W}$ and $N \times M$ matrix $\mathbf{Z}$ such that,

$$\mathbf{X} \approx \mathbf{Z}\mathbf{W}^T,$$

and such that the *reconstruction error* is minimised. This leads to the objective function,

$$\min_{\mathbf{W},\mathbf{Z}} J(\mathbf{W}, \mathbf{Z}) = \frac{1}{N} \sum_{i=1}^{N} ||\mathbf{x}_i - \hat{\mathbf{x}}_i||^2,$$

where $\hat{\mathbf{x}}_i = \mathbf{W}\mathbf{z}_i$ is the reconstruction of the ith row. The reconstruction error function can be minimised (and the optimal decomposition found) with an algorithm known as alternating least squares (ALS).

Assuming $\mathbf{W}$ is orthonormal, it can be shown that the optimal $\mathbf{Z}$ will be projections onto $\mathbf{W}$. Thus, is can be shown that the optimal $\mathbf{W}$ will equivalently:

- minimise the reconstruction error
- minimise the residuals of the projections
- maximise the variance of the projected data

Examples of this equivalence is given in Figure 17.

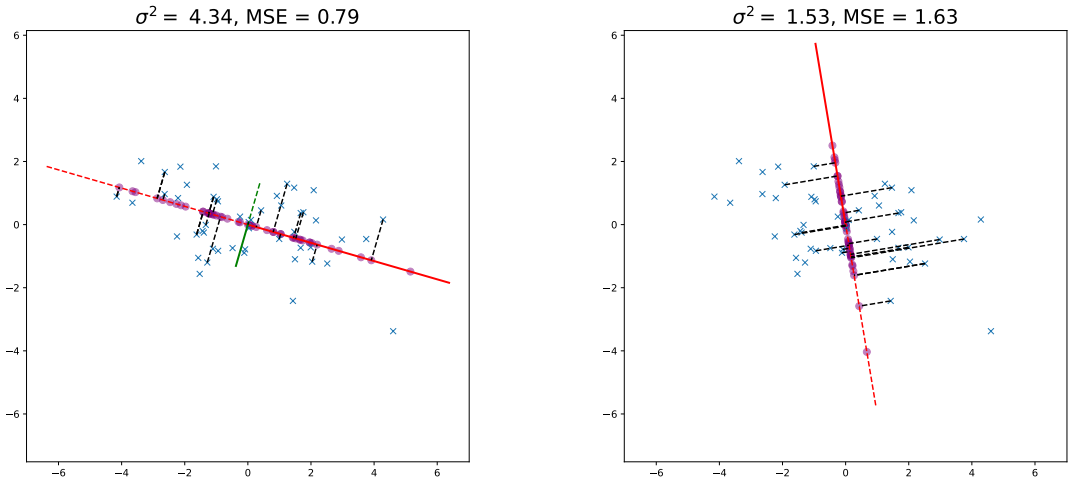
It can be shown analytically that the optimal orthonormal choice occurs when $\mathbf{W}$ is the matrix of the M most significant eigenvectors (highest eigenvalues) of the sample covariance matrix, $\hat{\mathbf{\Sigma}} = \frac{1}{N-1} \sum_i \mathbf{x}_i \mathbf{x}_i^T = \frac{1}{N-1} \mathbf{X}^T \mathbf{X} \in \mathbb{R}^D$ for the D features of $\mathbf{X}$. These are otherwise known as the *principal components* of $\mathbf{X}$, which brings us to principal component analysis (PCA), a highly popular technique for dimensionality reduction. If we can find $\mathbf{X} \approx \mathbf{Z}\mathbf{W}^T$ for lower-dimensional, orthonormal $\mathbf{W}$, then $\mathbf{X}\mathbf{W} \approx \mathbf{Z}\mathbf{W}^T\mathbf{W} \implies \mathbf{X}\mathbf{W} \approx \mathbf{Z}$, hence $\mathbf{W}$ maps our data to a lower-dimensional approximation. When the transformation is made, a model may be trained, perhaps more successfully than with the full data.

33.1. **Derivation of PCA.** Given a set of data $\mathbf{X} \in \mathbb{R}^{N \times D}$, compute the unit vector $\mathbf{w}^* \in \mathbb{R}^{D \times 1}$ that passes through the middle of the data manifold, that is, which minimises the residuals,

$$\mathbf{w}^* = \min_{\mathbf{w}} \sum_{i=1}^{N} ||\mathbf{r}_i||_2^2 \tag{50}$$

where the residual $\mathbf{r}_i = \mathbf{x}_i - proj_{\mathbf{w}}(\mathbf{x}_i)\mathbf{w}$, and such that,

$$\mathbf{w}^T\mathbf{w} = 1. \tag{51}$$



(A) Principal components (B) Other

FIGURE 17. Principal components analysis identifies the optimal line of projection (left, red line) versus any alternative (right, red line). The choice is optimal as it minimises the residuals (black dashed lines), or, equivalently, maximises the variance of the project data (spread of purple point along red lines).

Note that for unit $\mathbf{w}$, $proj_{\mathbf{w}}(\mathbf{x}_i) = \mathbf{x}_i^T\mathbf{w}$. By the method of Lagrange multipliers, the constraint Equation 51 may be substituted into Equation 50, yielding,

$$\mathbf{w}^*, \lambda^* = \min_{\mathbf{w},\lambda} \sum_{i=1}^{N} ||\mathbf{r}_i||_2^2 - \lambda \cdot (\mathbf{w}^T\mathbf{w} - 1) \tag{52}$$

Note that,

$$\begin{aligned} \sum_{i=1}^{N} ||\mathbf{r}_i||_2^2 &= \sum_{i=1}^{N}\sum_{j=1}^{D}(x_{ij} - \mathbf{x}_i^T\mathbf{w} \cdot w_j)^2 \\ &= tr((\mathbf{X} - \mathbf{X}\mathbf{w}\mathbf{w}^T)^T(\mathbf{X} - \mathbf{X}\mathbf{w}\mathbf{w}^T)) \\ &= tr(\mathbf{X}^T\mathbf{X} - \mathbf{w}\mathbf{w}^T\mathbf{X}^T\mathbf{X} - \mathbf{X}^T\mathbf{X}\mathbf{w}\mathbf{w}^T + \mathbf{w}\mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w}\mathbf{w}^T) \\ &= tr(\mathbf{X}^T\mathbf{X}) - \mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} \end{aligned} \tag{53}$$

where the second step is simply the squared Frobenius norm, and the final step leverages Equation 51. Substituting into Equation 52 and differentiating,

$$\frac{\partial}{\partial \mathbf{w}}\Big(tr(\mathbf{X}^T\mathbf{X}) - \mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w} - \lambda\cdot(\mathbf{w}^T\mathbf{w}-1)\Big) = 2\mathbf{X}^T\mathbf{X}\mathbf{w} - \lambda\mathbf{w} \tag{54}$$

Setting the derivative to zero and rearranging, one obtains,

$$\frac{1}{N-1}\mathbf{X}^T\mathbf{X}\mathbf{w}^* = \frac{\lambda}{2N-2}\mathbf{w}^*, \tag{55}$$

namely, $\mathbf{w}^*$ is an eigenvector of the sample covariance matrix $\frac{1}{N-1}\mathbf{X}^T\mathbf{X}$.

33.2. **Singular Value Decomposition.** Singular Value Decomposition (SVD) is a factorisation that exists for all matrices $\mathbf{X}$ written,

$$\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^T,$$

where matrices $\mathbf{U}$ and $\mathbf{V}$ are orthonormal, and $\mathbf{S}$ is the diagonal matrix of *singular values*. To relate SVD back to PCA, note that the empirical covariance matrix,

$$\hat{\mathbf{\Sigma}} = \mathbf{X}^T\mathbf{X} = \mathbf{V}\mathbf{S}\mathbf{U}^T\mathbf{U}\mathbf{S}\mathbf{V}^T = \mathbf{V}\mathbf{S}^2\mathbf{V}^T,$$

which is an eigenvalue decomposition, indeed the singular value decomposition of the square covariance matrix. Note the singular values are the squares of those of $\mathbf{X}$. This implies that $\mathbf{W} = \mathbf{V}$ and $\mathbf{Z} = \mathbf{X}\mathbf{W} = \mathbf{U}\mathbf{S}\mathbf{V}^T\mathbf{V} = \mathbf{U}\mathbf{S}$. The principal components are therefore the columns of the right singular matrix, $\mathbf{V}$, equivalent to the eigenmatrix of the covariance matrix. Thus, SVD (which may be performed with various techniques) is an alternative to ALS for acquiring the reconstruction matrices for PCA. In this form, it is clear that a matrix multiplication can be understood in three steps: a rotation $\mathbf{U}$, a scaling $\mathbf{S}$ and a final rotation $\mathbf{V}$. For a cloud of points, $\mathbf{X}$, the principal components give the trend lines in orthogonal directions that maximise the variance. These trends can be thought of as latent variables underlying the observations and this is the essence of information retrieval techniques such as latent semantic indexing (LSI).

Without truncation, the data may be decorrelated using the decomposition. Consider mapping $\mathbf{X}$ to $\mathbf{Z} = \mathbf{X}\mathbf{V}$. Then, the covariance matrix of $\mathbf{Z}$ is $\mathbf{Z}^T\mathbf{Z} = \mathbf{V}^T\mathbf{X}^T\mathbf{X}\mathbf{V} = \mathbf{V}^T(\mathbf{V}\mathbf{S}\mathbf{U}^T)(\mathbf{U}\mathbf{S}\mathbf{V}^T)\mathbf{V} = \mathbf{S}^2$. Thus, the covariance matrix of $\mathbf{Z}$ is diagonal. We can further *whiten* the data by the transformation $\mathbf{Z}_{\text{white}} = \mathbf{X}\mathbf{V}\mathbf{S}^{-1}$. It is easily shown that now that the covariance matrix, $\mathbf{Z}_{\text{white}}^T\mathbf{Z}_{\text{white}} = \mathbf{S}^{-1}\mathbf{V}^T(\mathbf{V}\mathbf{S}^2\mathbf{V}^T)\mathbf{V}\mathbf{S}^{-1} = \mathbf{I}$.

33.3. **Multi-dimensional Scaling.** Multi-dimensional scaling is a simple means of dimensionality reduction, usually used to visualise high-dimensional data in two or three dimensions. Starting with a distance matrix $\mathbf{D} \in \mathbb{R}^{N\times N}$ between N points in D dimensions, one can obtain a d-dimensional representation by invoking $\mathbf{x}_i \in R^d$ and minimising the convex function,

$$\sum_{i,j}(|\mathbf{x}_i - \mathbf{x}_j|_2^2 - D_{ij})^2$$

33.4. **t-distributed Stochastic Neighbour Embedding.** t-SNE (t-distributed stochastic neighbour embedding) defines probabilities between points in high-dimensional space by invoking a Gaussian centered on each point, then calculating the probability of choosing each other point. These are then normalised. Then t-SNE creates an embedding in a lower dimension by choosing coordinates such that the corresponding, similarly-defined probabilities are close to the former ones. This is ensured by minimising the KL-divergence between the set of distributions. t-SNE is not recommended as a general dimensionality reduction technique, rather a visualisation tool. It is somewhat similar to multi-dimensional scaling (MDS), MDS defines Euclidean distances (the log of a spherical Gaussian) and has a convex objective function.

34. Structured Sequence Learning

34.1. **Hidden Markov Models.** Hidden Markov models (HMMs) are a staple of natural language processing (NLP) and other engineering fields. A HMM models a probability distribution over an unknown, *hidden* sequence of states of length T, $\mathbf{y} = (y_1, y_2, ..., y_T)$, whose elements take on values in a finite set of states, S, and follow a Markov process. For each element in this hidden sequence, there is a corresponding observation element, forming a sequence of *observations*, $\mathbf{x} = (x_1, x_2, ..., x_T)$, similarly taking values in a finite set, O. The graphical structure of a HMM (Figure 18) shows the dependencies between consecutive hidden states (these are modelled with *transition* probabilities), and states and their observations (modelled with *emission* probabilities). The first dependency is referred to as the Markov condition, which postulates the dependency of each hidden state, y_t, on its k precursors in the hidden sequence, namely, $\mathbf{y}_{t-k:t-1}$[81]. In the discussion that follows, we assume the Markov condition to be of first degree, that is, $k = 1$. Incidentally, higher-order HMMs may always be reconstructed to this simplest form. For example, for $k = 2$ it suffices to define $S' = S \times S$ and $O' = O \times O$, that is, compound transition and emission probabilities. Thus, we have a first-order HMM with variables $Y'_t = (Y_{t-1}, Y_t)$. The second dependency assumption may be referred to as *limited lexical conditioning*, referring to the dependency of an observation only on its hidden state. Properties of the model may then be deduced through statistical inference, for example, a prediction of the most likely hidden sequence can be computed with the Viterbi algorithm (Section 34.1.1).

HMMs have been shown to be successful in statistical modelling problems. In Part of Speech (PoS) tagging, a classic NLP problem for disambiguating natural language sentences, the parts of speech (nouns, verbs, and so on) of a word sequence (sentence) are modelled as hidden states, and the words themselves are the observations. The PoS sequence may be modelled and predicted for as a HMM. Even a simple HMM can achieve an accuracy of well over 90%.

[81] In the following we use the notation $\mathbf{x}_{a:b}$ to refer to the elements of vector $\mathbf{x}$ from index a through b inclusive.

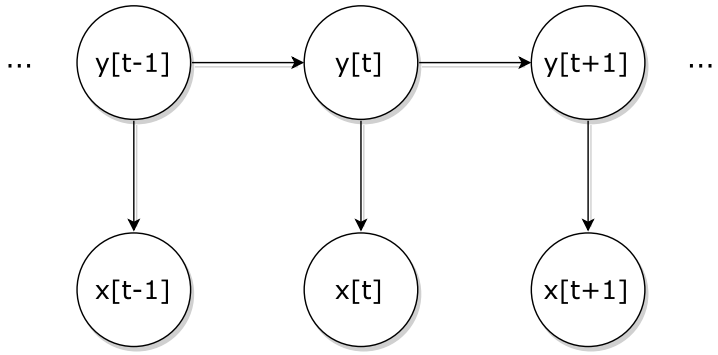


FIGURE 18. An illustration of the graphical structure of a Hidden Markov Model (HMM). The arrows indicate the dependencies running from dependee to dependent.

We may build a HMM by first forming the joint probability distribution of the hidden state sequence and the observation sequence,

$$\mathrm{p}(\mathbf{x}, \mathbf{y}) = \mathrm{p}(\mathbf{x}|\mathbf{y})\mathrm{p}(\mathbf{y}). \tag{56}$$

Applying the chain rule and the two dependency assumptions, we acquire,

$$\begin{aligned} \mathrm{p}(\mathbf{x}|\mathbf{y}) &= \mathrm{p}(x_1|\mathbf{y})\mathrm{p}(x_2|x_1, \mathbf{y})...\mathrm{p}(x_T|\mathbf{x}_{1:T-1}\mathbf{y}) \\ &= \mathrm{p}(x_1|y_1)\mathrm{p}(x_2|y_2)...\mathrm{p}(x_T|y_T), \end{aligned} \tag{57}$$

and,

$$\begin{aligned} \mathrm{p}(\mathbf{y}) &= \mathrm{p}(y_1)\mathrm{p}(y_2|y_1)...\mathrm{p}(y_T|\mathbf{y}_{1:T-1}) \\ &= \mathrm{p}(y_1)\mathrm{p}(y_2|y_1)...\mathrm{p}(y_T|y_{T-1}), \end{aligned} \tag{58}$$

where $\mathrm{p}(y_1|y_0) = \mathrm{p}(y_1)$. Combining 57 and 58, we may rewrite the factorisation of the HMM as,

$$\mathrm{p}(\mathbf{x}, \mathbf{y}) = \prod_{t=1}^{T} \mathrm{p}(y_t|y_{t-1})\mathrm{p}(x_t|y_t). \tag{59}$$

The probabilities $\mathrm{p}(y_t|y_{t-1})$ are known as *transition* probabilities, and $\mathrm{p}(x_t|y_t)$ as *emission* probabilities. These probabilities constitute the model parameters, $\theta = (\mathbf{A}, \mathbf{B}, \mathbf{I})$, where $\mathbf{A}$ is the $|S| \times |S|$ matrix of probabilities of transitioning from one state to another, $\mathbf{B}$ is the $|S| \times |O|$ matrix of probabilities of emitting an observation given an underlying hidden state, and $\mathbf{I}$ is the vector of probabilities of initial states. The model parameters

must be precomputed[82]. Now, given a sequence of observations, $\mathbf{x}$, we may predict the hidden state sequence, $\mathbf{y}*$, by maximising the conditional distribution, $\mathrm{p}(\mathbf{y}|\mathbf{x})$. That is,

$$\mathbf{y}* = \underset{\mathbf{y}}{\operatorname{argmax}} \left\{ \prod_{t=1}^{T} \mathrm{p}(y_t|y_{t-1})\mathrm{p}(x_t|y_t) \right\}. \tag{60}$$

The hidden state sequence prediction is chosen to be the one maximising the likelihood over all possible hidden sequences. This seemingly intractable problem may be solved in polynomial time using dynamic programming (see Section 34.1.1).

34.1.1. *Viterbi Algorithm.* The Viterbi algorithm is used to efficiently compute the most likely sequence, $\mathbf{y}$, given an observation sequence, $\mathbf{x}$. The algorithm can do this efficiently by working along the sequence from state to state, and choosing the transitions that maximise the likelihood of the sequence fragment. To show this we define, $v_t(s) = \max_{\mathbf{y}_{1:t-1}} \mathrm{p}(\mathbf{y}_{1:t-1}, y_t = s|\mathbf{x})$, that is, the most likely sequence from the first $t-1$ states, with the choice of state s at time t. Thus, we may write,

$$\begin{aligned} v_t(s) &= \max_{\mathbf{y}_{1:t-1}} \mathrm{p}(\mathbf{y}_{1:t-1}|\mathbf{x})\mathrm{p}(y_{t-1}, y_t = s)\mathrm{p}(x_t|y_t = s) \\ &= \max_{\mathbf{y}_{1:t-1}} v_{t-1}(y_{t-1})\mathrm{p}(y_{t-1}, y_t = s)\mathrm{p}(x_t|y_t = s), \end{aligned} \tag{61}$$

and we may see the recursion. Once all states have been computed at time t, the maximum may be chosen and the algorithm proceeds to time $t+1$. The algorithm must test all $|S|$ transitions from the previous state to each of the $|S|$ current states, and it does that for each of the $|T|$ steps in the sequence. Hence, the complexity of the algorithm is a workable $\mathcal{O}(T|S|^2)$.

34.1.2. *Forward-backward Algorithm.* Another key inference algorithm to sequence learning is the forward-backward algorithm, so called for its computation of variables in both directions along the sequence. It is another example of a dynamic programming algorithm and is used to compute the so-called *forward-backward* variables, which are the conditional probabilities of the individual hidden states at each time step (that is, not the whole sequence), given the observation sequence and model parameters, namely, $\mathrm{p}(y_t = s|\mathbf{x}, \theta)$. These conditional probabilities have many useful applications, for example in the Baum-Welch algorithm for estimating model parameters, but also in the training of *conditional random fields*, as we discuss in Section 34.2. We may write the forward-backward variables as,

$$\gamma_t(s) = \mathrm{p}(y_t = s|\mathbf{x}, \theta) = \frac{\alpha_t(s)\beta_t(s)}{\sum_{s' \in S} \alpha_t(s')\beta_t(s')}, \tag{62}$$

[82]For example, the model parameters can be estimated through application of the Baum-Welch algorithm (**?**) on an unsupervised training set.

where the *forward* variables, $\alpha_t(s) = \mathrm{p}(\mathbf{x}_{t+1:n}|y_t = s, \mathbf{x}_{1:t}) = \mathrm{p}(\mathbf{x}_{t+1:n}|y_t = s)$, and the *backward* variables, $\beta_t(s) = \mathrm{p}(y_t = s, \mathbf{x}_{1:t})$. To derive the forward-backward algorithm we write, by the law of total probability,

$$\begin{aligned}\alpha_t(s) &= \sum_{y_{t-1}} \mathrm{p}(y_{t-1}, y_t = s, \mathbf{x}_{1:t}) \\ &= \sum_{y_{t-1}} \mathrm{p}(y_t = s|y_{t-1})\mathrm{p}(x_t|y_t)\mathrm{p}(y_{t-1}, \mathbf{x}_{1:t-1}) \\ &= \sum_{y_{t-1}} \mathbf{A}(y_{t-1}, s)\mathbf{B}(x_t, y_t)\alpha_{t-1}(y_{t-1})).\end{aligned} \tag{63}$$

Thus, we may see the recursion, as well as the way the forward variables will be computed, traversing the sequence in the forward direction with each forward variable of a given time a weighted product of those from the previous time. Likewise, for the backward variables, we may write,

$$\begin{aligned}\beta_t(s) &= \sum_{y_{t+1}} \mathrm{p}(y_t = s, y_{t+1}, \mathbf{x}_{t+1:n}) \\ &= \sum_{y_{t+1}} \mathrm{p}(\mathbf{x}_{t+2:n}|y_{t+1}, x_{t+1})\mathrm{p}(x_{t+1}, y_{t+1}|y_t = s) \\ &= \sum_{y_{t+1}} \beta_{t+1}(y_{t+1})\mathbf{A}(s, y_{t+1})\mathbf{B}(x_{t+1}, y_{t+1})).\end{aligned} \tag{64}$$

From Equations 63 and 64 comes the algorithm, which combines the results to compute the forward-backward variables, $\gamma_t(s)$. The complexity of the algorithm comes from noting that at each of the T steps in the sequence (in either direction), we compute $|S|$ variables, involving a summation of $|S|$ products. Hence, like the Viterbi algorithm, the complexity of the forward-backward algorithm is $\mathcal{O}(T|S|^2)$.

34.2. **Conditional Random Fields.** Conditional Random Fields (CRFs) are a machine learning technique for making structured predictions. They are an improvement to the similar, Maximum Entropy Markov models (MEMM) (**?**), which combine aspects of maximum entropy classifiers and hidden Markov models (**?**). They are a member of a class of structured sequence models called *random fields*, which are part of a broader family known as *graphical models*, including within it *Bayesian networks*.

Classification over relational data can benefit greatly from rich features, that is, describing observed attributes of an observation beyond merely its identity (as with HMMs). Take for example the context of text processing, where we might consider describing a string token (observation) by non-lexical features such as by its capitalisation or punctuation. Furthermore, we may wish to model context-aware features that contrast a string token with its surroundings. However, the complexity of the interdependencies of such features will likely make their explicit modelling infeasible. With CRFs, we circumvent this problem

by instead modelling the conditional distribution, $\mathrm{p}(\mathbf{y}|\mathbf{x})$, of the underlying graph structure, giving us free choice over features and, in so doing, *implicitly* defining a distribution over $\mathbf{x}$ without having to model this distribution directly (**?**). Such a conditional model is called a *discriminative* model, in contrast to a *generative* model, whereby the joint probability distribution is modelled explicitly. If we wish to model the interdependencies in a generative model, we must either extend the model which may both be difficult and entail intractable solution algorithms, or we must simplify the model and thereby compromise model performance. Notice that modelling the conditional distribution is sufficient for classification, where the observation sequence is known. This freedom for rich feature engineering is what makes CRFs the current state-of-the-art in metadata extraction, where arbitrarily defined features often make for good indicators. One may be tempted to use a logistic regression and classify each part of a sequence separately, but this would fail to take into account the contextual relations between the entities. For example, in the metadata extraction of a bibliographic reference, it is more likely for a publication title to follow an author list, and for a journal name to follow a publication title. This is what we mean by structured sequence learning, where the data to predict exhibits interdependencies and are correlated.

When the graph structure of a CRF model is the same as for a HMM (Figure 18), we have what is called a *linear-chain* CRF. HMMs and linear-chain CRFs thereby form what is called a generative-discriminative pair. In the general case, where the graph structure is more complex, we have what is called *skip-chain* CRFs. In this case the problem becomes far more complex, and we will not discuss these models here. A HMM may alternatively be expressed by the joint probability,

$$\mathrm{p}(\mathbf{x},\mathbf{y}) = \exp\left\{ \sum_{i,j\in S} \lambda_{ij} F_{i,j}(y_t, y_{t-1}, x_t) + \sum_{i\in S}\sum_{o\in O} \mu_{io} F_{i,o}(y_t, y_{t-1}, x_t) \right\}, \tag{65}$$

where the parameters λ_{ij} are the transition probabilities and μ_{ij} are the emission probabilities. $F_{i,j}(y_t, y_{t-1}, x_t) = \sum_t^T \mathbb{1}_{\{y_t=i\}}\mathbb{1}_{\{y_{t-1}=j\}}$ is a *feature function* used to activate the transition probabilities, and $F_{i,o}(y_t, y_{t-1}, x_t) = \sum_t^T \mathbb{1}_{\{y_t=i\}}\mathbb{1}_{\{x_i=o\}}$ for the emissions. The indicator functions activate the probabilities in accordance with the identity of the states and observations. Regardless, this formulation is equivalent to Equation 59. With some notational abuse we can define the more compact expression,

$$\mathrm{p}(\mathbf{x},\mathbf{y}) = \exp\left\{ \sum_{k} \lambda_k F_k(y_t, y_{t-1}, x_t) \right\}, \tag{66}$$

where F_k is a general feature function and λ_k a general feature weight. Now we may define the discriminative counterpart to this joint distribution, the linear-chain CRF,

$$\mathrm{p}(\mathbf{y}|\mathbf{x}) = \frac{\mathrm{p}(\mathbf{x},\mathbf{y})}{\sum_{y'}\mathrm{p}(\mathbf{x},\mathbf{y}')} = \frac{1}{Z(\mathbf{x})}\exp\left\{\sum_k \lambda_k F_k(y_t, y_{t-1}, x_t)\right\}, \tag{67}$$

where $Z(\mathbf{x}) = \sum_{y'}\exp\left\{\sum_k \lambda_{ij}F_k(y'_t, y'_{t-1}, x_t)\right\}$ is known as the partition function, ensuring probabilities sum to 1. Whereas HMMs model only the *occurrence* of a word, with conditional random fields we may choose F_k to define arbitrarily complex features, describing rich information about a word, its attributes, and its context. Finally, we may define a cost function in the following way,

$$l(\theta) = \sum_{n=1}^{N}\sum_{t=1}^{T}\sum_{k=1}^{K}\lambda_k F_k(y_t, y_{t-1}, x_t) - \sum_{n=1}^{N}\log Z(\mathbf{x}^{(n)}) - \sum_{k=1}^{K}\frac{\lambda_k^2}{2\sigma^2}, \tag{68}$$

where $\theta = \{\lambda_k\}_{k=1}^K$. This is called penalised maximum log likelihood. The penalty term, $\sum_{k=1}^K \frac{\lambda_k^2}{2\sigma^2}$, imposes an l_2 regularisation on the solution parameters, θ. However, according to (**?**), varying the tuning parameter, σ^2 (the variance), even by orders of magnitude has little effect on the outcome, a claim we corroborate in Chapter **??**. This cost function represents a strictly convex function, solvable using numerical methods such as L-BFGS (see Section **??**). Forward-backward processing (Section 34.1.2) is performed at each iteration to compute the partition function, as well as the conditional probabilities resulting from deriving the partial derivatives required for gradient descent. Hence, the complexity of training is $\mathcal{O}(INT|S|^2)$, where I is the number of iterations required and N is the number of training samples. Finally, the Viterbi algorithm (Section 34.1.1) is used to make a prediction with the trained model, that is, the best state sequence is found given the optimal parameter set found in training.

34.3. **Kalman Filters.** Kalman Filters, named after Hungarian mathematician Rudolf Emil Kàlmàn (1930-2016), is a method for improving estimates from sensor inputs in a time series. The Kalman filter predicts a true state as a compromise between a noisy observation and a prediction given the previous state, based by some physical rules. Suppose we had a GPS readout of a moving vehicle, and information about its direction and velocity. The GPS signal alone would be erratic compared to the reality of the vehicle's movement. We could therefore regularise it by factoring in expectations about the vehicle's position based on simple relations about displacement, velocity, and acceleration from physics. The corrected prediction might be a product of two Gaussians, one for the state prediction and one for the measurement, yielding a third Gaussian of lower variance. A real-time system can then be imagined where each state is fed back into itself.

35. Unsupervised Machine Learning

Whereas supervised machine learning can be said to embody learning by example, *unsupervised* machine learning is concerned with finding inherent patterns in a dataset. In

this sense, its aim is *description*, rather than prediction. The two most common applications of unsupervised machine learning are feature extraction and *clustering*. Clustering is of particular interest to data mining applications. Clustering is the act of grouping data observations according to some grouping measure. The choice of clustering algorithm is usually problem-specific. Various approaches exist, for example density-based clustering algorithm DBSCAN (density-based spatial clustering with additive noise), or hierarchical clustering models[83]. These techniques are popular, but are more heuristic algorithms than machine learning. The techniques we will discuss are more sophisticated and are based on fitting probability densities to data.

35.1. **K-means.** One of the most popular clustering techniques, K-means, organises data into K clusters. One commonly used algorithm is closely related to (and often called by) Lloyd's algorithm for Voronoi diagrams. This efficiently gives a clustering, albeit without a guarantee on global optimality, as K-means remains NP-complete. The algorithm begins by initialising K centre points in D-dimensional space and alternates between two steps. The first step assigns observations to centres based on a distance measure (usually Euclidean distance). Observations assigned to a common centre can then be said to be in the same cluster. For each cluster, a new centre point is calculated as the average of all observations in the cluster. This is repeated until the centre points converge (which will occur when observations cease to change clusters). Note that K must be pre-selected–the algorithm does not tell us what value of K is most suitable. To formalise beyond this intuitive algorithm, we denote a set of parameters, r_{nk}, whose binary value indicates the membership (or not) of observation x_n in cluster k. A further set of parameters, u_k, denote the centre point of each of the K clusters. Thus, the K-means algorithm can be formulated as the optimisation problem,

$$\min_{\mu,\mathbf{r}} \mathcal{L}(\mu, \mathbf{r}) = \sum_k \sum_n r_{nk} ||\mathbf{x_n} - \mu_k||_2^2,$$

where $\sum_k r_{nk} = 1$, and the distance measure is the sum of squares, $(x_n - \mu_k)^T \cdot (x_n - \mu_k)$. Now, we may write step 1:

$$r_{nk} = \begin{cases} 1 & \text{if } k = \arg\min_k \ ||\mathbf{x}_n - \mu_k||_2^2 \\ 0 & \text{otherwise} \end{cases}$$

Step 2 can then be seen as an optimisation of $\mathcal{L}$ with respect to each u_k. Clearly,

[83]Hierarchical models cluster data by pairing closest data samples into groups repeatedly, creating a binary tree hierarchy in a bottom-up approach until all data belongs to a single super group. Such a clustering is usually visualised with a tree-like dendrogram.

$$\frac{\partial \mathcal{L}}{\partial \mu_k} = \frac{\partial}{\partial \mu_k} \sum_n r_{nk}(\mathbf{x}_n^T \mathbf{x}_n - 2\mathbf{x}_n^T u_k + u_k^T u_k) = 0$$

$$\implies \hat{\mu}_k = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}},$$

this is clearly the arithmetic mean of the points in the cluster. With this in mind, we may see that our loss function can be written as a maximum likelihood,

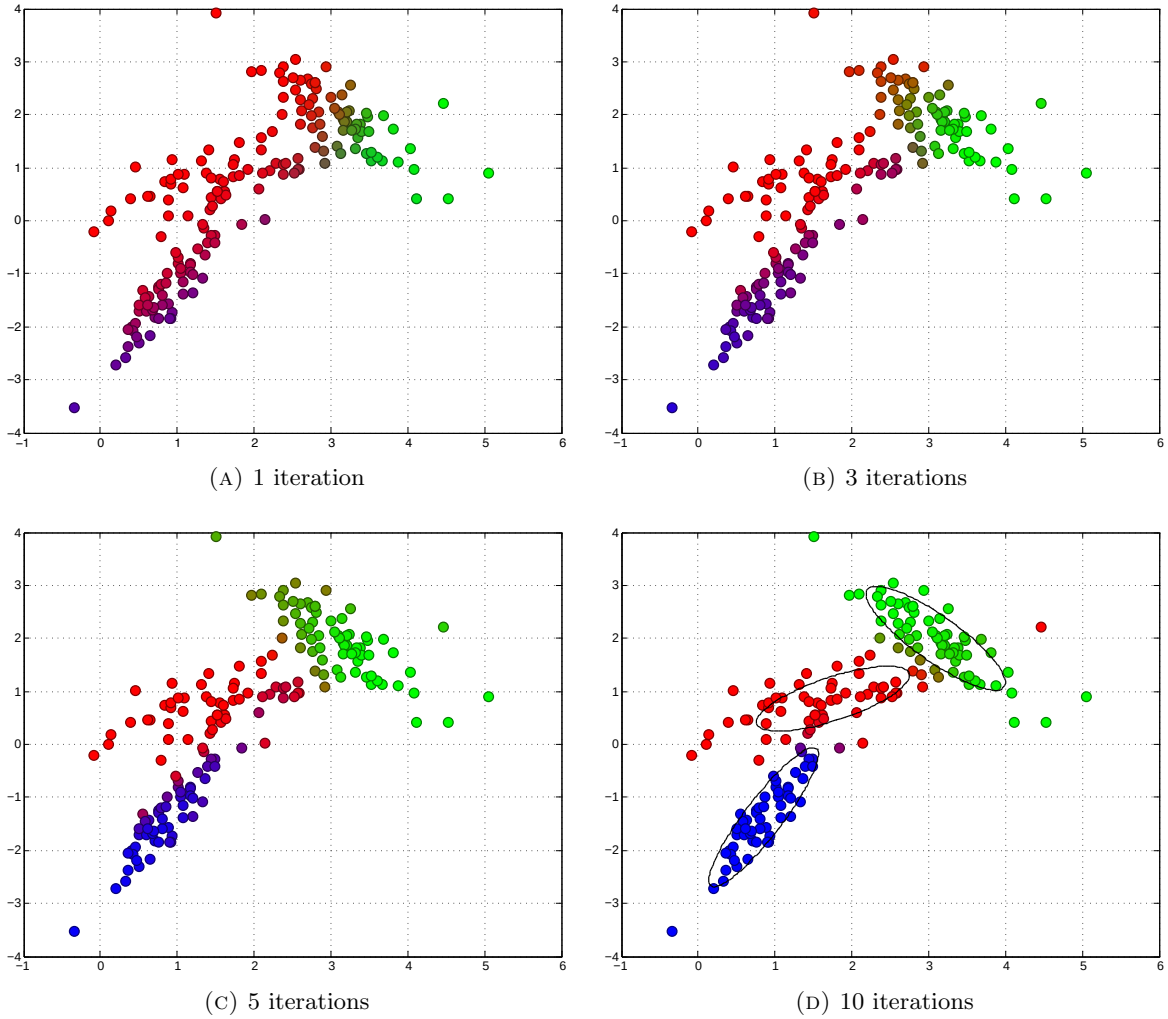
$$\min_{\mu,\mathbf{r}} \mathcal{L}(\mu, \mathbf{r}) = \max_{\mu,\mathbf{r}} \log p(\mathcal{D}|\mu, \mathbf{r}) = \log \prod_k \prod_n \mathcal{N}(\mathbf{x}_n; u_k, \mathbf{I})^{r_{nk}}$$

$$= -\sum_k \sum_n r_{nk}(\mathbf{x}_n - \mu_k)^T \mathbf{I}(\mathbf{x}_n - \mu_k),$$

where $\mathbf{I}$ is the identity matrix. This shows that K-means fits K Gaussian densities with unit variance. So, despite the initially simple algorithm, we see the deeper probabilistic meaning behind K-means. Because the clusters are all based on spherical Gaussians of fixed size and shape, the technique can alternatively be viewed as clustering points by nearness in Euclidean space. It is useful, however, to take note of the probabilistic notions, as these are expanded upon in the more powerful Gaussian mixture models.

35.2. **Gaussian Mixture Models.** A more powerful clustering technique, Gaussian mixture models (GMM) improve on two major shortcomings of K-means. Firstly, GMMs fit elliptical Gaussian densities, optimising over the choice of $\mathbf{\Sigma}$, the covariance matrix. Secondly, the parameters, r_{nk}, are replaced by random variables, r_n, of a special kind called *latent* random variables. These act as a bridge between each observation $\mathbf{x}_n$ (also a random variable), and its class, k. Thus, rather than hard binary values, the values $r_{nk} = p(r_n = k)$ indicate the probabilities of $\mathbf{x_n}$ belonging to cluster k. This is useful both for the fact that the parameters no longer grow with the size of the data, and also to quantify an uncertainty about $\mathbf{x}_n$, which can be used for analysis of outliers, and for example the selection of K. However, the complexity of the algorithm is far greater than K-means. Again, problem-specific considerations must be made when choosing the most suitable algorithm. Note that unlike spatial clustering algorithm DBSCAN, GMMs are capable of clustering densities that overlap, but are less useful when the data is not linearly separable. Thus, GMMs are good at clustering populations that are likely to be normally distributed in their features, but not, for example, geospatial data that exhibit irregular shapes. The likelihood function for GMMs is maximised using an iterative algorithm called expectation maximisation (EM) (Figure 19), which, similar to K-means, consists of two alternating steps, the E and M steps. In general this is,

$$\arg\max_\theta \mathbb{E}_{p(z_n)}[\log p(x_n, r_n|\theta)].$$



(A) 1 iteration

(B) 3 iterations

(C) 5 iterations

(D) 10 iterations

FIGURE 19. Expectation maximisation (EM) algorithm in action over 30 iterations for Gaussian mixture models with $K = 3$. Colour coding reflects the ratio of probabilities for each cluster–note that points in overlapping positions are more ambiguous. Contour plot of sample Gaussians given in final iteration. Created with `gmmDemo.m`.

36. Generative models

36.1. **Variational Autoencoders.** Autoencoding variational Bayes is about performing maximum likelihood estimation on the marginal probability distribution $p_\theta(\mathbf{x})$ for data sample $\mathbf{x}$, where it is supposed that latent variables $\mathbf{z}$ generate $\mathbf{x}$. The posterior distribution is defined as $q_\phi(\mathbf{z}|\mathbf{x})$. Applying Bayes theorem,

$$
\begin{aligned}
\log p_\theta(\mathbf{x}^{(i)}) &= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\big[\log p_\theta(\mathbf{x}^{(i)})\big] \\
&= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log \frac{p_\theta(\mathbf{x}^{(i)}|\mathbf{z})p_\theta(\mathbf{z})}{p_\theta(\mathbf{z}|\mathbf{x}^{(i)})}\right] \\
&= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log \frac{p_\theta(\mathbf{x}^{(i)}|\mathbf{z})p_\theta(\mathbf{z})}{p_\theta(\mathbf{z}|\mathbf{x}^{(i)})} \cdot \frac{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}\right] \\
&= \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log \frac{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}{p_\theta(\mathbf{z}|\mathbf{x}^{(i)})}\right] + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\left[\log \frac{p_\theta(\mathbf{x}^{(i)},\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x}^{(i)})}\right] \\
&= \underbrace{D_{KL}\big(q_\phi(\mathbf{z}|\mathbf{x}^{(i)})||p_\theta(\mathbf{z}|\mathbf{x}^{(i)})\big)}_{\geq 0} + \mathcal{L}(\theta,\phi;\mathbf{x}^{(i)}),
\end{aligned}
$$

and the divergence term may be eliminated, giving us what is known as a *variational* lower bound on the likelihood, and which can nevertheless be maximised. This expression can then be rearranged as,

$$
\mathcal{L}(\theta,\phi;\mathbf{x}^{(i)}) = -D_{KL}\big(q_\phi(\mathbf{z}|\mathbf{x}^{(i)})||p_\theta(\mathbf{z})\big) + \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}\big[\log p_\theta(\mathbf{x}^{(i)}|\mathbf{z})\big],
$$

that is, yet another KL divergence term (that can in many cases by addressed analytically), and an expectation. It is well worth pointing out here that when this formulation is instantiated as a vartiational autoencoder, the distribution $q_\phi(\mathbf{z}|\mathbf{x})$ is the encoder model, and $p_\theta(\mathbf{x}^{(i)}|\mathbf{z})$ is the decoder model. Because it is the distribution over which the expectation is defined, q_ϕ will be sampled according to a Monte Carlo strategy, which is why one samples the encoder model via the reparameterisation trick. Both models produce the parameters (mean and variance) for their respective distributions. There is nothing unusual about this. The Bayesian view on linear regression sees a linear model producing the expectation of a Normal distribution around the target variable (see Section 26.6).

Now, we would like to continue with MLE by computing gradients and performing gradient ascent. The expectation is the problem, as it would require evaluating an intractable interval over the distribution $q_\phi(\mathbf{z}|\mathbf{x})$. To bypass this, Monte Carlo techniques are used, but these suffer from high variance. This is where the main invention comes in: the *stochastic gradient variational bayes* (SGVB) estimator replaces the need for sampling from the posterior directly by instead predicting its distributional parameters and combining them with sampled Gaussian noise in a reparameterisation,

To remedy this, the authors come up with a novelty called the *reparameterisation trick*. This consists of replacing the random variable $\mathbf{z}$ with a deterministic function of $\mathbf{x}$ and a random noise variable $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$ such that,

$$
\mathbf{z} = g(\mathbf{x}, \boldsymbol{\epsilon})
$$

The function $g(\mathbf{x}, \boldsymbol{\epsilon})$ is chosen to be differentiable (in our case an *encoding* network). Now, we have the *stochastic gradient variational Bayes* (SGVB) estimator,

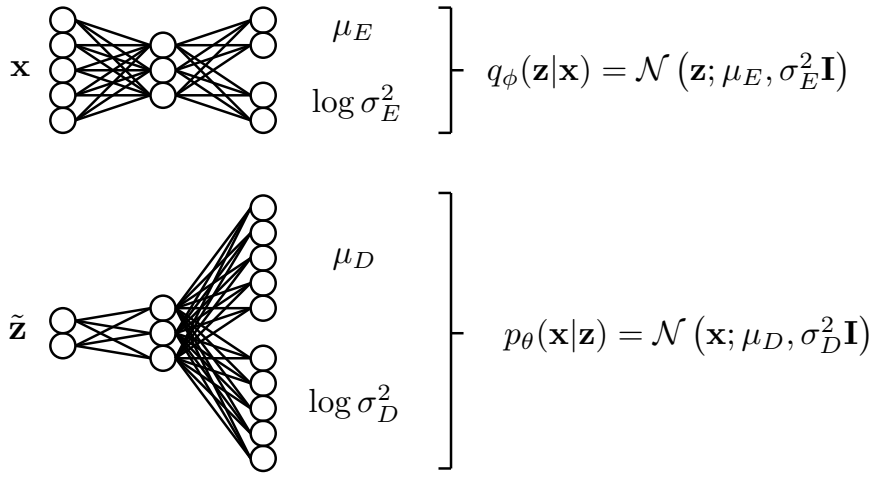


FIGURE 20. Recognition and generative networks of a variational autoencoder.

$$\widetilde{\mathcal{L}}(\theta, \phi; \mathbf{x}^{(i)}) = \underbrace{-D_{KL}\left(q_\phi(\mathbf{z}|\mathbf{x}^{(i)})||p_\theta(\mathbf{z})\right)}_{\text{analytic solution}} + \underbrace{\frac{1}{L}\sum_{l=1}^{L} \log p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,l)})}_{\text{Monte Carlo approximation}}$$

The function g of the reparameterisation embodies the sampling of the recognition model $q_\phi(\mathbf{z}|\mathbf{x})$ i.e. the posterior to compute the expectation. Thus, it is used for Monte Carlo estimation (which it turns out is stable for a single sample i.e. $L = 1$) of the likelihood $p_\theta(\mathbf{x}|\mathbf{z})$, which is the probability of the sample given the $\mathbf{z}$ sampled from the posterior (under the reparamterisation trick). The eponymous AEVB algorithm is simply all of this sampling wrapped in a gradient descent loop.

The nice thing about this framework is we can choose what the distributions are. One problem instance of all this is the variational autoencoder, where the prior $p_\theta(\mathbf{z}$ is unit Gaussian, and the posterior $q_\phi(\mathbf{z}|\mathbf{x}^{(i)}$ is diagonal. The KL divergence terms therefore ensures the posterior, as predicted by the recognition network, is close to the prior. Thus, the likelihood model becomes generative and can be sampled from at test time. The auxiliary noise variable of the reparameterisation follows a unit Normal distribution $p(\boldsymbol{\epsilon})$, which is sampled and combined with the predicted parameters of the recogition model to sample the posterior. The likelihood distribution $p_\theta(\mathbf{x}|\mathbf{z})$ is a Gaussian whose parameters are embodied by another neural network. Thus, applying MLE to the training data consists of predicting the mean of a Gaussian, just like in linear regression. The mean is the prediction for the target $\mathbf{x}$, and with the log probability this creates a MSE, and magically we have an autoencoder.

36.2. **Generative adversarial networks.** Generative adversarial networks (GANs) is a learning framework that trains a generator model to be capable of sampling new data from

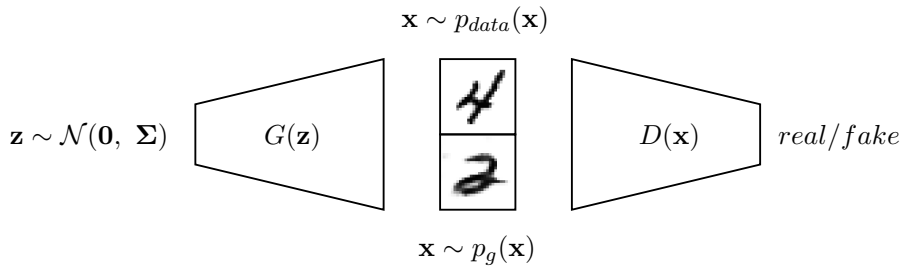


FIGURE 21. A GAN trains a generator network G to fool a discriminator network D in emitting counterfeit images $\mathbf{x} \sim p_g(\mathbf{x})$ by transforming noise input $\mathbf{z}$. In order to do so, G must (implicitly) learn the data-generating distribution, p_{data}.

a data distribution (in particular, images), learnt (implicitly) from a training set. GANs compete with variational autoencoders and, more recently, autoregressive models for image generation. GANs are arguably a more mature and versatile learning paradigm, however.

The GAN framework (Figure 21) is *adversarial* in that it pits a pair of neural networks in a two-player minimax *game*: a *generator*,

$$G : Z \to X, \tag{69}$$

where in the simplest case $\mathbf{z} \sim Z$ is a vector of (uniform or Gaussian) noise and $\mathbf{x} \sim X$ is a synthetic data point (in particular, an image); and a *discriminator*,

$$D : X \to \{0, 1\}, \tag{70}$$

that is, a mapping from data point to binary value. D aims to minimise its rate of failure in discerning true data from "fake" data sampled from the the generator, which aims to maximise the error made by the discriminator,

$$\max_G \min_D \mathcal{L}_{GAN}(D, G) = \mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]. \tag{71}$$

The stationary point of Equation 71 is a *Nash equilibrium* (a saddle point) between the two objectives. It is guaranteed that the distribution implicitly defined[84] by G, p_g is equal to p_{data} for G^* at (global) optimality and,

$$D^*(\mathbf{x}) = \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_g(\mathbf{x})}, \tag{72}$$

[84]GANs are implicit generative models, in contrast to explicit models such as VAEs, which model the probabilities directly. GANs serve as an sample-emitting oracle emulating the true distribution.

where p_{data} is the data-generating distribution. In practice, GANs are trained by back-propagation with alternating gradient descent between D and G. The weights of D are frozen while propagating the errors back through D to G.

GAN architecture design must strike a balance between a generator expressive enough to approximate the data distribution, and a discriminator powerful enough to hold it to the task. GANs are afflicted with a phenomenon known as *mode collapse*. Mode collapse is the condition in which the generator "collapses" to generating few or even a unique output, regardless of the noise input $\mathbf{z}$. Two varieties of mode collapse exist: *discrete mode collapse*, where the generator collapses to a subset of data modes (easily detected); and the more insidious *manifold collapse*, where the generator collapses to a subspace of the data distribution. They reason that mode collapse originates from the fact that in each iteration of training, the generator is impelled towards a delta function at the mode considered most "real" by the discriminator. The discriminator responds by lowering the probability of this mode, leading to oscillations in training. The "unrolled" GAN training approach that they advocate mitigates this tendency by informing the generator in advance of the discriminator's response to its prospective weight updates.

Another promising attempt to attenuate the mode collapse problem is to replace the Jensen-Shannon divergence loss function with the Earth Mover (EM) or Wasserstein-1 distance. Such is the strategy of Wasserstein GANs (WGANs). Intuitively, the EM distance measures a distance between two distributions. One can write the EM distance as the quantity $\mathbb{E}_{x\sim\mathbb{P}_r}[f_w(x)] - \mathbb{E}_{z\sim p(z)}[f_w(g_\theta(z))]$ maximised by choice of f_w (discriminator). The goal is then to minimise this distance through optimisation of the generator. The training algorithm is very similar to GANs. WGANs seem to be more stable, as the substituted loss function allows the discriminator to be trained to optimality. This greatly mitigates the mode collapse problem. Nevertheless, more subtle forms of partial mode collapse remain a recurrent problem for GANs. As a bonus, the discriminator loss (estimated EM distance) becomes meaningful during training, interpretable as image sample quality, rendering the implementation and fine-tuning of WGANs far easier.

36.2.1. *Deep convolutional GANs.* DCGANs first succeeded in training more powerful, high-resolution GANs, by identifying a set of design principles: replacing pooling layers with strided convolutions; removing fully-connected layers (apart from the first layer of the generator); using batch normalisation after all but the last layer; using a tanh activation at the end of the generator (and ReLU everywhere else); and using LeakyReLU activations everywhere in the discriminator. For image data, deep convolutional GANs (DCGANs) represent a vast improvement over the original fully-connected GANs.

36.2.2. *Conditional GANs.* Conditional GANs present a simple extension to GANs allowing for control over the data generating process. One simply includes an additional input $\mathbf{y} \sim Y$ (for example, class information), to obtain conditional GANs (CGANs). That is, now the generator is conditional,

$$G : Z, Y \to X, \tag{73}$$

as is the discriminator,

$$G : X, Y \to \{0, 1\}, \tag{74}$$

and the CGAN is trained against the objective,

$$\mathcal{L}_{CGAN}(D, G) = \mathbb{E}_{\mathbf{x}\sim p(\mathbf{x})}[\log D(\mathbf{x}|\mathbf{y})] + \mathbb{E}_{\mathbf{z}\sim p(\mathbf{z})}[\log(1 - D(G(\mathbf{z}|\mathbf{y})))]. \tag{75}$$

The intuition as to why conditional GANs works can be an instructive thing to consider. As D learns to associate conditional information such as object class with the ground truth images, it will declare "fake" any image emitted by G not conforming to the condition, no matter the quality of the image. If G is to succeed, it will be forced to generate images in adherence to the condition. The outcome of successful training is therefore a generator that can, for example, synthesise an authentic-looking image of a desired class on demand.

The implementation of CGANs is a straightforward modification to GANs. The conditional information is concatenated to the standard inputs of both D and G and fed through to their hidden layers. A DCGAN discriminator may, however, concatenate the conditional information after its convolutional layers.

36.2.3. *Assorted GANs.* The success of GANs have led to many interesting results in recent years. Image-to-image translation has been achieved with `pix2pix`, a GAN conditioned on a full image serving as a blueprint for the generated image. Such models are capable of an endless variety of applications, including segmentation and image colourisation. CycleGANs achieve *unpaired* image-to-image translation, learning to map between image domains in an unsupervised way. Applications include style transfer. RecycleGANs achieve the same for video-to-video retargeting.

Elsewhere, progressive growing of GANs produce dazzling high resolution outputs (up to 1024×1024 pixels). Here, training is done in stages, beginning at low resolutions (4×4), and doubling the resolution at intervals. This is done by dynamically adding new layers to the generator and discriminator, which maintain symmetry throughout training. Thus, the GAN learns global structures first at low resolution and progressively refining its output.

37. Performance Metrics

Accuracy is defined to be,

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

that is, the proportion of correct classifications to total classifications, where TP is the number of *true positives*, the number of times a class is correctly predicted to have occurred; TN is the number of *true negatives*, the number of times a given class is correctly predicted not to have occurred; FN is the number of *false negatives*, the number of times a class is incorrectly predicted to have occurred; and FN is the number of *false negatives*, the number of times a class is incorrectly predicted not to have occurred. Accuracy can be a misleading statistic when we have uneven representations of classes in the dataset. In

the event that we have a sufficiently high bias, we can achieve excellent accuracy simply by always predicting the dominant class. For this reason, we consider other statistics too. *Precision* is the number of times a class is *correctly* predicted proportional to the overall number of predictions for that class, that is,

$$\text{Precision} = \frac{TP}{TP + FP}.$$

This, however, does not inform us as to whether we have missed any occurrences of the class, which would be shown in the number of false negatives, FN. We could therefore have a very high precision with limited accuracy. *Recall* is the number of times a class is *correctly* predicted proportional to the number of occurrences of that class (equivalently, the accuracy with respect to the class), that is,

$$\text{Recall} = \frac{TP}{TP + FN}.$$

However, a simple strategy of always predicting one class will give perfect recall for that class, because then misclassifications are only captured by FP. The F_1 statistic is a common measure used to assess classifiers that combines precision and recall, and is defined as,

$$F_1 = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}},$$

that is, the harmonic mean of precision and recall (the "1" in F_1 indicates the two are evenly weighted). The F_1 statistic is a neat way of summarising both metrics at once. Furthermore, a large imbalance in precision and recall results in a lower F_1 score. It is necessary to be good in both precision and recall to have a good F_1 score; the harmonic mean of any data is always upper-bounded by its arithmetic mean. Thus, the F_1 score addresses their shortcomings simultaneously.

37.1. **ROC Curves.** Receiver Operator Characteristic (ROC) curves are a way of assessing the performance of a binary classifier, usually visualised graphically. A ROC curve compares the true positive and false positive rates of a binary classifier over a range of thresholding choices. In practice, a classifier's predictions are output as a set of real probabilities (taking care not to round early). Points on the ROC curve are generated by calculating the resultant TP and FP rates[85] as the classification threshold is varied from 0 to 1. To illustrate, consider a low threshold: a good classifier will tend be more sure about its negatives, avoiding false negatives, even at lower thresholds. This will place the curve high on the top left corner of the graph. Over the spectrum of thresholds, a good classifier will maintain the ROC curve high above the FP-axis. The area under the ROC curve (AUROC) $\in [0, 1]$ is therefore a good indicator of the strength of a binary classifier.

[85]Note that some authors prefer to refer to the true positive rate as *sensitivity* and the false positive rate as the complement of *specificity*. Specificity is in fact equivalent to the *true negative* rate, and is therefore complement to the false positive rate. For this reason, when these alternative terms are used, the specificity axis runs backwards on the ROC graph.

Further note that because *rates* are used rather than absolute positives and negatives, ROC curves address the same shortcoming the accuracy measure has for unbalanced data sets.

38. Image Analysis and Processing

Image analysis and image processing comprise of an amalgam of techniques for respectively analysing and transforming images. Image analysis calculates image characteristics, while image processing produces a new, modified image. Images are here primarily considered as two-dimensional arrays of pixels, although three-dimensional *voxel* images

38.1. **Connectivity.** In any region-growing algorithm on an image, one must first define the connectivity of pixels. That is, for each given pixel, to define its spatial neighbourhood. For a *four-connected* pixel at coordinate (x, y), its neighbourhood is defined as the four adjacent pixels with which it shares an edge. This is also known as the *Von Neumann neighbourhood*. In *eight-connectivity*, the neighbourhood of a pixel are all those pixels with which it shares a corner.

In either of these two schemes, a logical contradiction arises. Consider two image regions, sharing a single corner. In four-connectivity, the regions are not connected, yet neither is the background between them–a contradiction. In eight-connectivity, the regions are connected, yet the background also–a contradiction. This prompts the use of *six-connectivity*. In six-connectivity, pixels are conceptualised as being hexagonal, and alternating rows of pixels are virtually shifted by a half-pixel to give each (non-border) pixel six neighbours, with which it shares a single unique edge. This overcomes the paradoxes of four- and eight-connectivity, albeit with a computational burden of maintaining a virtual hexagonal grid.

38.2. **Image Scaling.** Image scaling or resampling is a form of image processing referring to the resizing of an image to a smaller or larger size. One begins with the two-dimensional grid of points constituting the original image. A new grid (with fewer or more points depending on the scale factor) is overlaid onto the original coordinate system. Each unknown point is approximated by some combination of the unit grid of four original image points around it. Note that there will be some unit grids containing no unknown points, or some with many, depending on the scale factor.

38.2.1. *2D Nearest Neighbour Interpolation.* In a 2D nearest neighbour scheme, the intensity of the unknown point is assigned simply as the intensity of the closet point in the surrounding unit grid. This is one of the cheapest, yet imprecise approaches, typically yielding pixelated results.

38.2.2. *Bilinear Interpolation.* In bilinear interpolation, a point (x, y) within the unit grid, $\{(x_1, y_1), (x_2, y_1), (x_1, y_2), (x_1, y_2)\}$ is interpolated as the linear interpolation in the y direction of the points, (x, y_1) and (x, y_2), obtained by linear interpolation of $(x_1, y_1), (x_2, y_1)$ and $(x_1, y_2), (x_1, y_2)$ respectively. This is equivalent to solving $f(x, y) = a_0 + a_1x + a_2y + a_3xy$ for the four unknowns a_i (fully-determined).

38.2.3. *Bicubic Interpolation.* Bicubic interpolation is similar to bilinear interpolation, but where we solve for the sixteen unknowns of $\sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij}x^i y^j$. Because we still only have the four points of the surrounding unit grid. However, we suppose have access to the x and y first derivatives at each point, and the xy second derivative at point. This gives us 16 equations, hence we are able to solve for all unknowns. Bicubic interpolation tends to give superior results, as it embodies prior knowledge of the typical smoothness of the gradients of an image. This performance does come at a higher computational cost, however.

38.3. **Image Compression.**

38.4. **Labeling.** In image analysis, labelling refers to the identification (via annotation) of the connected components in an image. If we conceptualise a binary image $\mathcal{I}$ as a graph where each pixel is a node and the connectivity of nodes are given by both adjacency and equal intensity (for grayscale or RGB images this could be relaxed to approximate equal intensity). A rough pseudocode is given in Algorithm 4.

Algorithm 4 Pseudocode for labelling connected components in an image.

```
label ← 0
queue ← sorted(vertices) // sorted top-to-bottom and left-to-right
procedure PROPAGATE(vertex, label)
    vertex.set_label(label)
    for connected_vertex in vertex.neighbourhood() do
        if not queue.contains(connected_vertex) then
            continue
        else
            propagate(connected_vertex, label)
while not queue.empty() do
    vertex ← queue.pop()
    propagate(vertex, label)
    label ← label +1
```

38.5. **Thresholding.** Thresholding can be considered to be a simple form of segmentation. Typically, thresholding results in a binary image (demarcating foreground and background), though several thresholds may be set for an n-ary segmentation.

38.5.1. *Otsu Thresholding.* Otsu thresholding is an image analysis technique that optimises the placement of a threshold k in a histogram. Noting that the technique was motivated for image segmentation, it is general-purpose, non-parametric approach to thresholding (a form of one-dimensional clustering), it may be used for any univariate feature readout. This *discriminant* idea is to maximise the ratio $\lambda = \sigma_B^2/\sigma_W^2$ of the *intra*-class variance[86], σ_W^2 and the *inter*-class variance[87], σ_B^2, creating a threshold that maximises similarity within

[86]Intra-class variance is the weighted sum of the respective variances of classes invoked by a choice of k

[87]Inter-class variance is merely the average squared distance of the class means from the global mean

the respective clusters, while maximising the distance between them. The problem is simplified, however, by noting the relation $\sigma_B^2 + \sigma_W^2 = \sigma_T^2$, where σ_T^2 is the (fixed) global variance. It can then be shown that maximising λ is the same as maximising $\eta = \sigma_B^2/\sigma_T^2$, as they are a function of one another (the problem has only one degree of freedom, not two). Since σ_T^2 is a constant we can optimise the threshold merely by maximising σ_B^2. So, the algorithm proceeds by varying k and calculating the inter-class variance each time, retaining the largest.

38.5.2. *Adaptive Thresholding.* If there is a steady gradient (for example, a shadow) in the image, a single global threshold will often be ineffective. Whereas in the Otsu framework a single scalar threshold is computed, adaptive thresholding algorithms compute a *threshold image*, to which the original image is compared element-wise to determine the binary mask output. This threshold image consists of local thresholds calculated in a fixed neighbourhood around each pixel, for example, the weighted average or Otsu threshold calculated inside a structuring element. The superior segmentation result will come at the cost of a (much) greater runtime.

38.6. **Linear Filters.** In image processing, A convolution kernel is a (typically) small matrix[88] $\mathbf{K}$ that is convolved with an image $\mathbf{I}$. A convolution in two dimensions is defined as,

$$(\mathbf{I} * \mathbf{K})(x, y) = \sum_{i=1}^{d} \sum_{j=1}^{d} \mathbf{I}(x + i - d/2, y + j - d/2) \times \mathbf{K}(i, j)$$

38.6.1. *Low-pass filters.* Low-pass filters are a means for removing noise, or smoothing, which can be thought of as a form of image regularisation. A popular choice is the *Gaussian filter*, whose kernel weights follow a Gaussian distribution, circular in two dimensions. A common kernel size is 5×5 with,

$$\mathbf{K}_{\text{blur}} = \frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

38.6.2. *High-pass filters.* High-pass filters can be used for edge detection. This is usually based on measuring the gradient at all points in the image. Gradients tend to be high when passing from one intensity region to another. To calculate the gradients, we use some variant of finite differences. Such a kernel could be,

[88] It is, however, possible to define non-square convolutional kernels, including single column or row, one-dimensional kernels.

$$\mathbf{K}_{\text{edge}} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{bmatrix},$$

that is, the sum of the second order central difference operators. This is known as the Laplacian filter. Note the linearity property of convolutions. As an aside, the *Canny edge detector*, named for Australian computer scientist John F. Canny (1958-), is a *pipeline* incorporating various filters. To sharpen an image, we can subtract its edges, darkening the contours. This can be achieved by combining an identity kernel and an edge detection kernel,

$$\mathbf{K}_{\text{sharpen}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}.$$

Sobel filters are another edge detection approach,

$$G_x = I * \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, G_y = I * \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}.$$

These are combined to obtain the gradient magnitude image,

$$G = \sqrt{G_x^2 + G_y^2}.$$

38.7. **Non-linear Filters.** Non-linear filters are any filter not producing a linear combination of its receptive field. They tend to be designed for noise removal. The three examined here are examples of *rank* filters, where the received intensities are sorted and a particular intensity is chosen.

38.7.1. *Median filtering.* Median filtering is a simple, robust technique for removing noise from a signal. Each pixel of an image is replaced with the median of the surrounding area. The median (rather than the mean, which can be), is robust to isolated, high intensity noisy pixels. As outliers in most pixel neighbourhood, they will usually be replaced and will usually not interfere with the replacement of non-noise. Depending on the image and the amount of noise, there is a tradeoff to be had–moderated by the chosen size of the neighbourhood–between removal of noise and loss of detail.

38.7.2. *Minimum and Maximum filtering.* Minimum and maximum filtering work in a similar way to median filtering and are the basis of image dilation and erosion respectively.

38.8. **Mathematical Morphology.** Mathematical theory is a theory for image processing inspired by geosciences, in which an image is conceptualised a three-dimensional terrain. It formalises image manipulation operations in terms of set operations. The primitive operations of dilation and erosion, and their composites, opening and closing, involve an image A, defined as a set for binary images and a function for grayscale images, on a space E, and a *structuring element* B, likewise defined, used to carry out the operation.

38.8.1. *Dilations.* A dilation is roughly an enlargement of an image, similar to a minimum filter. In the binary setting, a dilation produces a new image,

$$A \oplus B = \bigcup_{b \in B} A_b$$

where A_b is the image A displaced by the value b. One can therefore think of B as a set of *displacements*. The new image therefore consists of all points in the displacement range of B from some point in A.

On a greyscale image, A and B become functions, corresponding to different intensity values over a set of points. Note that now A and B are defined on the whole domain E, though they may be non-zero for only a part of it. In this case, a dilation is defined as,

$$(A \oplus B)(x) = \sup_{y \in E}[A(y) - B(x - y)]$$

That is, the dilated image at a point x is the largest value arising from the difference in A and B over all displacements of B (defined on the whole plane E). It is typical, however, to use a "flat" B that is 0 in a subset of E and $-\infty$ everywhere else. Thus defined, B becomes a set of displacements again, and the greyscale dilation is very similar to the binary version.

38.8.2. *Erosions.* An erosion is the dual operator of dilation, corresponding to a shrinking of an image, and similar to a maximum filter. In the binary setting we have,

$$A \ominus B = \bigcap_{b \in B} A_{-b}$$

that is, the eroded image consists of all pixels in A that remain in A under all (negative) displacements of B. Equivalently, it is the set of all pixels in A that intersect with B for all all possible displacements of B. The greyscale variant is defined as,

$$(A \ominus B)(x) = \inf_{y \in B}[A(x + y) - B(y)]$$

38.8.3. *Openings.* An opening is an important operation combining the two above primitives. It consists of an erosion followed by a dilation,

$$A \circ B = (A \ominus B) \oplus B$$

Openings are used for noise removal, as small elements are eliminated by the initial erosion, and everything else is approximately restored by the final dilation.

38.8.4. *Closings.* A closing consists of a dilation followed by an erosion,

$$A \bullet B = (A \oplus B) \ominus B$$

and serves to eliminate holes in surfaces.

38.8.5. *Gradient.* The morphological version of a gradient image can be calculated as the difference between a dilation and an erosion,

$$g(A) = (A \oplus B) - (A \ominus B)$$

38.8.6. *Reconstruction.* A morphological reconstruction is a more sophisticated operation involving both the original image, and its binary mask.

38.8.7. *Top Hat.*

38.8.8. *Diameter Openings. diameter openings*, based on a flooding technique,

$$\begin{aligned}[\gamma^{\circ}_{\lambda}(f)](x) =& \sup\{s \leq f(x) \mid \alpha(C_x[X^{+}_{s}(f)]) \geq \lambda\} \\ =& \bigcup\{\gamma_B(f) \mid \alpha(B) \geq \lambda\}\end{aligned}$$

where $X^{+}_{s}(f)$ is the set of all pixels with $f(x) \geq s$, $C_x[A]$ is the connected component of set A containing point x and $\alpha(C_x)$ its maximal extension.

Simply put, this is a *prominence-finding* algorithm, which simulates a flooding on the (inverted) image and terminates when the *maximal extension* (the longest axis of a possibly non-convex shape) has a length of at least λ. Diagrammatically, this is easy to see as isolating the regions that rise sharply (instead of gradually) to a significant intensity, that is, within a small area (diameter). There are two parameters to diameter openings: λ, the diameter; and t (not formalised), the threshold. In an unnecessary flourish, we represent this technique in two alternative formalisms:

(1) The revolting, practical way: as the supremum (union in the image algebraic lattice) of the intensities $f(x)$ on pixels x greater than or equal to s (is this part superfluous?) where the connected component C_x on pixels having intensities greater than or equal to s, $X^{+}_{s}(f)$, to which x belongs has a maximal extension of at least λ
(2) The elegant, impractical way: as the union of all closures (erosion followed by dilation) of the image intensities f, where the structuring element of the closure is at least λ. Recall a closure eliminates small elements first with an erosion and restores surviving elements with a dilation. The "union of all" implies that the lower intensities (which must have a wider diameter than the higher intensities they *subsume*[89]) are preserved also (by some larger structuring element). What are lost in this operation are the pointy parts (which are then recovered by subtraction).

[89]Remember, images $f(x)$ are surjective. That is, each x maps to a single intensity. This means that the image "terrain" may only go up or down (or lie flat); it cannot loop back on itself, a characteristic that would render morphological analysis impossible.

In either case (they are equivalent), we then "build" a difference to the original image.

38.8.9. *Granulometries.* Another approach based on *granulometries* is,

$$\sum_{x \in S_k} \gamma_{B_i} f(x) - \gamma_{B_{i+1}} f(x)$$

where the structuring elements B_i fulfill the condition $B_i \subset B_{i+1}$ and S_k is the k-th cell.

This algorithm is again based on openings (erosion $\rightarrow$ dilation), with this time a *series* of successively *larger* structuring elements, B_i. That is, B_{i+1} contains B_i. The difference of each consecutive pair is summed over the whole cell ($x \in S_k$). A larger difference therefore indicates a sharp rise from one level to another. Intuitively, this is again capturing the prominence of the region. It is not well formulated in the paper, but the indices i in B_i imply that we create granulometries at multiple levels (no free lunch–we must try them all).

38.9. **Segmentation.** Segmentation consists of partitioning an image into homogeneous regions. It is a difficult problem and the scope varies from low-level pixel grouping *super-pixels* to high-level semantic segmentation (segmentation of meaningful objects), the latter task best accomplished by deep neural networks.

38.9.1. *The Watershed Algorithm.* The Watershed algorithm is a popular segmentation algorithm from the morphology that simulates a "flooding" or "immersion" on the image topology. The flooding begins from each of a set of user-defined *markers*, defining respective *catchment zones*, and terminates when each catchment zone overflows into an adjacent one. The overflow boundary between catchment zones defines a *watershed line*. The weakness of the algorithm arises from the arbitrary placement of markers, which may arise in under- or over-segmentation, and must is user-defined and application-specific. In many application, substantial pre-processing is done on an image to transform it into an amenable form for the algorithm. The watershed algorithm is usually implemented with hierarchical queues.

38.9.2. *Superpixels.* Superpixels result in an intermediate, low-level segmentation, where pixels are grouped into homogeneous sets, crucially respecting object contours within an image. Superpixels are intended for use in various computer vision tasks, including as the basis of a higher-level segmentation of an image and feature extraction for classification.

The authors show that their Simple Linear Iterative Clustering (SLIC) algorithm achieves efficient state-of-the-art results. The technique works by performing k-Means on the 5-tuple of position (x, y) and *CIELAB* colour[90], but instead of over the entire image, over a spatial grid only. Note that since the colour and space coordinates are of different scales, some simple regularisation is introduced to ensure the clustering. There is a simple preprocessing step of repositioning cluster seeds initialised on image contours, and a final post-processing step reassigning any disjoint cluster components. The simplicity of SLIC seems to reinforce the intuition that forming super-pixels is a relatively simple problem.

[90]CIELAB by the International Commission on Illumination encode lightness, L, (from black to white), relative green-magenta intensity, denoted by A, and relative yellow-blue intensity, denoted by B

38.10. **Feature Extraction.** Features are vectorised quantities extracted from images. They so that imagine

38.10.1. *Local binary patterns.* Local binary patterns (LBP) characterise image texture by quantifying uniformity. An LBP algorithm divides an image up into square cells. For each pixel within a cell, the pixel is compared with pixels in its neighbourhood. Where the pixel's intensity is greater than its neighbour's, a 1 is recorded, otherwise a 0. This produces a binary string, which is an 8-bit integer under 8-connectivity. The histogram of all binary strings in the cell are the LBP features. LBP features from other cells are concatenated to produce a large feature vector.

38.10.2. *Histogram of Oriented Gradients.* In histogram of oriented gradient (HOG) features, the direction of the gradient at each pixel is computed (i.e. the angle of the sum of central difference gradients computed in the x- and y-directions). The directions are summarised as orientation histograms over square groups of pixels called cells, producing a local representation as a vector of proportions (they choose 9 per cell). The "vote" of each pixel gradient in the histogram is weighted by its magnitude. To account for local variation in illumination, the cell values are normalised over a wider multi-cellular area called a block. They use this representation to train a linear SVM (or kernelised at the expense of higher runtime). The SVM is deployed on test images as a sliding-window classifier, at multiple scales. The outputs of the detector are run through a conventional algorithm, "non-maximum suppression". The pipeline is so successful on the MIT benchmark dataset that they construct their own, more challenging, INRIA person detection dataset.

38.10.3. *Haralick Features.* Haralick features derive from the co-occurrence matrix of an image. The co-occurrence matrix is a matrix whose elements count the number of times pairs of pixel intensities occur in the image at a given offset. The elements are thus indexed by the pixel values themselves, and hence the co-occurrence matrix is square with dimension the dynamic range of the image. Haralick features somehow derive from this and are used to measure texture.

38.11. **Feature Detection.**

38.11.1. *Laplacian of Gaussian.* The Laplacian of Gaussian (LoG) is a linear filter that can be used for blob detection. From the multivariate Gaussian PDF,

$$\begin{aligned} G(x, y; \mathbf{\Sigma}, \boldsymbol{\mu}) &= \frac{1}{\sqrt{(2\pi)^k |\mathbf{\Sigma}|}} \exp\left\{ -\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \mathbf{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right\} \\ &= \frac{1}{2\pi\sigma^2} \exp\left\{ -\frac{x^2 + y^2}{2\sigma^2} \right\} \end{aligned}$$

where $k = 2$, $\boldsymbol{\mu} = \mathbf{0}$, and $\Sigma = \sigma^2 \mathbf{I}$. Differentiating,

$$\frac{\partial G}{\partial x} = -\frac{x}{2\pi\sigma^4} \exp\left\{ -\frac{x^2 + y^2}{2\sigma^2} \right\}$$

and differentiating,

$$\frac{\partial^2 G}{\partial x^2} = -\exp\left\{-\frac{x^2+y^2}{2\sigma^2}\right\}\left(\frac{1}{2\pi\sigma^4} - \frac{x^2}{2\pi\sigma^6}\right).$$

Now, the Laplacian of Gaussian,

$$\Delta G = \frac{\partial^2 G}{\partial x^2} + \frac{\partial^2 G}{\partial y^2} = -\frac{1}{\pi\sigma^4}\left(1 - \frac{x^2+y^2}{2\pi\sigma^6}\right)\exp\left\{-\frac{x^2+y^2}{2\sigma^2}\right\}$$

Recall that $\frac{\partial}{\partial x}\{f(x)\} * g(x) = \frac{\partial}{\partial x}\{f(x) * g(x)\}$, that is, the convolution of a derivative is the derivative of a convolution. Hence, the LoG computes a second derivate of a Gaussian-smoothed image. Blobs are topologically similar to a bell curve, so their second derivative is a sharp peak in the center, and easily thresholded after convolution with LoG.

The difference of Gaussians (DoG) is a linear blob detection filter consisting of the difference of two Gaussian filters, which may be shown to closely approximate LoG while being faster to compute.

38.11.2. *Harris Detector.* The Harris detector is a corner detection algorithm. The idea is to construct a small window W around a given pixel, and consider the difference in the window when one moves to a new pixel at an offset $(\delta x, \delta y)$,

$$E(\delta x, \delta y) = \sum_{(x,y)\in W} (I(x+\delta x, y+\delta y) - I(x,y))^2,$$

that is, the sum square error between the window pixels, and their counterparts in the offset window. For a small offset, the first term can be approximated by the first-order Taylor polynomial,

$$\begin{aligned} E(\delta x, \delta y) &\approx \sum_{(x,y)\in W} (I(x,y) + \delta x \cdot I_x, \delta y \cdot I_y - I(x,y))^2 \\ &= \begin{bmatrix}\delta x & \delta y\end{bmatrix}\left(\sum_{(x,y)\in W}\begin{bmatrix} I_x^2 & I_x I_y \\ I_y I_x & I_y^2\end{bmatrix}\right)\begin{bmatrix}\delta x \\ \delta y\end{bmatrix} \\ &= \boldsymbol{\delta}^T \mathbf{H} \boldsymbol{\delta} \end{aligned} \tag{76}$$

The Harris detector considers that windows around edges will see large E for many choices of $\boldsymbol{\delta}$, however not all[91]. A corner, however will see changes in all directions. A way of detecting this, therefore, is by identifying pixels for which E is always large, that is, the $\boldsymbol{\delta}$ minimising E for a given W is still large. Such pixels are likely to be corners, and can be identified with eigen-analysis of the matrix $\mathbf{H}$.

Consider a unit length $\boldsymbol{\delta}$ step from the target pixel. The extrema over choices of step minimise/maximise E. Thus, we have,

[91]Suppose the window moves perpendicular to the curve, the window contents could remain highly similar

$$\begin{aligned} \min_{\boldsymbol{\delta}} \quad & \boldsymbol{\delta}^T \mathbf{H} \boldsymbol{\delta} \\ \text{subject to} \quad & \boldsymbol{\delta}^T \boldsymbol{\delta} = 1 \end{aligned}$$

By the method of Lagrange multipliers, we can replace the above with the unconstrained problem,

$$\min_{\boldsymbol{\delta}} \quad \boldsymbol{\delta}^T \mathbf{H} \boldsymbol{\delta} - \lambda(\boldsymbol{\delta}^T \boldsymbol{\delta} - 1) \ .$$

Differentiating the objective function gives the normal equation,

$$(\mathbf{H} + \mathbf{H}^T)\boldsymbol{\delta} - \lambda \boldsymbol{\delta} = \mathbf{0}.$$

Noting that due to symmetry $\mathbf{H} = \mathbf{H}^T$ it is clear $\boldsymbol{\delta}$ is an eigenvector of $\mathbf{H}$ and λ is its eigenvalue. Further note that because $\mathbf{H} \in \mathbb{R}^{2\times 2}$ its other eigenvector will maximise E. Whereas an edge point will have a large eigenvalue λ_+, it will also usually have a small eigenvalue λ_-. Thus, a strategy for discerning corners from edges and other points is to compute the ratio λ_-/λ_+, which wll be high for corners. The Harris detector exploits the fact that the harmonic mean of two values of different magnitudes is close to the smaller value,

$$\begin{aligned} \lambda_- &\approx \frac{\lambda_- \cdot \lambda_+}{\lambda_- + \lambda_+} \\ &= \frac{det(\mathbf{H})}{tr(\mathbf{H})}, \end{aligned} \tag{77}$$

which can be computed quickly. Further note that in practice, the sum of derivates over W for each image pixel is implemented as a Gaussian filter (that is, a weighted sum product)[92].

38.11.3. *Hough Transform.* The Hough transform was originally designed to find straight lines in an image. Starting with some gradient-based rule, candidate edge points are detected by searching the image pixel-by-pixel. Straight lines are drawn at various angles from each given candidate. Each resulting line can be expressed as an angle and perpendicular distance from the origin, (ρ, θ). From this, each candidate can be represented as a curve in ρ-θ space, the Hough transform space. Repeating for all candidates results in a collection of curves. The intersection of two curves corresponds with the alignment of two points in a straight line. Multiple intersections correspond with multiple aligned points. To find the most heavily weighted intersections, we can simply add a weight to each point in the curve of each candidate, and sum the curves. The peaks in the resulting Hough space image correspond with groups of aligned points, and these peaks can easily be identified through a global threshold.

[92]https://github.com/cournape/scikits-image/blob/master/skimage/feature/harris.py#L40-L49

Since the original formulation, the Hough transform has been extended to find points aligned in arbitrary shapes, in particular circles and ellipses.

38.11.4. *SIFT.* Scale-invariant feature transform (SIFT) is motivated by image matching, that is, finding corresponding sets of keypoints (homographies) of image contents. SIFT both identifies these keypoints, and computes features around them, yielding a scale- and rotation-invariant image representation. Keypoint detection occurs in two phases: an inital blog detection phase; and a refinement phase.

Blob detection is based on the difference of Gaussian (DoG), a fast approximation to Laplacian of Gaussian (LoG), in which two Gaussian blurs with different variances are calculated and subtracted. To account for scale, this blob detection is performed across a set of geometrically scaled (1.5x bilinear interpolation according to the paper), which further incorporate Gaussian noise. Local minima/maxima are detected by comparing with all adjacent pixels, first in space, then in scale.

Keypoint refinement involves a number of steps to eliminate weak keypoints, but one is essentially Harris detection: remove curve keypoints in favour of corners.

At this point, we return to the original image, now with keypoints that have stable location l and scale s properties. The next criterion is orientation o. A histogram of gradients is computed, and the peak is chosen as the dominant orientation of the keypoint (l, s, o). Any competing peak within a certain margin is taken to create a duplicate keypoint (l, s, o').

Now, essentially magnitude-weighted gradients are computed in a grid of cells in a window (reflecting the originating scale) around the keypoint, and normalised ("sorted", although the paper does not use this term) with respect to the dominant orientation from the previous step. Each histogram is normalised to a unit vector to control for variance in illumination (the magnitudes used for weighting would all by larger if illumination is locally high).

Note that varying numbers of keypoints are detected for each image. The image matching appllication is based on nearest neighbour matching between sets of keypoints. RANSAC is a fully-fleshed algorithm for this. The homography can be computed as a least squares of the coordinates to solve for the homography matrix (affine transformation). A nice application is image stitching for panoramas: first compute SIFT, then match keypoints, then compute homography and warp to a reference image.

39. Computer Architecture

The von Neumann architecture, named after Hungarian polymath John von Neumann (1903-1957), is the basic template for assembling a computer. It centres around a central processing unit (CPU) and primary memory source. Input and output devices may be connected to this processing core. In a modern personal computer, the components are organised across a printed circuit board (PCB)[93], known as a *motherboard*. A chip is a

[93]A printed circuit board (PCB) is a plate of layered material. The top layer is made of a conductive copper and sits atop a non-conductive fibreglass layer. The circuit pattern is created by pressing a mask of the printed design to the copper plate and soaking it in an acid solution to remove the surrounding

standalone circuit with a specific function, such as memory or processing. These components are soldered onto the motherboard to connect them to the rest of the computer hardware. The wiring groups between components are known as *buses*, and are used to transfer information throughout the system. However fine the motherboard circuitry is, the internal circuitry of an integrated circuit is far finer, having of the order of billions of transistors per square centimetre. When a processor is contained in a single integrated circuit, it is referred to as a *microprocessor*. This is the defining characteristic of a microcomputer or personal computer (PC). Personal computers have long supplanted the large, mainframe computers of earlier decades.

The distinction between hardware and software is at times not well defined. Thus, it is difficult to talk about one without the other. Ultimately, everything exists in the physical layer, software just being structured (low entropy) configurations of electrical signals buried in the hardware. Software has varying levels of abstraction, right up to the user-friendly graphical interfaces, but at least in some sense, this is all just an illusion created by organising lower-level parts.

39.1. **Hardware.** As it is in the von Neumann architecture, the centrepiece of computer activity is the interaction between the central processing unit (CPU) and the (primary) memory–all other components are arguably peripheral to this. When a computer is turned on, a component within the CPU begins to oscillate. This is like the heartbeat of the computer, and provides a periodic *clock* bit to each computation. In a Turing machine, the clock would be the invisible force driving the transitions along the tape. With some additional circuitry, the clock signal is accelerated to a *clock rate* of the order of billions of hertz (Hz). This largely determines the speed of the computer, though the practical rate of computation, or floating point operations per second (FLOPS), will depend on other things, such as the efficiency of the circuitry.

The CPU first runs a primitive piece of firmware that solves the bootstrap problem[94]. On a PC, this is called the BIOS (Basic Input Output Software), and UEFI (Unified Extensible Firmware Interface) on a Mac. The BIOS performs checks to ensure all the hardware is functional. It also provides a user interface for making low-level changes to the system[95]. The BIOS is usually located on a special chip of read-only memory (ROM), installed by the manufacturer. The BIOS finds the boot loader stored in secondary memory–typically a hard disk drive (HDD) or solid-state drive (SSD), but other secondary memory such as

copper. Finally, a solder mask in the familiar blue or green material is placed on top of the copper traces to protect them from dust and debris, while leaving holes at the connection points. Components may then be connected to the copper circuit using melted solder (a lead-tin alloy) to fuse components to the exposed connection points. Thus, the PCB replaces traditional wiring with the copper traces, functioning as the circuitry as well as a base to which integrated circuits and other components are attached.

[94]The bootstrap is conceptually the necessary prime mover for all further software on the system.

[95]For example, a CPU can be *overclocked* to increase performance. This can be controlled from the BIOS. As a general rule, the more active the CPU is, the more voltage in its circuits per unit time. More voltage creates more heat, and thermal sensors in the motherboard cause the fans to work harder to cool the chip. Overclocking can have detrimental effects if the circuitry gets too hot.

a USB drive, CD-ROM, or the former floppy disk, are sometimes used[96]. The boot loader is then loaded into memory. Its task is to load the operating system (OS) into memory in turn. Again, the user may interrupt this procedure in order to choose between operating systems installed on the drive. Once the operating system is in memory it has control of the system, giving the user the ability to control the computer and load and execute other software.

39.2. **Software.** Software exists in memory as a sequence of precompiled instructions, usually loaded from secondary memory. The control unit (CU) within the CPU is hardwired to take each instruction in turn and run it through the arithmetic/logic unit (ALU) circuitry. *Registers* are small, localised pieces of memory for storing inputs for CPU operations. At this level, the code is not interpreted, but rather hardwired. Each instruction has an operator code which channels execution to the appropriate circuit. The form of the instructions are defined in an assembly language, which reflects the hardwired functions provided by the processor. An example of this is the x86 and x64 Intel resp. 32-bit and 64-bit instruction sets. Assembly code is usually automatically generated from high-level language code by a compiler. The assembly code is then transformed into a binary encoding called machine code. There is more or less a one-to-one mapping between assembly and machine code. In this way, machine code is more of an encoding than a language. As an example, take the simple high-level operation `a + b`, as it would be in `C`, or another programming language. When translated to assembly code, this will look something like Algorithm 5.

Algorithm 5 Assembly pseudocode for a `a + b` operation.

```
1: LDR R1 a // load of 1 with variable a
2: LDR R2 b // load register 2 with variable b
3: ADD R1, R2, R3 // add register 1 and 2 and put the answer in register 3
```

This follows what is called *reverse Polish notation* (RPN), a convenient way to encode such instructions. This might then be translated to machine code as `01001100 00000010 00000011 00000100`, that is, perform operation 76 (addition) on registers 1 and 2 and put answer in register 3. Binary code is directly translatable to a system of high and low voltages. Thus, the program can exist in secondary storage as a series of electrical signals, ready to be loaded into memory by the operating system instructions for processing. To better understand how this processing works, it is necessary to descend to the electronics level.

39.3. **Digital Circuitry.** The fundamental component of electrical computers is the transistor. Conceptualised in 1926 by Hungarian physicist, Julius Edgar Lilienfeld (1882-1963), it was first implemented by a trio of Bell labs researchers in 1947, work for which they received the Nobel prize in Physics in 1956. A transistor is made of semiconductor material such as silicon (Si). It functions as a bridge between a source and sink wire, providing a third, base wire that controls the conductivity of the silicon material. If current is passed

[96]A user can usually interrupt the BIOS and choose where to boot from.

through the base wire, the bridge is opened and current flows from source to sink. In this way, the current on the main wire can be switched on or off very quickly and with no moving parts. The size of a computer overall therefore depends crucially on how small we can make transistors. Moore's law[97] predicts that the number of transistors per integrated circuit doubles every two years. This allows for building computers that are twice as powerful while retaining the same approximate size, cost, and power consumption. However, we are rapidly approaching the physical limit on transistor size. At 1.4×10^{-10}m, quantum effects are starting to interfere with accuracy.

39.3.1. *Logic Gates.* Using combinations of transistors, we can create logic gates. As it is in boolean normal forms, the only gates we need are conjunction (`AND`), disjunction (`OR`), and negation (`NOT`), and we can wire any logical circuit. An `AND` gate can be created by placing two transistors in sequence. If both transistors are activated, current will flow overall. If either or both transistors are not activated, current will not flow. Thus, it may be seen that submitting some combination of on/off (`TRUE`/`FALSE`) currents to the transistors will result in an overall on/off current through the wire, in accordance with the truth table of a logical `AND` operator. Thus, we have a circuit capable of computing a logical `AND`. An `OR` gate may be created by placing two transistors in *parallel*, and a `NOT` gate or *inverter* by combining positive and negative transistors.

39.3.2. *Adders.* From logic gates we can construct adders. The exclusive-or (`XOR`) operation handles the first part of binary addition: $0 \oplus 0 = 1 \oplus 1 = 0$ and $1 \oplus 0 = 0 \oplus 1 = 1$. The only thing that is not accounted for is the *carry bit*. For this reason, `XOR` is called a *half-adder*. It can be created in a number of ways, for example, as `OR AND NOT AND`. To create a *full-adder*, it is combined with an `AND` gate to represent the carry bit. To add an 8-bit number, eight full-adders can be arranged in sequence, with each carry bit feeding into the next. More complex operations such as multiplication can be created by combining multiple full-adders. Examples of other CPU operations are the `COMPARE` and `JUMPIF` instructions. These are used in sequence to dynamically move among instructions in memory based on some logical test. This is what enables loops and if statements to occur.

39.3.3. *Memory.* The type of memory usually used as primary memory is *random access memory* (RAM). Random access refers to the fact that any memory address can be accessed equally quickly. This is in contrast to a hard disk drive used for *secondary* memory, where a latency of a few milliseconds is incurred to move the requested memory sector under the drive's read-write head. Increased affordability has brought about the advent of the much faster *solid state*[98] (SSD) secondary memory. Note that SSD memory is still orders of magnitude slower than RAM. Despite being fast, RAM is *volatile*. Volatility is a property referring to the loss of memory when the power is switched off. This contrasts with the *persistent* memory provided by secondary memory sources. There are two main types of RAM: static RAM (SRAM) and dynamic RAM (DRAM). Usually SRAM is used in a CPU cache. A cache is used to store recently used data in fast memory near to the CPU.

[97]Named after Gordon Moore (1929-), one of the founders of Intel Corporation.

[98]That is, no moving parts.

This is an heuristic strategy that helps to minimise the delay incurred on frequently used data by the much slower main memory access, the CPU being far faster. Typically, only a small cache is sufficient to speed up processing dramatically. Like logic gates, a unit of static RAM is constructed by wiring transistors in a special way. We start by wiring two `NOR` gates together such that one of the inputs of each is the output of the other. The remaining input wires are denoted the *set* and *reset* wires. With this configuration, it is possible to send current through the set and reset wires to switch the signals between the `NOR` gates, and such that this signal persists when the current stops. This mechanism is called a RS latch, and provides a way to store a binary signal. When combined with a wire for write data, and the CPU clock bit, we can create what is called a *flip-flop*. Each flip-flop is used to store a single bit of information, and these are arranged into an array that can be addressed by row and column number. In contrast, DRAM is cheaper and more suited to primary memory storage. This consists of an array of capacitors to store bits. The capacitors only hold the bit signal for a small amount of time (milliseconds), so a mechanism is used to quickly read and rewrite memory periodically (every 100 nanoseconds or so). This is why DRAM is known as *dynamic*, though it might better be referred to as *extra-volatile* memory.

39.4. **Operating Systems.** An operating system (OS) is the primary software running on a computer. As well as generally providing the user with a means of manipulating the computer, it facilitates the running of other programs on the hardware. A program is loaded into memory as a *process*, with its own *heap* of memory. A heap is a contiguous block of main memory. When memory is allocated dynamically in a language like `C` it takes it from the process heap. Segmentation faults occur when memory on the heap is abused. A process consists of one or more *threads*, which represent streams of execution. Each thread has a *stack*, which is a FIFO queue for instructions and data. The stack is pushed and popped in accordance with the requirements of the program. The stack grows with the initialisation of local variables, but also with each new call to a nested function. Therefore, a maximum recursion depth is imposed by many languages to prevent the stack memory allowance being exceeded, an event known as a *stack overflow*. A process scheduler in the operating system controls the access to hardware resources for each process. This is essential to a multiprogramming operating system. Input and output devices are connected to the bus with addresses. IN/OUT instructions can send or retrieve data from these devices via a data bus. The operating system provides *driver* software for interfacing with hardware components. The heart of an operating system is the kernel. This provides various services for memory allocation of processes, organisation of the OS file system, handling interrupts from peripheral devices, and so on.

39.4.1. *The Unix Story.* Unix is an important operating system that was developed at Bell Labs by American computer scientists Ken Thompson (1943-) and Dennis Ritchie (1941-2011). It is the ancestor of many of the most widely used operating systems used today. At the time, Unix was favoured due to its speed, stability, modularity, and security. Unix was the first operating system written in a high-level language (`C`, itself created by the same people), and was thus portable to any hardware with a `C` compiler. It also had

just the right set of innovations to make it the ideal operating system, for example *piping*, used to chain software together via their outputs. However, Unix was proprietary software owned by AT&T[99], and interest developed to create free equivalents. In particular, MIT AI lab programmer Richard Matthew Stallman (1953-), started the Free Software Foundation (FSF) and the GNU (GNU's Not Unix!) project to create a free operating system. This was a pioneering initiative in the history of open-source software.

Apart from being free to use, the idea of a GNU operating system was to expand *user land*, that is, the part of the OS flexible to a user's preferences, having only a "micro-kernel" and a set of replaceable daemons[100]. The sense of *free* software is really to do with free control over the software–gratuity is only one part of that. The GNU micro-kernel is in contrast to a complete, *monolithic* kernel such as the Linux kernel. The micro-kernel was to be known as Hurd (a gnu is a type of antelope). Due to some technical difficulties, progress stalled and the project was supplanted by the Linux project, developed by Finnish programmer, Linus Torvalds (1969-), taking with it much of the support and enthusiasm for GNU. Later, divisions arose in the GNU community due to the divisive political vision of GNU. The GNU project did, however, create a large amount of very important software, for example the GNU compiler collection (GCC), the GNU Core Utilities (reimplementations of `cat`, `ls`, `rm`, etc.) the BASH shell[101], `glibc` (GNU's implementation of the C standard library), the GNOME desktop environment (which runs on the X Windows system for bitmap displays), and Emacs. These tools are used on many other operating systems, including Windows, OS X, and Linux. It is for this reason that Richard Stallman refers to Linux as a *GNU* operating system with Linux kernel. It is unclear whether the pure GNU operating system will ever be finished, but its legacy endures. Another notable output of the movement was the GNU Public License (GPL), a software license supporting *copyleft* practices, that is, free reuse of intellectual property, provided the same copyleft open-source license is retained, and therefore the ability to reuse that software too. Such an arrangement can be described as *share-alike*, and is adopted by other open source licenses[102].

At approximately the same time, the systems lab at the University of California, Berkeley, were extending their licensed version of Unix, in particular, extending its TCP/IP functionalities. This came to be known as the Berkeley Software Distribution (BSD) of

[99]Bell Labs, NJ, was owned by AT&T for most of the 20th century. It is credited with inventing radio astronomy, the transistor, information theory, `C`, `C++`, Unix, and the one time pad (Vernam cipher). It is now owned by Nokia.

[100]In Unix, daemons are named with a trailing 'd'. This is why the Apache HTTP Server, `httpd`, (the most commonly used web server) is so named.

[101]The Bourne Again Shell was an open-source replacement for the Unix Bourne Shell, an improvement to the Thompson Shell (no scripting).

[102]There are more permissive licenses such as Berkeley Software Distribution (BSD) and the MIT license, arguably even more permissive than BSD. This comes only with the disclaimer that the author shall not be responsible for adverse effects caused by the free use of their software. The GNU Lesser General Public License (LGPL) was created as a compromise between MIT/BSD and GNU GPL. Note also that these licenses are free to use, the copyright is applied without the need to register the software. The license text is usually put in a project file marked `LICENSE`. In short, GPL is a copyleft license, BSD is an acknowledgement, and MIT is a disclaimer.

Unix, but it remained subject to AT&T licensing restrictions. The Unix parts were gradually rewritten, and BSD and its variants (FreeBSD, OpenBSD, and others) came under the BSD license, a highly permissive license, allowing the full reuse of software, even in proprietary products, provided inclusion of the license text. Many of the operating systems of the present day are descended from Unix via the free alternatives of BSD. For example, BSD forms a large part of Apple's OS X operating system. BSD was the subject of a law suit by the Unix proprietors in the early 90s, during a time nicknamed the "Unix wars", with BSD ultimately winning the case. Unlike Linux, BSD does not use GNU, it has its own OS software including, for example, the famous vi editor. Linux is more user-friendly, however, with BSD having a command line interface (CLI) orientation. Thus, whereas GNU reverse engineered Unix, BSD evolved from it directly like the ship of Theseus. Both extended it far beyond its initial capabilities. Because of them, we now have whole families of software that are deemed "Unix-like".

40. Useless Maths Facts

- The length : width ratio of all standard paper sizes (A4, A5, etc.) is $\sqrt{2} : 1$. It is easy to show folding a sheet of paper in half width-ways retains this same ratio.
- There are 10! seconds in 6 weeks
- Mersenne primes are primes of the form $2^n - 1$ for some integer n. The Great Internet Mersenne Prime Search (GIMPS) is an ongoing collaborative experiment to find Mersenne primes. The largest Mersenne prime (and largest known prime) found to date is $2^{74207281} - 1$. Only 49 Mersenne primes have ever been found.
- $1/e \approx 37\%$
- Pairs of primes of difference two (e.g. 5, 7) are known as twin primes. Pairs of primes of difference four (e.g. 7, 11) are known as cousin primes. Pairs of primes of difference six (e.g. 11, 17) are known as sexy primes.
- According to legend, the Pythagorean mathematician, Hippasus, was killed for proving the existence of irrational numbers. This conflicted with the dogmatic worldview of the other Pythagoreans.
- For any map (for example, one of a geographic region), there is always a way to colour the map with at most four colours such that no two sections of the map sharing a border have the same colour. This comes from the four colour theorem, the first theorem in the world to be proved (partially) by computer (brute force).
- The mirror reflection of 3.14 closely resembles the word 'pie'.
- Perfect numbers are those whose factors (including 1) sum to itself. For example, $6 = 1 + 2 + 3$. Amicable numbers are pairs of numbers each of whose factors sum to the other. For example 220 and 284.
- 23 is the first number at which the Birthday paradox gives a greater than 50% chance of a collision
- $2^{20} \approx 10^6$
- One googol is 10^{100}, that is, 1 followed by 100 zeroes. A googolplex is $10^{\text{googol}} = 10^{10^{100}}$. A *googolplexian* is $10^{\text{googolplex}} = 10^{10^{10^{100}}}$.

- Graham's number is a famous named number larger by incomprehensible magnitudes. It provides an upper bound to the solution of a problem in graph theory. It is so large, it can only practically be written in terms of amalgamations of hyper-operations, exponentiation being absurdly insufficient. Using the specialised Knuth's arrow notation, Graham's number can be written as the last in a sequence 64 numbers, the first being $g_1 = 3 \uparrow\uparrow\uparrow\uparrow 3 = g \uparrow^4$, that is, 3 hexation 3 (already unimaginably big), the next being $g_2 = 3 \uparrow^{g_1} 3$, where $\uparrow^{g_1}$ is a hyperoperation whose order is g_1 (recall exponentiation is order 3 and hexation is order 6). It is necessary to pause and try to think what this means (you can't). Thus, the sequence continues, with $g_n = 3 \uparrow^{g_{n-1}}$, all the way to $g_{64} = 3 \uparrow^{g_{63}}$, and g_{64} is Graham's number. There are simply no conceivable analogies to describe or relativise this number, or even the first number in the sequence. If, for example, you wrote 1 digit on every Planck length volume in the universe, once every Planck time unit, from the start to the end of the universe, you would still have completed 0.000000...% (with another (slightly smaller) inconceivably large number of zeroes) of even g_1.
- With 42 folds of a single sheet of paper, you would have a tower high enough to reach the moon. Given the thickness of paper is 1×10^{-4} meters (0.1 mm) and the moon is 3.8×10^9 meters (380,000 km) away, the number of folds required solves the equation $10^{-4} \times 2^{\text{folds}} = 3.8 \times 10^9 \implies \text{folds} \approx 42$. Unfortunately, it is not possible to fold paper more than six or seven times! 103 folds would span the size of the universe.
- $1729 = 10^3 + 9^3 = 12^3 + 1^3$ is the smallest number expressible by two positive cubes in two different ways. According to legend, this observation was first made by Ramanujan to G. H. Hardy when Hardy informed him he had travelled in taxi number 1729 to visit him. The number makes a number of other famous appearances in number theory.
- The 'number of the beast', 666, is likely to have originated from an encoding of the Hebrew name of the Roman emperor, 'Nero Caesar'.
- $0.999... = 1$. A paradox arises if one does not appreciate what 'forever recurring' means. To see this, let $x = 0.999....$ Then $10x = 9.999...$ and so $10x - x = 9.999 - 0.999 = 9 \implies x = 1$.
- There is a fairly compelling argument for replacing π with tau, $\tau = 2\pi$. Many formulas from the normal pdf, to the Fourier transform, to the period of a sine wave use a superfluous 2π. It is also more difficult to teach with half circles than full circles.
- There is a good argument for replacing the decimal system (which is rooted in the metric system that came from revolutionary France) with the *dozenal* system (base-12). Common fractions are more easily expressible in base-12, since it has more factors than 10, so measurements and arithmetic become easier to learn and work with. Though humans have 10 fingers in total, they have 3 segments on each of their 4 fingers on each hand, making base-12 equally convenient.
- Most integers contain the digit 3. Consider the numbers before the first power of 10, namely, $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. Clearly there is only one number containing a

3 (3 itself), and we write $N(1) = 1$, and the proportion is $P(1) = 1/10$. For the first hundred numbers, the pattern is the same, with each group of tens containing one number with a 3, which are $3, 13, 23, \ldots, 93$, except for the thirties, which have $30, 31, 32, 33, \ldots, 39$. Thus we have $N(2) = 9\times1+10 = 19$, and $P(2) = 19/100$. For the thousands, the same is true for each hundred, except for the three hundreds, all of which contain a 3. Hence, $N(3) = 9 \times (9 \times 1 + 10) + 100 = 199$, and $P(3) = 199/1000$. It is clear that in general, for the kth power of 10 we have $N(k) = 10^{k-1} + 9 \cdot N(k-1) = 10^{k-1} + 9(10^{k-2} + 9(10^{k-3} \ldots 9 \cdot N(1))) = \sum_{i=1}^{k} 9^{i-1} \cdot 10^{k-i} = (9/10^k) \cdot \sum_{i=1}^{k} (9/10)^i$. Thus, we have a geometric series displaced by one term, and so $N(k) = (9/10^k) \cdot ((1-(9/10)^k)/(1-(9/10))-1)$, and $P(k) = N(k)/10^k$. As $k \to \infty$, $P(\infty) = \lim_{k\to\infty} P(k) = (1/9) \cdot (10 - 1) = 1$, hence the proportion of numbers containing at least one digit 3 goes to 1. Of course, the same can be shown for the other 9 digits, and the general finding is that almost all numbers contain every digit.

- Mill's constant is a number that, conditioned on the (unproven) Riemann hypothesis, is the base of a double exponential function that always yields an integer part that is prime. That is, Mill's number is $A \approx 1.306$ and $\lfloor A^{3^n} \rfloor$ is always prime.
- Fractals have very interesting self-similar properties. For example, Koch's snowflake is a geometric shape with infinite perimeter but finite area. By inspection the area is finite, as a circle of finite diameter can be drawn around it provided it is sufficiently large. The perimeter is the result of repeating infinitely many times a procedure that reduces side-length by a factor of three, but nevertheless quadruples the number of sides each time. Repeating ad infinitum therefore results in a perimeter that is $\lim_{N\to\infty}(4/3)^N = \infty$ times the length of the initial perimeter.
- A Möbius strip is a three dimensional shape with a single surface and one edge. It can be emulated by twisting a ribbon by a half-turn and connecting its ends. A Klein bottle is a *four*-dimensional solid with a single surface and *no* edges! It is formed by fusing two Möbius strips along their edges.
- Skewes' number, $10^{10^{10^{34}}}$ is another famous large number, and the old record holder before the discovery of Graham's number. Named after one of John Littlewood's students, Stanley Skewes, it gives an upper bound on the smallest number for which Gauss' prime-counting function, $\pi(n)$ is greater than the logarithmic integral function, $\mathrm{li}(n)$ (another prime-counting function initially thought to always be greater than Gauss').
- The harmonic series, $h = \sum_{n=1}^{\infty} 1/n$ is (surprisingly) divergent. Clearly, $h = 1 + 1/2 + (1/3 + 1/4) + (1/5 + 1/6 + 1/7 + 1/8) + \cdots > 1 + 1/2 + (1/4 + 1/4) + (1/8 + 1/8 + 1/8 + 1/8) \cdots = 1 + 1/2 + 1/2 + 1/2 + \cdots$, which is clearly divergent.