A Dummies Guide to ML and Statistics

Benedict King

Last Edited: October 18, 2021

Contents

1 Important terms and psychology to be aware of

1.1 Confirmation Bias

Searching for evidence in data that supports a skewed prior belief even in the face of poor data support, or without exploring other conclusions that could be drawn.

Nate Silver, *The Signal and the Noise*:

'The instinctual shortcut when we have too much information is to engage with it selectively, picking out the parts we like and ignoring the remainder, making allies with those who have made the same choices and enemies of the rest.'

1.2 Availability Bias

The tendency to have a bias to recent past outcomes when predicting future outcomes, or to overweight easily available information in making predictions.

Gambler's/Monte Carlo Fallacy:

The incorrect viewpoint that if an (stationary) event occurs more (or less) frequently than expdected in the past, then it will be less (or more) likey to occur next time - for example if a fair coin has shown heads 10 times in a row, it is still just as likely to show heads next throw, despite a tendency to predict a tails.

1.3 Anchoring

A cognitivie bias, putting too much information on an initial piece of information (the *anchor*), even if unrelated, to make predictions on future pieces of information.

Examples:

A salesperson can exaggerate the value of an item as an initial starting price, and the buyer's view of the item's value is then skewed as as result of this anchor, even if the price decreases. Anchoring can happen between unrelated events as well, for example, if asked to name the colour produce mixing red and yellow and then asked to name a vegetable, a person is more likely to answer 'carrot' for the second answer due to the anchor of the answer to the first.

1.4 Framing

For a fixed event with constant probability outcomes, the way in which the outcomes are described can affect decisions that a person makes based on the event probabilities.

Example:

Say a treatment is available for a disease that has a 80% chance of success, but with a 10% chance of adverse side effects. A person offered the treatment is more likely to go for it if told that it has an 80% chance of success and 90% chance of no adverse effects instead of being told that it has a 20% chance of failure and a 10% chance of adverse side effects.

1.5 Misunderstanding the Law of Averages

The very simple law of averages says that values taken from a given distribution tend to average out at the expectation value. Practically this is often misunderstood due to the availability bias mentioned above - if an observed value falls far from the mean, there is a tendency to assume that there is some cause behind it, and that subsequent values might be affected by this cause, when in fact it is just random fluctuation.

Leanord Mlodinow,*The Drunkards Walk*:

'I've often praised people warmly for beautifully executed maneuvers, and the next time they always do worse. And I've screamed at people for badly executed maneuvers, and by and large the next time they improve. Don't tell me that reward works and punishment doesn't...' - The learning curve for pilots isn't fast snough for one session to deviate from the mean, this is a misunderstanding of the law of averages.

1.6 Expectation Bias

A psychological trick in which a person subconsciously changes their judgements based on a previous expectation.

2 Clustering and Categorisation

2.1 k-Nearest Neighbours (k-NN)

A simple **supervised** algorithm that clusters data points based on their distance (Euclidean/Chebyshev) seperation in feature space. k represents the number of neighbours to a data point to be found to classify the 'belonging' of the test data point.

The process iterates as follows for each data point in the test set:

1. Calculate the distance metric from the current point to every point in the training set (pre-classified dataset is created by supervisor)
2. Sort the list of distances by increasing distance and select the first (nearest) k points
3. For the selected points, find the modal (most common) class they belong to - this is the prediction for the current data point
4. The error rate is dictated by the ratio of incorrect classes selected in th k nearest neighbours to k

A low k will lead to overfitting - very high training accuracy but low validation accuracy. Increasing k increases computation time however and also decreases stability.

2.2 k-Means Clustering

Similar to k-Nearest Neighbours clustering, but k-Means clustering doesn't require a pre-classified dataset. Instead the classification is learned given a specified k (number of different categories). Therefore this represents an **unsupervised** learning algorithm.

The algorithm is as follows:

1. Randomly choose the coordinates of the k mean points of the clusters
2. For each point in the dataset, find which of the means is closest (by any specified distance metric) to that point - the point then belongs to the category of the closest mean
3. After the whole dataset is classified, update the k mean values as the mean position of all the points in that category
4. Iterate from (2) again with the updated means, and stop after a certain epoch number or if the incremental change in the means is below a threshold

The downside to this algorithm is that the number of clusters must be specified at the start and that it can't pick up shaped clusters, as most distance metrics (particularly Euclidean distance) calculates distance as the radius of an n-dimensional sphere, so all clusters must be roughly spherical. In addition, the clustering speed and results relies on the initialisation which is random, but this isn't a dramatic problem.

Repeated in more detail in Cambridge pre-reading Section 11.3.1

2.3 Density-Based Spatial Clustering for Applications with Noise (DBSCAN)

3 Feature Selection and Reduction

3.1 Principal Component Analysis (PCA)

The Covariance matrix represents the variance along each feature down the diagonal, and the covariance between features on the off-diagonals. The greater the magnitude of the covariances (positive and negative values represent positive and negative correlation), the greater the dependence of the features on each other.

$$C = \begin{pmatrix} \sigma_1^2 & \sigma_1\sigma_2 & \dots & \sigma_1\sigma_n \\ \sigma_2\sigma_1 & \sigma_2^2 & \dots & \sigma_2\sigma_n \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_n\sigma_1 & \sigma_n\sigma_2 & \dots & \sigma_n^2 \end{pmatrix} \tag{1}$$

PCA projects the feature space onto a linearly independent set of basis vectors such that the new basis covariance matrix is strictly diagonal. The basis vectors of this set are the eigenvectors of the Covariance matrix (as the covariance matrix is positive definite, and therefore has linearly independent eigenvectors that diagonalise the covariance matrix), and the corresponding eigenvalues are the variances in each of the new principal componenents.

$$C^{'} = P^{-1}CP \tag{2}$$

$$= \begin{pmatrix} \boldsymbol{v_1} & \boldsymbol{v_2} & \dots & \boldsymbol{v_n} \end{pmatrix} \tag{3}$$

$$C^{'} = \begin{pmatrix} \sigma_1^{'2} & 0 & \dots & 0 \\ 0 & \sigma_2^{'2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_n^{'2} \end{pmatrix} \tag{4}$$

If the eigenvalues (variance of principal components) are below a certain value, then those principal components can be discarded, thus reducing the dimensionality of the dataset.

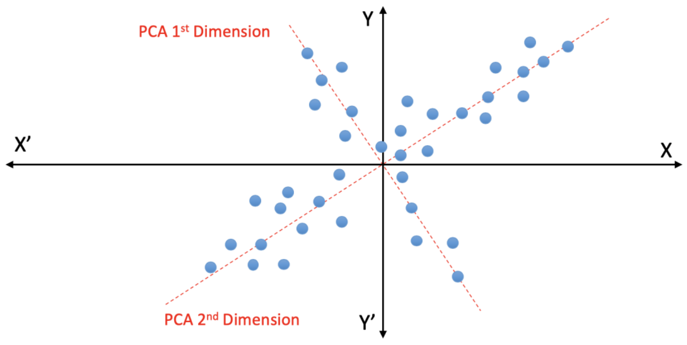


Figure 1: PCA example in 2 dimensions.

Repeated in more detail in LGMs Section 6.1

3.2 Independent Component Analysis (ICA)

3.3 (t-distributed) Stochastic Neighbour Embedding (t-SNE)

4 Miscellaneous

4.1 Probability Basics

1. **Product rule:** if $X \perp\!\!\!\perp Y$

$$P(X,Y) = P(X)P(Y) \tag{5}$$

2. **Marginalisation:**

$$P(Y) = \sum_{x \in X} P(Y,x) = \sum_{x \in X} P(Y|x)P(x) \tag{6}$$

3. **Bayes' theorem:**

$$P(X|Y) = \frac{P(X,Y)}{P(Y)} = \frac{P(Y|X)P(X)}{P(Y|\neg X) + P(Y|X)} \tag{7}$$

   NB: The *Posterior* distribution is the distribution of parameters given the observed data, $P(\theta|\vec{y})$. The *Posterior Predictive* distribution is then the distribution of unobserved values ($\vec{x}$) given these parameters and the observed data:

$$P(\vec{x}|\vec{y}) = \int_\theta P(\vec{x}|\theta,\vec{y})P(\theta|\vec{y})d\theta \tag{8}$$

4. **PDF mapping:**

$$P(x)dx = P(y)dy \tag{9}$$

$$P(y) = P(x)\frac{dx}{dy} \text{ for } y = y(x) \tag{10}$$

5. **Probability chain rule:** for if $X \not\perp\!\!\!\perp Y$

$$P(X,Y) = P(Y|X)P(X) \tag{11}$$

$$P(X_1, X_2, ..., X_n) = P(X_n|X_{n-1}, ..., X_1)P(X_1, ..., X_{n-1}) = P(X_1|X_2, ..., x_n)P(X_2, ..., X_n) \tag{12}$$

   An example of its use is in relating probability distributions:

$$P(X,Y|Z) = \frac{P(X,Y,Z)}{P(Z)} = \frac{P(X|Y,Z)P(Y,Z)}{P(Z)} = P(X|Y,Z)P(Y|Z) \tag{13}$$

4.2 χ^2 Test and Distribution

As a test of statistical independence between the two variables. With dataset rows i and columns j, the χ^2 statistic measures their independence via:

$$\chi^2 = \sum_i \sum_j \frac{(O_{ij} - E_{ij})^2}{E_{ij}} \tag{14}$$

Here O is the observed frequency, and E the expected frequency.

Alternatively, when concerned with the distribution, for observed data with a measurement standard error α:

$$\chi^2 = \sum_i \frac{(\hat{Y}_i - E[Y])^2}{\alpha_i^2} = \sum_i Z_i^2 \tag{15}$$

for $Z_i = \frac{Y_i - E[Y_i]}{\sigma_i}$ and $Z_i \sim \mathcal{N}(0,1)$. Note that **all $Y_i's$ are independent**.
For example: If Y is the height of individual bricks in a wall (assume no mortar), then sum of the heights of the bricks, ie. the overall height of the wall, is distributed as a χ^2 distribution with mean ν and variance 2ν, for ν d.o.f. (The number of d.o.f. is equal to the number of free parameters, eg. the number of data points minus number of fitted parameters).

4.3 Kullback-Leibler Divergence

The Kullback-Leibler divergence is a measure of the distance (i.e. the difference) between two distribution, calculated by measuring the difference between their Shannon entropies. Importantly, it is asymmetric, and therefore is not counted as a statistical measure of spread. It is defined as:

$$\boxed{D_{KL}(P \parallel Q) = \sum_x P(x) \ln\left(\frac{P(x)}{Q(x)}\right) = -\sum_x P(x) \ln\left(\frac{Q(x)}{P(x)}\right)} \tag{16}$$

The non-symmetric nature means that it is useful to think of a context to what the KL-divergence represents. $D_{KL}(P \parallel Q)$ can be thought of as the information gained if distribution P was to be used instead of Q which is already being used (hence using Q if Q already used would give $D_{KL}(Q \parallel Q) = 0$ gained information). This can be inferred from the definition above, which can be rewritten as:

$$D_{KL}(P \parallel Q) = H[P(x)] - \sum_x P(x) \ln\left(\frac{1}{Q(x)}\right) = H[P(x)] - H[P(x), Q(x)] \tag{17}$$

Thus the KL-divergence is the difference between the entropy of the $P(x)$ distribution (i.e. the cross-entropy with itself) and the cross-entropy. KL-divergence is simply a conventional adjustment to cross-entropy, as it would make sense that the distance between identical distributions is 0, but $H[P(x), P(x)] = H[P(x)] \neq 0$, so $H[P(x)]$ is subtracted to adjust the cross-entropy to make a more intuitive distance measure.

4.3.1 Mutual Information

The mutual information describes how much information knowing one distribution gives us about another. Intuitively, the mutual information between a joint distribution of independent variables must be 0, as if they are independent, knowing the distribution of the first tells us nothing about the second. Therefore, also intuitively, it must be the case that the mutual information is greater the higher the absolute value of the correlation coefficient between the random variables. It makes sense then that the mutual information is defined as the statistical distance between the joint distribution of two random variables and the product of their individual distributions - if they are independent then the joint distribution is equivalent to the product of their unique distributions, and thus the statistical distance is 0, but the more dependent they are on each other, the further the product is from the joint distribution, and hence the greater the (positive semi-definite) KL-divergence. Mathematically therefore the MI is:

$$I(X, Y) \equiv I(Y, X) := D_{KL}\left[P(X, Y) \parallel P(X)P(Y)\right] = \sum_x \sum_y P(x, y) \ln\left(\frac{P(x, y)}{P(x)P(y)}\right) \tag{18}$$

An alternative and equivalent way to think about the mutual information is the difference between the entropy of a single distribution and the entropy of the conditional distribution - 'what is the difference of information between one distribution and the distribution given another variable's knowledge?'

$$I(X, Y) := H[P(X)] - H[P(X|Y)] \equiv H[P(Y)] - H[P(Y|X)] \tag{19}$$

Aside: Interestingly, minimising the Kullback-Leibler divergence between a predicted distribution and observational data is asymptotically identical to maximising the Likelihood function, and is therefore also an effective method of finding the optimal parameters to fit data to a distribution. The proof of this is as follows:
For simplicity, let us assume that we are already using the correct distribution, just with non-optimal parameters: $P(x|\theta)$, and that the true distribution with optimal parameters is: $P(x|\theta_0)$. With this set-up, the problem of finding the optimal parameters becomes one of making the estimate distribution as close to the true distribution - i.e. making $D_{KL}\left[P(x|\theta_0) \parallel P(x|\theta)\right]$ as close as possible to 0 (as the KL-divergence is semi-positive-definite, this is equivalent to minimising the KL-divergence). Therefore:

$$\begin{aligned} D_{KL}\left[P(x|\theta_0) \parallel P(x|\theta)\right] &= E_{x\sim P(x|\theta_0)}\left[\ln\left(\frac{P(x|\theta_0)}{P(x|\theta)}\right)\right] \\ &= E_{x\sim P(x|\theta_0)}\left[\ln(P(x|\theta_0)) - \ln(P(x|\theta))\right] \\ &= E_{x\sim P(x|\theta_0)}\left[\ln(P(x|\theta_0))\right] - E_{x\sim P(x|\theta_0)}\left[\ln(P(x|\theta))\right] \\ &= H[P(x,\theta_0)] - E_{x\sim P(x|\theta_0)}\left[\ln(P(x|\theta))\right] \end{aligned} \tag{20}$$

The left hand side of the subtraction in the last line is the entropy of the true distribution, but does not depend on the optimisable variable θ, and can thus be ignored in the minimisation of the KL-divergence. Next, by using the law of large numbers (the frequentist mean of a distribution of N samples approaches the statistical mean as $N \to \infty$), we can simplify further:

$$\begin{aligned} D_{KL}\left[P(x|\theta_0) \parallel P(x|\theta)\right] &\overset{N\to\infty}{=} H[P(x,\theta_0)] - \frac{1}{N}\sum_{i=1}^{N}\ln(P(x_i|\theta)) \\ &= H[P(x,\theta_0)] - \frac{1}{N}\ln\left(\prod_{i=1}^{N}P(x_i|\theta)\right) \\ &= H[P(x,\theta_0)] - \frac{1}{N}\mathcal{L}(\theta) \text{ iff } x_i \text{ i.i.d.} \end{aligned} \tag{21}$$

Here we have used the definition of the log-likelihood:

$$\mathcal{L}(\theta) := \ln\left[P(x_1, x_2, ..., x_N|\theta)\right] \tag{22}$$

and the fact that the data x_is must be i.i.d. for the probability chain rule to simplify in this way.
The final part of the proof then comes by realising that as N and $H[P(x,\theta_0)]$ are constant w.r.t. θ, then minimising the above simplified large-N expression for the KL-divergence is the same as maximising the expression for the log-likelihood because of the relative sign-change:

$$\min_\theta D_{KL}\left[P(x|\theta_0) \parallel P(x|\theta)\right] \equiv \frac{1}{N}\min_\theta\left(-\mathcal{L}(\theta)\right) = \max_\theta \mathcal{L}(\theta) \text{ Q.E.D.} \tag{23}$$

4.4 The Kalman Filter

Used for the rolling update of a parameter from continually sourced observations, for example, a moving object reading its environment at discrete time-steps.

The process involves a 'motion step' and a sensory step. This 'motion' decreases the confidence in the estimated parameter, and thus the new *a priori* distribution is flattened prior to the subsequent sensing step. Similar to Bayesian Inference, the sensory step uses observed data along with the observation error, to update the *a priori* distribution to an *a posteriori* distribution.

The Kalman filter algorithm computes an expected value for the parameter with a corresponding error, and then the Kalman filter itself is a gate that dictates how much a subsequent observation changes that estimation - ie. if the observation error is far less than the estimation error, then the Kalman gate puts greater weighting to the observation, and vice versa.

The algorithm is as follows:

1. **Motion -** where we'd expect it to be next time-step:

$$\overrightarrow{x}_t = \boldsymbol{A}\overrightarrow{x}_{t-1} + \boldsymbol{B}\overrightarrow{u}_{t-1} \tag{24}$$

$$\boldsymbol{P}_t = \boldsymbol{A}\boldsymbol{P}_{t-1}\boldsymbol{A}^T \tag{25}$$

$$\text{e.g. } \boldsymbol{A} = \begin{pmatrix} 1 & \delta t \\ 0 & 1 \end{pmatrix}, \ \boldsymbol{B} = \begin{pmatrix} \frac{1}{2}a(\delta t)^2 \\ a(\delta t) \end{pmatrix} \tag{26}$$

   Note how the covariance matrix changes with motion (covariance can even be created), due to the possible dependence of the elements of $\overrightarrow{x}$ on each other during motion under $\boldsymbol{A}$ and $\boldsymbol{B}$.

2. **Sensing -** How much the observation agrees with the estimate, and how that adjusts the estimate:

$$\overrightarrow{K} = \boldsymbol{P}_t\boldsymbol{H}^T\boldsymbol{S}^{-1}, \text{ for } \boldsymbol{S} = \boldsymbol{H}\boldsymbol{P}_t\boldsymbol{H}^T + R \tag{27}$$

$$\text{e.g. } \boldsymbol{H} = \begin{pmatrix} 1 & 0 \end{pmatrix} \tag{28}$$

   For S being the estimated error ($\boldsymbol{H}\boldsymbol{P}_t\boldsymbol{H}^T$) plus the measurement error (R).

   Then, with $\overrightarrow{K}$ known, $0 \leq K_i \leq 1 \ \forall i$:

$$\overrightarrow{x}_t \leftarrow \overrightarrow{x}_t + \overrightarrow{K}(y - \boldsymbol{H}\overrightarrow{x}_t) = (\hat{\boldsymbol{I}} - \overrightarrow{K}\boldsymbol{H})\overrightarrow{x}_t + \overrightarrow{K}y \tag{29}$$

$$\boldsymbol{P}_t \leftarrow (\hat{\boldsymbol{I}} - \overrightarrow{K}\boldsymbol{H})\boldsymbol{P}_t \tag{30}$$

   For the last equation above, note how $(\hat{\boldsymbol{I}} - \overrightarrow{K}\boldsymbol{H})\boldsymbol{P}_t$ is equivalent to the Gaussian distribution variance update formula for after a measurement is taken:

$$\sigma_i\sigma_j \leftarrow \frac{1}{1/(\sigma_i\sigma_j) + 1/R^2} \tag{31}$$

5 Statistics Prerequisites

5.1 Related Distributions - Jacobian Method

An analogue to the simple relation between distribution functions, derived by considering integrals of probability distributions over all space:

$$P(x)dx = P(y)dy \tag{32}$$

$$P(y) = \left|\frac{dx}{dy}\right| P(x) \tag{33}$$

Consider a function $f_{UV}(u,v)$ and a related distribution $f_{XY}(X,Y)$, where U,V are functions of X,Y. By the same thought process as above, we can write:

$$f_{UV}(u,v)dudv = f_{XY}(x,y)dxdy \tag{34}$$

and by the laws of changing coordinate systems:

$$dxdy = Jdudv \tag{35}$$

$$J = \left|\begin{bmatrix} \partial x/\partial u & \partial x/\partial v \\ \partial y/\partial u & \partial y/\partial v \end{bmatrix}\right| = \frac{\partial(x,y)}{\partial(u,v)} \tag{36}$$

Therefore we can see that:

$$\boxed{f_{UV}(u,v) = \frac{\partial(u,v)}{\partial(x,y)}^{-1} f_{XY}(x,y)} \tag{37}$$

Here we use the 'inverse of the inverse' Jacobian, to reflect how U,V are expressed as functions of X,Y and not the other way round.

Example: X and Y independent variables, with $Z = XY$. In order to use the Jacobian method, we require 4 variables, this is simply obtained by introducing the fourth, $W = Y$.

$$Z = XY,\ W = Y \tag{38}$$

$$\frac{\partial(z,w)}{\partial(x,y)} = \left|\begin{bmatrix} y & x \\ 0 & 1 \end{bmatrix}\right| = y \tag{39}$$

$$f_{ZW}(z,w) = \frac{1}{y} f_{XY}(x,y) = \frac{f_{XY}(z/w,w)}{w} \tag{40}$$

and due to independence:

$$f_{ZW}(z,w) = \frac{f_X(z/w)f_Y(w)}{w} \tag{41}$$

Integrating over the marginal distribution $f_{ZW}(z,w) = f_Z(z)f_W(w)$:

$$f_Z(z) = \int_{-\infty}^{\infty} \frac{f_X(z/w)f_Y(w)}{w} dw \tag{42}$$

5.2 Moment Generating Functions

The moment generating function for a random variable is defined as:

$$M_X(t) = E[e^{tX}],\ t \in \Re \tag{43}$$

$$\text{Thus } M_X(t) = \sum_{k\in X} e^{tk} P_X(k) \text{ (Disc.)} = \int_{-\infty}^{\infty} e^{tk} P_X(k)dk \text{ (Cont.)} \tag{44}$$

The next trick uses the fact that $e^x = 1 + x + x^2/2! + ...$, and expresses the exponential Moment Generating Function as a polynomial sum, where the coefficient of the k^{th} power is the k^{th} moment over $k!$:

$$M_X(t) = E[e^{tX}] = E\left[\sum_{k=0}^{\infty} \frac{(tX)^k}{k!}\right]$$
$$= \sum_{k=0}^{\infty} E\left[\frac{X^k t^k}{k!}\right] = \sum_{k=0}^{\infty} E[X^k]\frac{t^k}{k!} = \sum_{k=0}^{\infty} \mu_k \frac{t^k}{k!} \tag{45}$$

Therefore, if we compare this to the n^{th} term of the Maclaurin series of $M_X(t)$, we see that:

$$\boxed{\mu_n = E[X^n] = \tilde{M}_X^{(n)}(0)} \tag{46}$$

Special Case:

Let $Y = \sum_i X_i$ be a random variable, for X_i i.i.d. random variables. The MGF of Y can be derived as follows:

$$M_Y(t) = E[e^{tY}] = E[e^{t(X_1+X_2+\ldots)}] = E\left[\prod_i e^{tX_i}\right] = \prod_i E\left[e^{tX_i}\right] \tag{47}$$

$$M_Y(t) = \prod_i \int_{-\infty}^{\infty} e^{tx} P_{X_i}(x)dx = \prod_i M_{X_i}(t) \tag{48}$$

Note that at the end of the first line above, the fact that the $X_i s$ are independent is used, as the expectation of the product is equal to the product of the expectations for independently distributed variables.

This also gives a method of finding $f_Y(y)$, as if $M_Y(t)$ is found, an inverse method can be applied. Importantly, we can see from the integral definition of $M_Y(t)$ that the expectation is analogous to a Laplace or Fourier transform, if t is suitably substituted, e.g.:

$$M_Y(t) = \int_{-\infty}^{\infty} e^{tk} f_Y(k)dk = \int_{-\infty}^{\infty} e^{-sk} f_Y(k)dk = \bar{f}_Y(s) = \bar{f}_Y(-t) \tag{49}$$

Thus, we need to raplce t with $-s$ in the MGF, and then inverse Laplace transform to find the distribution function.

5.3 Probability Generating Functions

Similarly to MGFs, another representation for discrete distributions is the PGF, defined by (for as long as t is chosen such that the sum converges):

$$G_X(t) := E\left[t^x\right] = \sum_{x\in X} t^x P(x) \ \forall \ t \in \mathcal{R} \tag{50}$$

The PGF can be related to the MGF by replacing t in the PGF with e^t:

$$G_X(e^t) = E\left[(e^t)^x\right] = E\left[e^{tx}\right] = M_X(t) \tag{51}$$

And then of course by converse, if t in the MGF is replaced by $\ln(t)$:

$$M_X(\ln(t)) = E\left[e^{x\ln(t)}\right] = E\left[t^x\right] = G_X(t) \tag{52}$$

A couple of useful properties from PGFs are the following (given without proof):

1. The probability density function can be returned by finding the derivatives (note that the PGF is only defined for discrete distributions over integers):

$$P(X = k) = \frac{G^{(k)}(0)}{k!} \tag{53}$$

2. The k^{th} raw moment of X is given by:

$$E\left[x^k\right] = \left(z\frac{d}{dz}\right)^k G(z)\Bigg|_{z=1^-} \tag{54}$$

5.4 Maximum Likelihood Estimation

The Maximum Likelihood Estimation takes *mutually independent* observed data $\{X_i\}_{i=1...n}$ distributed according to some known distribution function with unknown parameters, and finds the values of the parameters that maximise the probability of obtaining the observed data. Therefore, we define the likelihood as the probability of obtaining all observed data given a distribution and the unknown parameters (assuming independence):

$$\boxed{L(\{\theta\}) = P(X_1 = x_1, X_2 = x_2, ...|\{\theta\}) = \prod_{i=1}^{n} f(x_i|\{\theta\})} \tag{55}$$

$$\frac{\partial}{\partial \theta_i} L(\{\theta\}) = 0 \tag{56}$$

For mathematical simplicity, it is instructive to take the natural logarithm of the likelihood function first and then maximise. Note that this will not affect the optimal parameter choice, sa $x \to \ln(x)$ is a one-to-one mapping of an increasing function to an increasing function.

$$\frac{\partial}{\partial \theta_i} \ln\left(L(\{\theta\})\right) = 0 \tag{57}$$

5.4.1 The Expectation-Maximisation Algorithm

In practice, it is often difficult to find the above stationary point exactly, and easier to make a series of approximations and optimisations via the EM algorithm. Let us imagine that there is a purely Gaussian and linear process (which is to say that every stochastic cog in the linear process follows as Gaussian distribution, and therefore the output distribution will also be Gaussian). If the process is time dependent, then the a linear system can be mathematically described as follows:

$$\begin{aligned} \vec{x}_{t+1} &= A\vec{x}_t + \vec{w}_. \text{ s.t. } \vec{w}_. \sim N(0, Q) \\ \vec{y}_t &= C\vec{x}_t + \vec{v}_. \text{ s.t. } \vec{v}_. \sim N(0, R) \end{aligned} \tag{58}$$

where A and C are simply set to 0 if the process is time-independent. In this model, A, C, $\vec{w}_.$ and $\vec{v}_.$ are the (stationary) stochastic parameters, which will be referred to as $\theta(\equiv \{\theta\})$.
Often the purpose of such a model is either to predict the parameters, θ, that cause the observations, $\vec{y}_t$, or to use these observations to infer the distribution of the inputs $\vec{x}_t$ if the model parameters are known. For either use, the first step is to approximate the parameters, and that is what this method does. The log-likelihood of some output given the model parameters is defined as:

$$\mathcal{L} := \ln\left(P(\vec{y}_t|\theta)\right) = \ln\left(\int_{\mathbf{x}} P(\vec{y}_t, \vec{x}_t|\theta) d\mathbf{x}\right) \equiv \ln\left(\int_{\mathbf{x}} \frac{P(\vec{y}_t, \vec{x}_t|\theta)}{Q(\vec{x}_t)} Q(\vec{x}_t) d\mathbf{x}\right) \tag{59}$$

for any distribution $Q(\vec{x}_t)$ (see section (6.4)). Jensen's inequality, relating a convex function of an integral to an integral of a convex function, gives a lower bound for the above expression, and therefore gives a means of approximating the expression to:

$$\mathcal{L} \geq \int_{\mathbf{x}} Q(\vec{x}_t) \ln\left(\frac{P(\vec{y}_t, \vec{x}_t|\theta)}{Q(\vec{x}_t)}\right) dx \equiv \int_{\mathbf{x}} Q(\vec{x}_t) \ln\left(P(\vec{y}_t, \vec{x}_t|\theta)\right) dx - \int_{\mathbf{x}} Q(\vec{x}_t) \ln\left(Q(\vec{x}_t)\right) dx \equiv \mathcal{F}(Q, \theta) \tag{60}$$

Because $Q(\vec{x}_t)$ is a fixed distribution with no changing parameters, it plays no part in the maximisation of $\mathcal{L}$, and therefore the subtracted part of the above expression be effectively ignored in the optimisation method. The EM algorithm is then basically a hill-climb, where firstly a $Q(\vec{x}_t)$ is chosen while holding θ constant to give an approximation for $\mathcal{L}$, and then a new value for θ is chosen whilst holding $Q(\vec{x}_t)$ constant that increases the lower bound for $\mathcal{L}$ (and hence approaches the maximum). The iteration is terminated once the parameters stop changing significantly between runs, or when a set number of runs is achieved. Mathematically then, this can be done as follows:

1. **E-step**: This step is to choose a distribution for $Q(\vec{x}_t)$, and clearly as we are trying to maximise $\mathcal{L}$, a sensible choice is the maximum value that $Q(\vec{x}_t)$ can take for a fixed value of θ - which is when the inequality becomes an equality:

$$\boxed{\mathcal{L} = \mathcal{F}(Q_{k+1}, \theta_k) \text{ iff } Q_{k+1} = P(\vec{x}_t|\vec{y}_t, \theta)} \tag{61}$$

Proof: If $Q_{k+1} = P(\vec{x}_t|\vec{y}_t, \theta)$ then $\mathcal{F}$ becomes (NB: in second line, note that a pdf always has to integrate to 1, regardless of pdf parameters):

$$\begin{aligned}\mathcal{F}(Q_{k+1}, \theta_k) &= \int_{\mathbf{x}} P(\vec{x}_t|\vec{y}_t, \theta) \ln\left(\frac{P(\vec{y}_t, \vec{x}_t|\theta)}{P(\vec{x}_t|\vec{y}_t, \theta)}\right) dx = \int_{\mathbf{x}} P(\vec{x}_t|\vec{y}_t, \theta) \ln\left(P(\vec{y}_t|\theta)\right) dx \\ &= \ln\left(P(\vec{y}_t|\theta)\right) \int_{\mathbf{x}} P(\vec{x}_t|\vec{y}_t, \theta) dx = \ln\left(P(\vec{y}_t|\theta)\right) = \mathcal{L}, \text{ Q.E.D.}\end{aligned} \tag{62}$$

This choice is perhaps less arbitrary than it seems, as if the maximum of $Q(\vec{x}_t)$ was not chosen, then there would still be an inequality ($\mathcal{L} \geq \mathcal{F}(Q_{k+1}, \theta_k)$), and therefore in the next step where θ is varied, a value of θ_{k+1} can be chosen that increases $\mathcal{F}(Q_{k+1}, \theta_{k+1})$ without increasing $\mathcal{L}$ (and therefore without improving our parameter estimate).

2. **M-step**: The next step then chooses θ_{k+1} while holding Q_{k+1} constant, and as a result of the last step providing an equality with $\mathcal{L}$, we are guaranteed in this step to increase $\mathcal{L}$ for *any* new choice of θ_{k+1} (although the one that does the most maximisation is preferable):

$$\boxed{\theta_{k+1} \leftarrow \arg\max_{\theta} \mathcal{F}(Q_{k+1}, \theta_k) = \arg\max_{\theta} \int_{\mathbf{x}} P(\vec{x}_t|\vec{y}_t, \theta_k) \ln\left(P(\vec{y}_t, \vec{x}_t|\theta)\right) dx} \tag{63}$$

5.5 Monte-Carlo and Importance Sampling

The expectation value of some function of interest, $f(x)$ can be expressed as:

$$\boxed{E[f(x)] = \int_{-\infty}^{\infty} f(x)P(x)dx \approx \frac{1}{N}\sum_{n=1}^{N} f(X_n)} \tag{64}$$

with a convergence of $1/\sqrt{N}$ by the Central Limit Theorem (for the summation of independent random variables).

Therefore, if we wish to estimate the expectation value for $f(x)$ computationally, we can randomly select a set values for X_n in a 'dart-board' like fashion, and take the mean of the $f(X_n)$s. However, for any sort of efficient convergence, it is instructive to choose the values of X_n that occur more often (i.e. have a higher probability of occurring). Therefore, the values for X_n are chosen according to a *sampling distribution*, $P(x)$. This way the most expected X_ns are chosen, and thus each $f(X_n)$ is likely to be closer to the mean.

Sampling from a probability distribution:

The simplest (and most naive) approach is to use another 'dart-like' method to sample. Here a scalar value, m, is chosen that is understood to be greater or equal to the maximal value of the sampling distribution. Two values, one of x and one of $r \in [0,1]$ are then *uniformly* selected, and if $r \cdot m \leq P(x_i)$, then that value of x is added to the output sample. This process is repeated until the desired number of sampled points are generated.

A still simple, but less naive, approach is to change the specification for selecting m, and instead let m be chosen from a readily distributable function (i.e. one with preloaded packages), *for as long as every point on this distribution is greater than or equal to the value of the sampling distribution at that point.* This alone would break the maths, so we need to adapt the choice of r... THINK OF SOLUTION

Importance sampling (for difficult distributions):

Occasionally it might be that the distribution we wish to sample from is very peaked at one particular value, and thus the other values hardly ever get chosen by the simple sampling approach described above, and perhaps for the particular use, this is a problem. In this situation, we can 'persuade' the sampling to favour a particular region for x by introducing a second distribution, $Q(x)$.

Recall from above:

$$E_P[f(x)] = \int_{-\infty}^{\infty} f(x)P(x)dx \approx \frac{1}{N}\sum_{n=1}^{N} f(X_n) \tag{65}$$

Therefore:

$$\boxed{E_Q[f(x)] = \int_{-\infty}^{\infty} \frac{f(x)P(x)}{Q(x)} Q(x)dx \approx \frac{1}{N}\sum_{n=1}^{N} \frac{f(X_n)P(X_n)}{Q(X_n)}} \tag{66}$$

Thus we can recover the expectation value for $f(x)$, but this time, with respect to the distribution function $Q(x)$, which could give preference to values very rarely chosen by $P(x)$.

Notice also how the variance can be improved by careful selection of $Q(x)$ (in particular $f(x)P(X) \approx Q(x)$):

$$\sigma_Q^2(f(x)) = E_Q[f(x)^2] - E_Q[f(x)]^2 = \int_{-\infty}^{\infty} \frac{[f(x)P(x)]^2}{Q(x)} dx - E_Q[f(x)]^2 \tag{67}$$

5.6 Bilinearity of Variance and Covariance

$$\text{Cov}(X, Y) := E[XY] - E[X]E[Y] \tag{68}$$

Notice that by the definition of independence of two variables, $E[XY] = E[X]E[Y]$, and the above covariance becomes equal to 0.

The bilinear property means that the covariance is linear in both variables, such that, for examples:

$$\begin{aligned}\text{Cov}(aX + bY, cZ) &= \text{Cov}(aX, cZ) + \text{Cov}(bY, cZ)\\ &= ac \cdot \text{Cov}(X, Z) + bc \cdot \text{Cov}(Y, Z)\end{aligned} \tag{69}$$

A consequence of this is the expression for the covariance of a sum of random variables, which expands similarly to how a binomial expansion does:

$$\boxed{\begin{aligned}\text{Var}\left(\sum_{i=1}^{n} a_i X_i\right) &= \text{Cov}\left(\sum_{i=1}^{n} a_i X_i, \sum_{j=1}^{n} a_j X_j\right) = \sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j \text{Cov}(X_i, X_j)\\ &= \sum_{i=1}^{n} a_i^2 \text{Var}(X_i) + 2\sum_{i<j} a_i a_j \text{Cov}(X_i, X_j)\end{aligned}} \tag{70}$$

5.7 Compound Distributions

Toss two coins, and for each head that comes up, throw a die. What is the distribution of the total? The possible totals can range from 0, which occurs if you toss two tails, to 12, which occurs if you toss two heads, and get a 6 on each of the two throws of the die, but what is the distribution?

If a random variable is a distribution over two random variables then its distribution is known as a compound distribution. An example is a sum of stochastic variables where the number of variables in the sum is also a stochastic distribution, this can be donated by:

$$S = \langle X, N\rangle = X_1 + X_2 + ... + X_N \tag{71}$$

The moments of this compound distribution can be found explicitly by exploiting the properties of the distributions of X_i, for example the first moment (the mean) can be derived as:

$$\begin{aligned}E[S] &= \sum_{n=1}^{n} E\left[S|N = n\right] P(N = n)\\ &= \sum_{n} nE\left[X|N = n\right] P(n) = \sum_{n} nE\left[X\right] P(n) = E\left[X\right] E\left[N\right]\end{aligned} \tag{72}$$

where in the last line the fact that X is independent of N, and the statistical definition of a discrete mean has been used.
Similarly, further characteristics like the variance can be derived:

$$\begin{aligned}
E[S^2] &= \text{Var}\,(S) + E[S]^2 \\
&= \sum_n P(n) \left\{ \text{Var}\,(S|N=n) + E[S|N=n]^2 \right\} \\
&= \sum_n P(n) \left\{ n\text{Var}\,(X) + n^2 E[X]^2 \right\} \\
&= E[N]\text{Var}(X) + E[N^2]E[X]^2
\end{aligned} \tag{73}$$

where the third line above requires both that X_is are independent to each other, and to N.

5.7.1 MGFs of Compound Distributions

The same marginalisation technique can be used to obtain the MGFs of S, by exploiting the MGF of $E[e^{tS}|N = n]$. Following through the algebra, we obtain:

$$E\left[e^{tS}|N=n\right] = E\left[e^{t(X_1+X_2+...+X_n)}\right] = E\left[e^{tX_1}\right] E\left[e^{tX_2}\right] ...E\left[e^{tX_n}\right] \tag{74}$$

Above the first equality requires that all X_is are independent of N and the second equality requires the X_is to once again be independent of each other. By comparing the last expression to the definition of the MGF $M_X(t)$ given in (6.2), it is clear to see that the above expression can be rewritten as:

$$E\left[e^{tS}|N=n\right] = M_{X_1}(t)M_{X_2}(t)...M_{X_n}(t) \tag{75}$$

If we assume that all X_is are (independent and) identically distributed, we can immediately conclude that their MGFs are identical (as an MGF is a one-to-one mapping from a distribution), and thus go one step further to say:

$$E\left[e^{tS}|N=n\right] = M_X(t)^n \tag{76}$$

Therefore, using the usual marginalisation technique, the result is obtained:

$$\boxed{M_S(t) = \sum_{n=0}^{\infty} E\left[e^{tS}|N=n\right] P(N=n) = \sum_n M_X(t)^n P(n)} \tag{77}$$

5.8 Raw and Central Moments

The *raw* n^{th} moment of a random variable X is defined to be:

$$\mu_n := E\left[X^n\right] = \int_{x\in X} x^n P(x)dx \equiv \sum_{x\in X} x^n P(x) \tag{78}$$

Similarly, the n^{th} *central* moment is the same, just first normalised to the mean first:

$$\hat{\mu}_n := E[(x - E[X])^n] = \int_{x\in X} (x - E[X])^n P(x)dx \equiv \sum_{x\in X} (x - E[X])^n P(x) \tag{79}$$

Some special examples of moments are the **skewness** and **kurtosis**, both of which are used to quantify the departure of a distribution from the Gaussian distribution, through different asymmetry or peaks respectively. They are defined as follows:

1. **Coefficient of Skewness**:

$$\gamma_1 := \frac{\hat{\mu}_3}{\sigma^3} \equiv \frac{E[X^3] - 3E[X]E[X^2] + 2E[X]^3}{\left(E[X^2] - E[X]^2\right)^{3/2}} \tag{80}$$

A positively skewed distribution has $\gamma_1 > 0$, and has a tail to the right of the mean. Conversely, a negatively skewed distribution has $\gamma_1 < 0$ and a tail to the left of the mean. Clearly a Gaussian distribution therefore has $\gamma_1 = 0$ due to its perfect symmetry about the mean.

2. **Coefficient of Kurtosis**:

$$\gamma_2 := \frac{\hat{\mu}_4}{\sigma^4} \equiv \frac{E[X]^4 - 4E[X^3]E[X] + 6E[X^2]E[X]^2 - 3E[X]^4}{\left(E[X^2] - E[X]^2\right)^2} \tag{81}$$

This measure describes how peaked the centre of the distribution is, with $\gamma_2 > 3$ representing more peaked than a Gaussian, and $\gamma_2 < 3$ being less peaked (i.e. flatter) than a Gaussian - this is because the kurtosis of a Gaussian is $\gamma_2 = 3$.

6 Linear Gaussian Models

6.1 Static LGMs

Special Principal Component Analysis (SPCA), PCA and Factor Analysis (FA) are all variants of an overarching model called the (Static) Linear Gaussian Model (LGM), as described in (6.3.1). The model is repeated here for convenience, and is:

$$\begin{aligned} \vec{x}_{t+1} &= A\vec{x}_t + \vec{w}_. \text{ s.t. } \vec{w}_. \sim N(0, Q) \\ \vec{y}_t &= C\vec{x}_t + \vec{v}_. \text{ s.t. } \vec{v}_. \sim N(0, R) \end{aligned} \tag{82}$$

As explained in (6.3.1), each realisation of the process at each discrete time step is Gaussian as the model is a linear combination of Gaussian processes, and Gaussianity is conserved in linear mappings. The above equations therefore satisfy the requirements of a *first-order Gauss-Markov process*, which is when the distribution of a random variable at time step t depends only on the value of the variable at the previous time step, and not on any prior information - i.e.:

$$P(\vec{x}_{t+1}|\vec{x}_t, \vec{x}_{t-1}, ..., \vec{x}_1) = P(\vec{x}_{t+1}|\vec{x}_t) = P(A\vec{x}_t + \vec{w}_.|\vec{x}_t) = \mathcal{N}(A\vec{x}_t, Q) \tag{83}$$

$$P(\vec{y}_t|\vec{x}_t, \vec{x}_{t-1}, ..., \vec{x}_1) = P(\vec{y}_t|\vec{x}_t) = P(C\vec{x}_t + \vec{v}_.|\vec{x}_t) = \mathcal{N}(C\vec{x}_t, R) \tag{84}$$

In the equations above, A and C are definite matrices (representing linear mappings), and therefore all of the variance is caused by the uncertainty parameters Q and R. In the case that each time step is independent of all other time steps, the above equations simplify to a *static* LGM (rather than a *dynamic* one), and the matrix A becomes the zero matrix:

$$\begin{aligned} \vec{x}_. &\sim \mathcal{N}(0, Q) \\ \vec{y}_. &\sim \mathcal{N}(0, CQC^T + R) \end{aligned} \tag{85}$$

where the CQC^T is simply the way of transforming the uncertainty in $\vec{x}_.$ space into $\vec{y}_.$ space under a linear mapping via C (when vectors are mapped by a matrix, matrices are pre- and post- multiplied by the matrix and its transpose).

Aside: one way to justify treating the covariance matrix Q in this way is by manualling expressing the covariance of $\vec{y}_.$:

$$\begin{aligned} \text{Cov}(\vec{y}_.) &= \text{Cov}(C\vec{x}_. + \vec{v}_.) = E\left[(C\vec{x}_. - C\mu_x)(C\vec{x}_. - C\mu_x)^T\right] + R \\ &= E\left[C(\vec{x}_. - \mu_x)(\vec{x}_. - \mu_x)^T C^T\right] + R = CE\left[(\vec{x}_. - \mu_x)(\vec{x}_. - \mu_x)^T\right] C^T + R \\ &= CQC^T + R, \text{ Q.E.D.} \end{aligned} \tag{86}$$

We now have degeneracy here, as it does not matter whether the uncertainty is represented in C or in Q, so a convenient convention is to fix the degeneracy by diagonalising Q (and then projecting that diagonalisation onto C):

$$\begin{aligned} &Q = EDE^T, \text{ then: } Q' = I, \ C' = CED^{1/2} \\ &\text{s.t. } CQC^T = C(EDE^T)C^T = (CED^{1/2})I(CED^{1/2})^T = C'Q'C'^T \end{aligned} \tag{87}$$

With degeneracy fixed, the static LGMs can then be separated into different classes by restricting the value of the noise matrix R (note that if R was left completely unrestricted then in learning, the algorithm would simply set $C = 0$ in learning and set R to represent all of the sample covariance in the data, thus looking effectively like a 'perfect fit' whilst giving no insight into the model parameters).

The classes of classification algorithm can be identified as follows:

1. Principal Component Analysis (PCA): $Q = I$ and $R = \lim_{\epsilon \to 0} \epsilon I$.

   An infinitesimal R means that all of the variance and covariance in $\vec{y}_.$ now has to be explained by the covariance present in $\vec{x}_.$ - in other words, any covariance in $\vec{x}_.$ space is projected into $\vec{y}_.$ space via C, and therefore if we are to find the C that minimises the covariance in $\vec{y}_.$ space, we will have found the projection matrix into the principal components of $\vec{x}_.$.

Note: PCA is not technically a statistical model as it has no variance or covariance in the principal axes space (i.e. in $\vec{y}_.$ space) due to R effectively being 0.

2. Sensible Principal Component Analysis (SPCA): $Q = I$ and $R = \alpha I$ for $\alpha \in \mathcal{R}$.

   Restricting R to be proportional to the diagonal matrix means that, in the space spanned by $\vec{y}_.$, the noise is the same in each axis, and there is no covariance between axes in $\vec{y}_.$'s space that is caused by noise - i.e. any covariance in $\vec{y}_.$ *must* be due to covariance in $\vec{x}_.$, but with added flexibility compared to PCA, that there can be some added noise in $\vec{y}_.$ (sensor noise).

3. Factor Analysis (FA): $Q = I$ and $R \in \text{diag}(\cdot)$.

   Factor analysis tries to explain links between components in $\vec{x}_.$ by putting all coordinate unique variance in $\vec{y}_.$ into R, and putting information relating to correlation structure into C. An effect of this is invariability to rescaling in $\vec{y}_.$, as that would change the variance and not the covariance, thus changing R only. PCA and SPCA do not have this freedom as R is fixed to have equal or zeros variance.

The learning algorithm:

Firstly, it is helpful to use the wholly Gaussian nature of this model to undertake inference, i.e. to find the probability distribution of the inputs given the observed outputs (and knowledge of the model parameters θ), or mathematically, in the fixed-degeneracy convention:

$$P(\vec{x}_.|\vec{y}_.,\theta) = \frac{P(\vec{y}_.|\vec{x}_.,\theta)P(\vec{x}_.|\theta)}{P(\vec{y}_.|\theta)} = \frac{\mathcal{N}(C\vec{x}_.,R)\mathcal{N}(0,I)}{\mathcal{N}(0,CC^T+R)} = \mathcal{N}(\beta\vec{y}_.,I-\beta C) \tag{88}$$

for $\beta = C^T(CC^T+R)^{-1}$, which is obtained by using the explicit forms of the multivariate Gaussian distribution PDFs. The likelihood is also easily expressed as the denominator of the above expression (for $Q := I$):

$$P(\vec{y}_.|\theta) = \mathcal{N}(0, CC^T + R) \tag{89}$$

The EM algorithm, as discussed in (5.4) now gives a means of learning the parameters C and R that maximise the (log)-likelihood.

Example: Classifying symptoms to psychological disorders

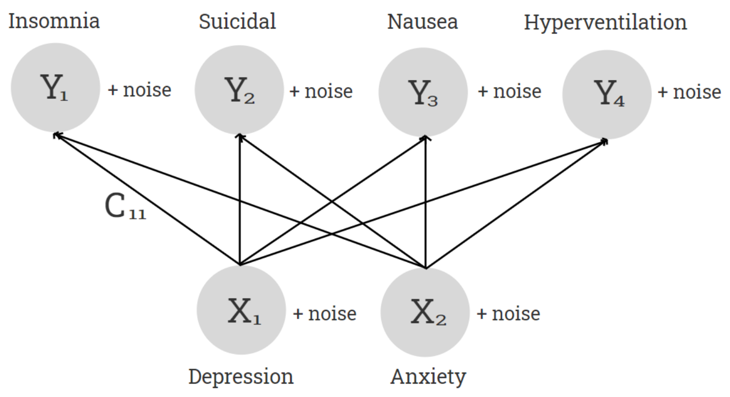


Figure 2: An example of a static LGM in a psychological context. The LGM is static as the population distribution of underlying disorders is assumed constant with time, and the symptoms observed under psychiatric assessment may have correlations that can be explained by the dependency on a smaller set of disorders. Note that if the sensor noise is the same, we have SPCA, or if it is different, we have FA.

7 Markov Models and Hidden Markov Models

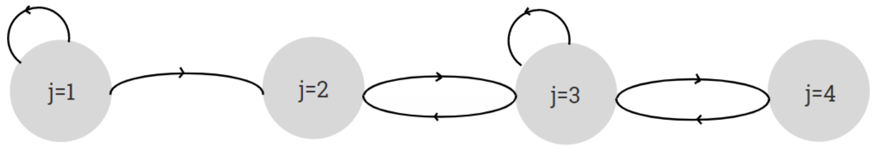


Figure 3: An example of a static (or time independent) Markov Chain with 4 states. The arrows indicate directions in which the state can transform, and have associated probabilities that sum to 1 leaving each state.

7.1 Markov Models

A Markov model is one that relies on the Markov assumptions, that the current state of a system relies only on a selected number of previous states. Explicitly, an n^{th} order Markov model is defined as:

$$P(X_{T+1}|X_{1:T}) = P(X_{T+1}|X_T, X_{T-1}, X_{T-2}, ..., X_{T-n}) \tag{90}$$

From now on, we assume that the Markov models are first order. If we define a transition matrix A_{ij} between states then the probability of transitioning from states i to state j in n steps can be expressed as:

$$A_{ij}(n) := P(X_{t+n} = j|X_t = i) \tag{91}$$

The *Chapman-Kolgorov* equation describes how consequent steps affect each other, and is intuitively thought of as the sum over all possible paths from an earlier time step to the desired time step:

$$A_{ij}(n+m) = \sum_{k=1}^{K} A_{ik}(m)A_{kj}(n) \tag{92}$$

$$A(n+m) = A(m)A(n) \tag{93}$$

where K is the number of states available at each time step, and the second equation above is the matrix form. If we take into account the fact that $A(1) \equiv A$, then the n-step transition matrix can be simply written as a power term:

$$A(n) = A^n \tag{94}$$

As an example, the transition matrix for the chain in Figure 3 would be:

$$A = \begin{bmatrix} A_{11} & A_{12} & 0 & 0 \\ 0 & 0 & A_{23} & 0 \\ 0 & A_{32} & A_{33} & A_{34} \\ 0 & 0 & A_{43} & 0 \end{bmatrix} \tag{95}$$

where as stated above, A_{ij} is the transition probability from state i to state j: $P(X_{t+1} = j|X_t = i)$, meaning each row of A must add to 1.

7.2 Hidden Markov Models (HMMs)

Architecture: The name *Hidden* Markov Model comes from the idea that now there is a system of hidden (unobserved) states that follow the Markov assumption, which influence the distributions of a set of observed states (which may or may not themselves be Markovian). The transition between states in the Markovian system is described by a Markov Chain, such as that shown in Figure 3, but then there is an additional need to account for *emission probabilities*, the probability of an observation at each sequential time step given the Hidden Markov states. Graphically, a HMM will have the following architecture:
NB: Note how the graph in Figure 4 is different to the chain in Figure 3, as Figure 3's chain dictates the transition probabilities between states (S_t in this case), whereas Figure 4 describes the sequential (time-directed) relationships between hidden states and observed states, and demonstrates the Markovian property that $P(S_T|S_{1:T-1}) = P(S_T|S_{T-1})$ only.

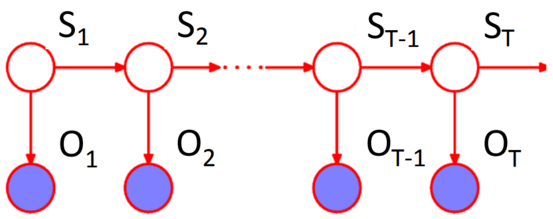


Figure 4: The graphical sequential layout of a HMM, where $\{S_t\}_{t=1}^T$ is the unobserved Markov model, creating observed data $\{O_t\}_{t=1}^T$. Figure taken from *http://compneurosci.com/wiki/images/0/0a/HMM-parisa.pdf*.

Example: The Occasionally Dishonest Casino

Imagine that there is a casino with two die, one which is fair, s.t. $S_1 \sim \{1:1/6, 2:1/6, 3:1/6, 4:1/6, 5:1/6, 6:1/6\}$, and one which is loaded towards 6, s.t. $S_2 \sim \{1:1/10, 2:1/10, 3:1/10, 4:1/10, 5:1/10, 6:1/5\}$. The probability of one dice being used is dependent only on which dice was used in the last through, and is a 0.05 probability of changing, or a 0.95 probability of staying with the same dice between throws. The players at the casino do not know which dice is being used, only the resulting numbers that come from the throws. The dependence of the type of dice only on the previous throw means that it is a Markov system, and the hidden nature of the type of dice makes this problem a HMM.

We know the transition probabilities of the 1st-order MC:

$$P(S_{t+1} = L(F)|S_t = F(L)) = 0.05, \; P(S_{t+1} = L(F)|S_t = L(F)) = 0.95 \tag{96}$$

and we know the emission probabilities given by the distributions above:

$$\begin{aligned} P(O_t = i|S_t = F) &= 1/6 \; \forall \; i \in [1,6] \\ P(O_t = i|S_t = L) &= \begin{cases} 1/10 & i \in [1,5] \\ 1/5 & i = 6 \end{cases} \end{aligned} \tag{97}$$

If we observe a sequence of dice throws, e.g. 1663254616426, there are three questions we might want to ask:

a. What is the probability of obtaining such a sequence of observations given we know the parameters of the model?

b. Which parts of the sequence were generated by the fair dice, and which were generated using the loaded dice?

c. Now assuming that we don not know how the casino works: How biased is the loaded dice and is the fair dice completely fair (i.e. what are the emission probabilities)? And how often does the casino change between them (i.e. what are the transmission probabilities)?

These questions are the subject of three algorithms, for *evaluation*, *decoding*, and *learning* respectively.

7.2.1 Evaluation

The question being answered in evaluation is ***"how likely is our observed sequence of data given a HMM of known parameters?"***

Mathematically that means that we are looking to find the joint distribution over all of the observations, which can be found explicitly through marginalising over the hidden states:

$$\begin{aligned} P(\{O_t\}_{t=1}^T) &= \sum_{\{S_t\}_{t=1}^T} P(\{O_t\}_{t=1}^T, \{S_t\}_{t=1}^T) \\ &= \sum_{\{S_t\}_{t=1}^T} \prod_{t=1}^T P(O_t|S_t) \prod_{t=2}^T P(S_t|S_{t-1}) P(S_1) \end{aligned} \tag{98}$$

where the second equation above represents a marginalisation of the broken down for of the joint distribution of the observations and hidden states (think about the graph in Figure 4 to justify, remembering that the hidden system is 1^{st}-order Markovian). However this requires summing over all of the possible states that all of the hidden system can take at every time-step, which gives an unwieldy complexity of K^T for T time steps and K possible hidden states.

Therefore, as is usual in ML algorithms, a recursive method can be used to give a faster approach. To do this, we rewrite the joint observations distribution as a different marginalisation, and define a new property $\alpha_t(k)$ s.t.:

$$P(\{O_t\}_{t=1}^T) = \sum_{k=1}^K P(\{O_t\}_{t=1}^T, S_T = k) := \sum_{k=1}^K \alpha_T(k) \tag{99}$$

Now, then, the problem boils down to finding $\alpha_T(k)$, which is where the recursion comes in, using the relation:

$$\alpha_t = P(O_t|S_t = k) \sum_{i=1}^K \alpha_{t-1}(i) P(S_t = k|S_{t-1} = i) \tag{100}$$

Proof:

First note what happens when we switch down from the t index to $t-1$:

$$\begin{aligned} \alpha_{t-1}(i) : &= P(\{O_{t'}\}_{t'=1}^{t-1}, S_{t-1} = i) \\ \therefore\ \alpha_{t-1}(i) P(S_t = k|S_{t-1} = i) &= P(\{O_{t'}\}_{t'=1}^{t-1}, S_t = k, S_{t-1} = i) \end{aligned} \tag{101}$$

where the second line above can be understood by thinking about the graph in Figure 4. So now if we marginalise over the probabilities of S_{t-1} taking each state i:

$$P(\{O_{t'}\}_{t'=1}^{t-1}, S_t = k) = \sum_{i=1}^K P(\{O_{t'}\}_{t'=1}^{t-1}, S_t = k, S_{t-1} = i) = \sum_{i=1}^K \alpha_{t-1}(i) P(S_t = k|S_{t-1} = i) \tag{102}$$

Again, by thinking about the graph in Figure 4, we can then make the final step:

$$P(\{O_{t'}\}_{t'=1}^{t}, S_t = k) = P(O_t|S_t = k) P(\{O_{t'}\}_{t'=1}^{t-1}, S_t = k) = P(O_t|S_t = k) \sum_{i=1}^K \alpha_{t-1}(i) P(S_t = k|S_{t-1} = i) \tag{103}$$

or more explicitly, in terms of the α parameter:

$$\alpha_t = P(O_t|S_t = k) \sum_{i=1}^K \alpha_{t-1}(i) P(S_t = k|S_{t-1} = i), \text{ Q.E.D.} \tag{104}$$

Evaluation (*Forwards*) algorithm: The forwards algorithm can be summarised as:

1. Initialise $\alpha_1(k) = P(O_1, S_1 = k) = P(O_1|S_1 = k) P(S_1 = k)$
2. Iterate for $t = 2...T$: $\alpha_t(k) = P(O_t|S_t = k) \sum_{i=1}^K \alpha_{t-1}(i) P(S_t = k|S_{t-1} = i)$

   (note here that $P(O_t|S_t = k)$ is simply the emission probability and is defined for a known-parameter HMM, and $P(S_t = k|S_{t-1} = i)$ is the transfer probability, and is also defined (defined by a Markov Chain like that in Figure 3).)
3. Terminate the algorithm at $t = T$, and calculate the joint distribution over observations as $P(\{O_t\}_{t=1}^T) = \sum_{k=1}^K \alpha_T(k)$

7.2.2 Decoding

Decoding now approaches a different question: ***"what are the hidden states given the observations?"***

This question sounds a bit open to interpretation, and therefore needs some clarification. With this in mind the decoding problem can be split into further subsections:

a. **Filtering:** an online (and therefore more uncertain) distribution of the current hidden state value given the observations *so far*:

$$P(S_t = k|\{O_{t'}\}_{t'=1}^{t}) \tag{105}$$

b. **Smoothing:** an offline (and therefore less uncertain) distribution of the current hidden state given *all* of the observations:

$$P(S_t = k|\{O_t\}_{t=1}^{T}) \tag{106}$$

c. **Fixed-lag filtering:** the middle-ground between filtering and smoothing, where a slight lag is taken to increase certainty of an online distribution:

$$P(S_{t-l} = k|\{O_{t'}\}_{t'=1}^{t}) \tag{107}$$

d. **Prediction:** a means of predicting states on a horizon, h, given the current sequence of observations:

$$P(S_{t+h}|\{O_{t'}\}_{t'=1}^{t}) \tag{108}$$

e. **Holistic:** finding the most likely entire sequence of hidden states after observing the complete sequence. This is the pretext behind the *Viterbi decoding algorithm* as described at the end of (7.2.2):

$$\arg\max_{\{S_t\}_{t=1}^{T}} P(\{S_t\}_{t=1}^{T}|\{O_t\}_{t=1}^{T}) \tag{109}$$

Filtering:

The simplest of them all is the filtering algorithm. Recall how, in the evaluation problem discussed in (7.2.1), the joint distribution over observations was found:

$$P(\{O_{t'}\}_{t'=1}^{t}) = \sum_{k=1}^{K} \alpha_t(k), \text{ for } \alpha_t(k) := P(\{O_{t'}\}_{t'=1}^{t}, S_t = k) \tag{110}$$

Now it is just a case of employing Bayes' rule to find the posterior:

$$P(S_t = k|\{O_{t'}\}_{t'=1}^{t}) = \frac{P(\{O_{t'}\}_{t'=1}^{t}, S_t = k)}{P(\{O_{t'}\}_{t'=1}^{t})} = \frac{\alpha_t(k)}{\sum_k \alpha_t(k)} \tag{111}$$

And it is as simple as that, when $\alpha_t(k)$ is found recursively using the forwards algorithm described in (7.2.1).

Smoothing:

This time we are concerned with finding the probability of a hidden state given all of the observations. As with the evaluation algorithm, we start with a joint distribution, which can be split into two terms through Bayes' rule:

$$P(S_t = k, \{O_t\}_{t=1}^{T}) = P(S_t = k, \{O_{t'}\}_{t'=1}^{t})P(\{O_{t'}\}_{t'=t+1}^{T}|S_t = k) \tag{112}$$

which can intuitively be thought of as the joint distribution of the probability of this current state and the observations so far with the probability of the future observations given the current state. This expression above is then redefined:

$$P(S_t = k, \{O_t\}_{t=1}^{T}) = \alpha_t(k)\beta_t(k) \tag{113}$$

$$\beta_t(k) := P(\{O_{t'}\}_{t'=t+1}^{T}|S_t = k) \tag{114}$$

If we were to know $\beta_t(k)$ (and $\alpha_t(k)$ from the evaluation algorithm) then we can build the desired smoothed distribution:

$$P(S_t = k|\{O_t\}_{t=1}^{T}) = \frac{P(S_t = k, \{O_t\}_{t=1}^{T})}{P(\{O_t\}_{t=1}^{T})} = \frac{\alpha_t(k)\beta_t(k)}{\sum_k \alpha_t(k)\beta_t(k)} \tag{115}$$

Thus, again, we have only one problem left, which is to find $\beta_t(k)$ for all t recursively. Unlike in the forwards algorithm though, this recursion is done backwards, as the starting point is $t = T$ by its definition. Because of this it is called the *backwards* algorithm, and gives the following result:

$$\beta_t(k) = \sum_{i=1}^{K} P(S_{t+1} = i|S_t = k)P(O_{t+1}|S_{t+1} = i)\beta_{t+1}(i) \tag{116}$$

Proof:

If we are to recurse backwards with starting point $\beta_T(k)$, then it makes sense to put $\beta_t(k)$ in terms of $\beta_{t+1}(k)$:

$$\begin{aligned}
&\beta_{t+1}(i) &&= P(\{O_{t'}\}_{t'=t+2}^{T}|S_{t+1}=i)\\
\therefore\ &\beta_{t+1}(i)P(O_{t+1}|S_{t+1}=i) &&= P(\{O_{t'}\}_{t'=t+1}^{T}|S_{t+1}=i)\\
\therefore\ &\beta_{t+1}(i)P(O_{t+1}|S_{t+1}=i)P(S_{t+1}=i|S_t=k) &&= P(\{O_{t'}\}_{t'=t+1}^{T}|S_{t+1}=i,S_t=k)
\end{aligned} \tag{117}$$

(Think about the graph dependencies in the HMM model of Figure 4 to understand this). And so, when the last line of Equation (117) is marginalised over i, the following result is found:

$$\sum_{i=1}^{K}\beta_{t+1}(i)P(O_{t+1}|S_{t+1}=i)P(S_{t+1}=i|S_t=k)=\sum_{i=1}^{K}P(\{O_{t'}\}_{t'=t+1}^{T}|S_{t+1}=i,S_t=k)=\beta_t(k),\ \text{Q.E.D.} \tag{118}$$

Smoothing (*Backwards*) algorithm: The backwards algorithm can be summarised as:

1. Carry out the forwards algorithm to obtain $\alpha_t(k)$ for all time steps t for all K states.
2. Initialise $\beta_T(k)=1$ for all K states (since $P(\emptyset|S_T=k)=1$).
3. Iterate for $t=T-1...1$: $\beta_t(k)=\sum_{i=1}^{K}P(S_{t+1}=i|S_t=k)P(O_{t+1}|S_{t+1}=i)\beta_{t+1}(i)$
4. Terminate the algorithm at $t=1$, and calculate $P(S_t=k|\{O_t\}_{t=1}^{T})=\frac{\alpha_t(k)\beta_t(k)}{\sum_k\alpha_t(k)\beta_t(k)}$

The time complexity to find all values of each recursive parameter (i.e. α or β) is of the order $O(K^2T)$, so a lot slower increasing in both K and T than $O(K^T)$, which is the time complexity if marginalising over every possible hidden state.

Viterbi decoding:

This algorithm is concerned with finding the most likely *sequence* of hidden states given the sequence of observations - which is *not* the same thing as finding the most likely state at each time-step (something which is easily done by finding the argmax of the filtering or smoothing algorithm). Thus the target result is the set of most probable state labels $\{S_t^*\}_{t=1}^{T}$, with the final label given by:

$$S_T^* := \arg\max_{\{S_t\}_{t=1}^{T}} P\left(\{S_t\}_{t=1}^{T}|\{O_t\}_{t=1}^{T}\right) \tag{119}$$

The benefit of finding the most likely sequence over most likely state is that finding $\{S_t^*\}_{t=1}^{T}$ is equivalent to finding the shortest path through a *trellis diagram*, as shown in Figure 5. In terms of application, looking holistically for the most probable sequence can be favourable in certain situations, like offline speech recognition, when one noisy an misinterpreted word is less likely to cause error if interpreted in the context of all other hidden states.

In order to find the argmax over the conditional distribution in Equation (119), we can equally well find it over the joint distribution due to the independence of the argmax on the observations:

$$\arg\max_{\{S_t\}_{t=1}^{T}} P(\{S_t\}_{t=1}^{T}|\{O_t\}_{t=1}^{T})=\arg\max_{\{S_t\}_{t=1}^{T}}\frac{P(\{S_t\}_{t=1}^{T},\{O_t\}_{t=1}^{T})}{P(\{O_t\}_{t=1}^{T})}=\arg\max_{\{S_t\}_{t=1}^{T}}P(\{S_t\}_{t=1}^{T},\{O_t\}_{t=1}^{T}) \tag{120}$$

Now we can break the joint distribution apart in hunt for a parameter that we can iterate over as in previous algorithms:

$$\begin{aligned}
\arg\max_{\{S_t\}_{t=1}^{T}} P(\{S_t\}_{t=1}^{T},\{O_t\}_{t=1}^{T}) &\equiv \arg\max_{S_T,\{S_t\}_{t=1}^{T-1}} P(S_T,\{S_t\}_{t=1}^{T-1},\{O_t\}_{t=1}^{T})\\
&\equiv \arg\max_k \max_{\{S_t\}_{t=1}^{T-1}} P(S_T=k,\{S_t\}_{t=1}^{T-1},\{O_t\}_{t=1}^{T})\\
&= \arg\max_k V_T(k)
\end{aligned} \tag{121}$$

where, as is becoming commonplace now, we have defined a new iterative parameter, $V_T(k)$, which is the probability of the most probable sequence ending with hidden state k:

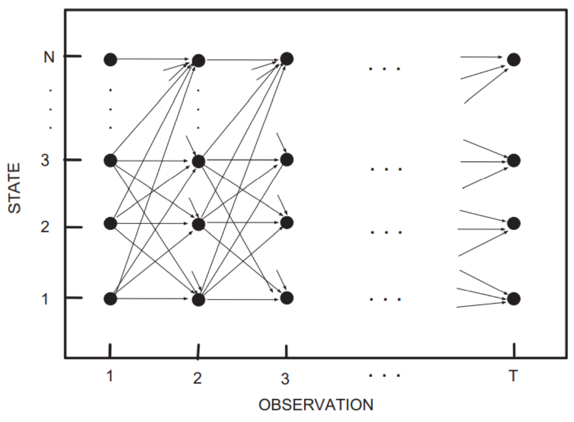


Figure 5: A trellis diagram representing all possible states of the hidden Markovian system at each observation time. Each edge has an associated (log) transfer probability determined by the transition matrix. Figure taken from Kevin Murphy's *Machine Learning: A Probabilistic Perspective* book.

$$V_T(k) := \max_{\{S_t\}_{t=1}^{T-1}} P(S_T = k, \{S_t\}_{t=1}^{T-1}, \{O_t\}_{t=1}^{T}) \tag{122}$$

If we find $V_T(k)$ then we find $\{L_t\}_{t=1}^{T}$, and we can find $V_T(k)$ through recursion using the result:

$$V_t(k) = P(O_t|S_t = k) \max_i P(S_t = k|S_{t-1} = i)V_{t-1}(i) \tag{123}$$

Proof:

Changing indices to $t-1$ we see:

$$\begin{aligned}
V_{t-1}(i) &= \max_{\{S_{t'}\}_{t'=1}^{t-2}} P(S_{t-1} = i, \{S_{t'}\}_{t'=1}^{t-2}, \{O_{t'}\}_{t'=1}^{t-1}) \\
P(S_t = k|S_{t-1} = i)V_{t-1}(i) &= \max_{\{S_{t'}\}_{t'=1}^{t-2}} P(S_t = k, S_{t-1} = i, \{S_{t'}\}_{t'=1}^{t-2}, \{O_{t'}\}_{t'=1}^{t-1}) \\
\max_i P(S_t = k|S_{t-1} = i)V_{t-1}(i) &= \max_{\{S_{t'}\}_{t'=1}^{t-1}} P(S_t = k, \{S_{t'}\}_{t'=1}^{t-1}, \{O_{t'}\}_{t'=1}^{t-1}) \\
\therefore\ P(O_t|S_t = k) \max_i P(S_t = k|S_{t-1} = i)V_{t-1}(i) &= \max_{\{S_{t'}\}_{t'=1}^{t-1}} P(S_t = k, \{S_{t'}\}_{t'=1}^{t-1}, \{O_{t'}\}_{t'=1}^{t})
\end{aligned} \tag{124}$$

$$V_t(k) = P(O_t|S_t = k) \max_i P(S_t = k|S_{t-1} = i)V_{t-1}(i), \text{ Q.E.D.}$$

Note how in the second line of Equation (124) we have used the fact that $P(S_t = k|S_{t-1} = i)$ is independent of $\{S_{t'}\}_{t'=1}^{t-2}$ to move it to the right of the max statement and combine the distributions using Bayes' rule (or intuition from Figure 4).

Now that we have the most probable last state in the hidden system, we can use that to work back and see where that last state most likely came from, as this is the previous most probable state L_{t-1}. This is done by:

$$S_{t-1}^* = \arg\max_i P(S_t^*|S_{t-1} = i)V_{t-1}(i) \tag{125}$$

Proof:

$$\begin{aligned}
P(S_t^*|S_{t-1} = i)V_{t-1}(i) &= P(S_t^*|S_{t-1} = i) \max_{\{S_{t'}\}_{t'=1}^{t-2}} P(S_{t-1} = i, \{S_{t'}\}_{t'=1}^{t-2}, \{O_{t'}\}_{t'=1}^{t-1}) \\
&= \max_{\{S_{t'}\}_{t'=1}^{t-2}} P(S_t^*|S_{t-1} = i)P(S_{t-1} = i, \{S_{t'}\}_{t'=1}^{t-2}, \{O_{t'}\}_{t'=1}^{t-1}) \\
&= \max_{\{S_{t'}\}_{t'=1}^{t-2}} P(S_t^*, S_{t-1} = i, \{S_{t'}\}_{t'=1}^{t-2}, \{O_{t'}\}_{t'=1}^{t-1})
\end{aligned} \tag{126}$$

The last line of the equation above is the joint probability of the most probable path up to S_{t-1}, with S_{t-1} taking value i, and with S_t taking its most probable value, given all of the observed data (recall that argmax of the joint of the observations is the same as argmax over the conditional of the observations as the argmax is independent of them). Therefore if we find the state i that makes S_{t-1} give Equation (126) the maximum probability, we are finding the most probable path up to S_t *and including* S_{t-1} given the observations. Hence the most most value of S_{t-1} that gives the most probable overall path is:

$$\begin{aligned} S^*_{t-1} &= \arg\max_i \max_{\{S_{t'}\}_{t'=1}^{t-2}} P(S^*_t, S_{t-1} = i, \{S_{t'}\}_{t'=1}^{t-2}, \{O_{t'}\}_{t'=1}^{t-1}) \\ &= \arg\max_i P(S^*_t | S_{t-1} = i) V_{t-1}(i), \text{ Q.E.D.} \end{aligned} \tag{127}$$

Viterbi decoding algorithm: The viterbi algorithm can be summarised as:

1. Initialise $V_1(k) = P(S_1 = k, O_1) = P(O_1|S_1 = k)P(S_1 = k)$ for all K states.
2. For $t = 2...T$: $V_t(k) = P(O_t|S_t = k) \max_i P(S_t = k|S_{t-1} = i)V_{t-1}(i)$ for each state.
3. Terminate at $t = T$, and compute $S^*_T = \arg\max_k V_T(k)$.
4. Traceback from S^*_T to the most probable states for previous time-steps through $S^*_{t-1} = \arg\max_i P(S^*_t|S_{t-1} = i)V_{t-1}(i)$

7.2.3 Learning

The final question we may want answered is ***"what are the parameters of the model, given that we have observed a sequence?"*** - assuming we don't know like we do in the example above.

As usual, we find the likelihood function, and then choose the parameters that maximise it - either automatically (if it is a simple model with low time complexity), or recursively (as we did with the LGMs):

$$\hat{\theta} \leftarrow \arg\max_\theta P(\{O_t\}_{t=1}^T|\theta) \equiv \arg\max_\theta \ln\left[P(\{O_t\}_{t=1}^T|\theta)\right] = \arg\max_\theta \mathcal{L}(\theta) \tag{128}$$

In the case of HMMs, the likelihood function does not factorise nicely as we have no requirement for the observations to be i.i.d. This makes the above expression unwieldy and so again we require the recursive method. This is done in the form of an Expectation-Maximisation algorithm, called specifically the *Baum-Welch* algorithm.

Baum-Welch (forward-backwards) algorithm:

The problem of learning is markedly more difficult than the last two problems. This algorithm tries to present a common sense solution, that we estimate the parameters the same way that a human watching a casino select dice might: we take the *frequentist* approach and ask 'how many times does the hidden system take each state over all of the time-steps?' (Or more accurately, we ask 'how many times do we *expect* the hidden system to take each state over all of the time-steps?', as we cannot directly observe the hidden system.)

This means we need frequency distributions, and the important ones are those which estimate the parameters that we are looking for, namely $\theta = \{\vec{\pi}, A, q\}$:

$$\begin{aligned} \hat{\pi}_i &= \text{expected frequency that } S_1 = i \\ \hat{A}_{ij} &= \frac{\text{expected \# of transitions leaving hidden state } i \text{ and entering hidden state } j}{\text{expected \# of total transitions out of hidden state } i} \\ \hat{q}^k_j &= \frac{\text{expected \# of times observing } k \text{ from hidden state } j}{\text{expected \# of total times being in hidden state } j} \end{aligned} \tag{129}$$

where, for brevity, we have made the following definitions, and is simply:

$$\begin{aligned} \pi_i \quad &:= P(S_1 = i|\{O_t\}_{t=1}^T, \theta) \\ A_{ij}(t) &:= P(S_{t+1} = j|S_t = i, \theta) = A_{ij}(\cdot) \text{ iff static HMM} \\ q_i^{O_t} \quad &:= P(O_t|S_t = i, \theta) \\ \gamma_i(t) &:= P(S_t = i|\{O_t\}_{t=1}^T, \theta) \\ \xi_{ij}(t) &:= P(S_t = i, S_{t+1} = j|\{O_t\}_{t=1}^T, \theta) \end{aligned} \tag{130}$$

The first parameter in Equation (129) is easiest to express:

$$\hat{\pi}_i = \gamma_i(1) = \frac{\alpha_1(i)\beta_1(i)}{\sum_k \alpha_1(k)\beta_1(k)} \tag{131}$$

Next we have the emission probability matrix, q:

$$\hat{q}_j^k = \frac{\sum_{t=1}^{T} \gamma_j(t)\delta(O_t - k)}{\sum_{t=1}^{T} \gamma_j(t)} \tag{132}$$

where $\gamma_t(j)$ is defined as in Equation (115), and the delta symbol in the numerator indicates that the probability of the hidden system being in state j is only counted if the observed system is in state k for that time-step. Notice also how the sum is over time. This is to change the probabilities to an expectation value (as the probabilities are defined as the expected number of times something happens per time-step).

Last but not least, the transition matrix must be predicted. To find the expectation as used in the relevant definition in Equation (129), we require the probability of a transition between $S_t = i$ and $S_{t+1} = j$. Note also that we need the joint distribution not the conditional here, as we are looking at the system from a top-down perspective, not a perspective where we assume that $S_t = i$ already. Therefore we use a new parameter, defined in Equation (130):

$$\hat{A}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_t(i)} \tag{133}$$

Notice above how the sum runs to $T-1$ this time, that is because we are concerned with the expected number of times *leaving* a state, not *being* in a state like in the other parameters (if $t = T$ then we cannot *leave* that state). This new parameter has a previously undefined probability though, the joint distribution of two hidden states taking certain values given the observations and parameters. Therefore it is easier to rewrite it:

$$\xi_{ij}(t) = \frac{\alpha_t(i)A_{ij}q_j^{O_{t+1}}\beta_{t+1}(j)}{\sum_{k=1}^{K} \alpha_T(k)} \tag{134}$$

A lot of the literature likes to give intuitive reasoning behind this that goes something like: *$\beta_{t+1}(j)$ is the probability of the future observations given $S_{t+1} = j$, and $\alpha_t(i)$ is the probability of the past observations and $S_t = i$, so what is missing is the probability of transferring from $S_t = i$ to $S_{t+1} = j$ and observing O_{t+1} in the process.* Thankfully it is still possible to get to the same result without needing a deep sense of intuition and instead just trusting the maths: (the θ is dropped below for the sake of simplicity, as it is always assumed a condition given we are always working with the same model)

Proof:

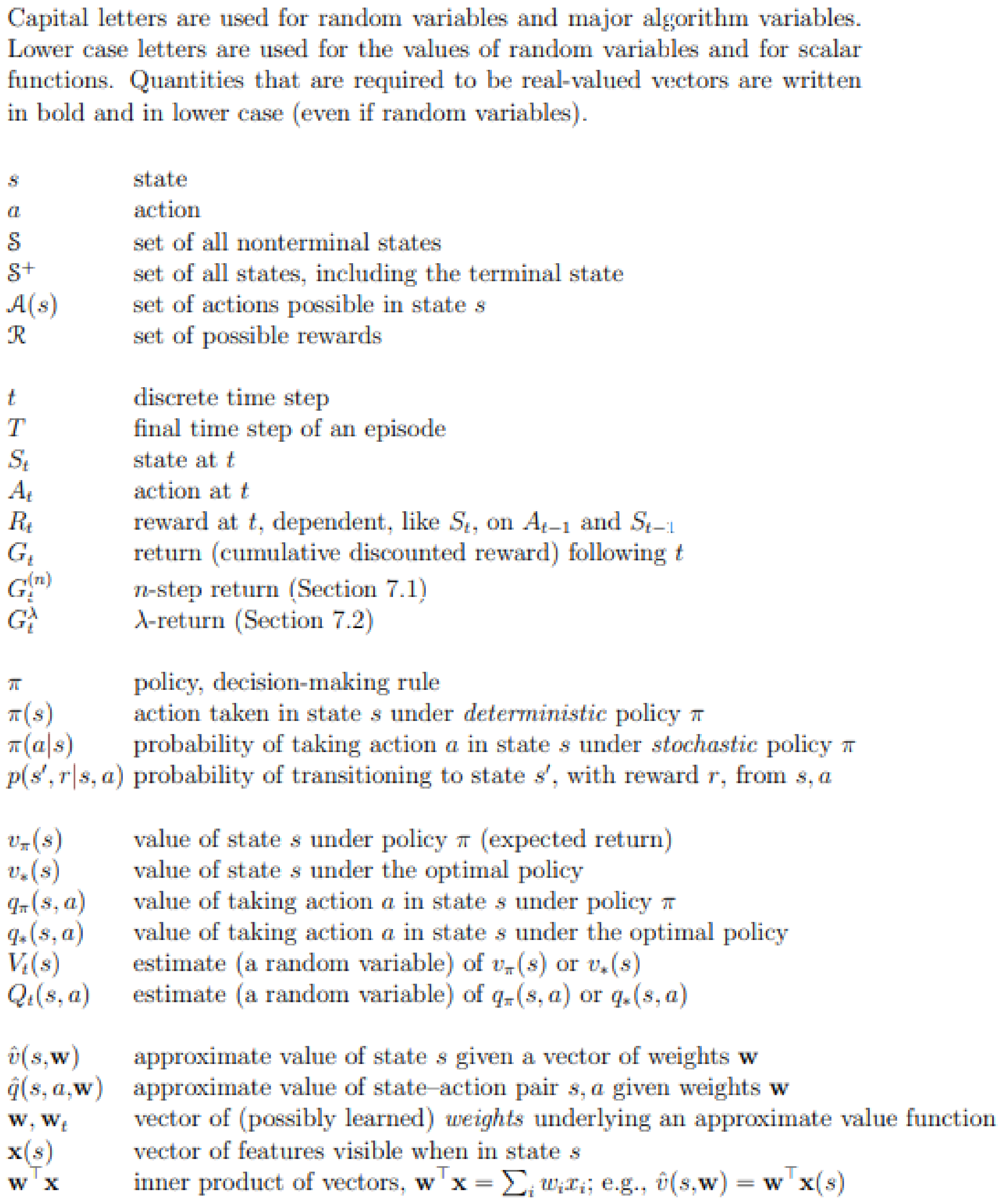
$$\begin{aligned}
P(S_{t+1} = j, S_t = i|\{O_t\}_{t=1}^T) &= \frac{P(S_{t+1} = j, S_t = i, \{O_t\}_{t=1}^T)}{P(\{O_t\}_{t=1}^T)} \\
&= \frac{P(S_{t+1} = j, S_t = i, \{O_{t'}\}_{t'=1}^{t+1})P(\{O_{t'}\}_{t'=t+2}^T|S_{t+1} = j)}{P(\{O_t\}_{t=1}^T)} \\
&= \frac{P(S_{t+1} = j, S_t = i, \{O_{t'}\}_{t'=1}^{t})P(O_{t+1}|S_{t+1} = j)P(\{O_{t'}\}_{t'=t+2}^T|S_{t+1} = j)}{P(\{O_t\}_{t=1}^T)} \\
&= \frac{P(S_t = i, \{O_{t'}\}_{t'=1}^{t})P(S_{t+1} = j|S_t = i)P(O_{t+1}|S_{t+1} = j)P(\{O_{t'}\}_{t'=t+2}^T|S_{t+1} = j)}{P(\{O_t\}_{t=1}^T)} \\
&= \frac{\alpha_t(i)A_{ij}q_j^{O_{t+1}}\beta_{t+1}(j)}{\sum_{k=1}^{K} \alpha_T(k)}, \text{ Q.E.D.}
\end{aligned} \tag{135}$$

Note how, between lines 3 and 4, we have used the Markovian property of the HMM, as $P(S_{t+1} = j, S_t = i, \{O_{t'}\}_{t'=1}^t) = P(S_t = i, \{O_{t'}\}_{t'=1}^t)P(S_{t+1} = j|S_t = i, \{O_{t'}\}_{t'=1}^t)$, but as S_{t+1} is dependent only on the hidden state at the previous time-step and future observations, it simplifies. (S_{t+1} is dependent on past observations, but only through S_t, so if S_t is already explicitly specified as a condition, then the past observations become redundant - the dependency is: $P(S_{t+1}|\{O_{t'}\}_{t'=1}^t) = \sum_{S_t} P(S_{t+1}|S_t)P(S_t|\{O_{t'}\}_{t'=1}^t)$)

Baum-Welch *learning* algorithm: The learning algorithm can be summarised as:

1. Initialise the parameters $\theta = \{\vec{\pi}, A, q\}$ either randomly or through some sensible initialisation algorithm like K-means.
2. Calculate $\{\alpha_i(t), \beta_i(t), \gamma_i(t), \xi_{ij}(t)\}$ for each t for each state i. This is the Expectation (E) step.
3. Use these parameters to re-determine imporved parameter estimates. This is the Maximisation (M) step.
   a. $\hat{\pi}_i = \gamma_i(1) = \frac{\alpha_1(i)\beta_1(i)}{\sum_k \alpha_1(k)\beta_1(k)}$
   b. $\hat{q}_j^k = \frac{\sum_{t=1}^{T} \gamma_j(t)\delta(O_t - k)}{\sum_{t=1}^{T} \gamma_j(t)}$
   c. $\hat{A}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_t(i)}$
4. Terminate when the change in parameters becomes less than some predetermined tolerance.

8 Introduction to Reinforcement Learning



Capital letters are used for random variables and major algorithm variables. Lower case letters are used for the values of random variables and for scalar functions. Quantities that are required to be real-valued vectors are written in bold and in lower case (even if random variables).

s	state
a	action
$\mathcal{S}$	set of all nonterminal states
$\mathcal{S}^+$	set of all states, including the terminal state
$\mathcal{A}(s)$	set of actions possible in state s
$\mathcal{R}$	set of possible rewards
t	discrete time step
T	final time step of an episode
S_t	state at t
A_t	action at t
R_t	reward at t, dependent, like S_t, on A_{t-1} and S_{t-1}
G_t	return (cumulative discounted reward) following t
$G_t^{(n)}$	n-step return (Section 7.1)
G_t^λ	λ-return (Section 7.2)
π	policy, decision-making rule
$\pi(s)$	action taken in state s under *deterministic* policy π
$\pi(a\|s)$	probability of taking action a in state s under *stochastic* policy π
$p(s', r\|s, a)$	probability of transitioning to state s', with reward r, from s, a
$v_\pi(s)$	value of state s under policy π (expected return)
$v_*(s)$	value of state s under the optimal policy
$q_\pi(s, a)$	value of taking action a in state s under policy π
$q_*(s, a)$	value of taking action a in state s under the optimal policy
$V_t(s)$	estimate (a random variable) of $v_\pi(s)$ or $v_*(s)$
$Q_t(s, a)$	estimate (a random variable) of $q_\pi(s, a)$ or $q_*(s, a)$
$\hat{v}(s,\mathbf{w})$	approximate value of state s given a vector of weights $\mathbf{w}$
$\hat{q}(s, a,\mathbf{w})$	approximate value of state–action pair s, a given weights $\mathbf{w}$
$\mathbf{w}, \mathbf{w}_t$	vector of (possibly learned) *weights* underlying an approximate value function
$\mathbf{x}(s)$	vector of features visible when in state s
$\mathbf{w}^\top\mathbf{x}$	inner product of vectors, $\mathbf{w}^\top\mathbf{x} = \sum_i w_i x_i$; e.g., $\hat{v}(s,\mathbf{w}) = \mathbf{w}^\top\mathbf{x}(s)$

Figure 6: The notation taken in this section, as adopted in Sutton and Barto's *Reinforcement Learning: An Introduction.*

8.1 An introduction Using n-armed Bandits

The n-armed bandit problem is the simplest of RL models: there are n possible actions $\{a_1, ..., a_n\}$ with each giving a certain reward if chosen, and that reward being unknown (but often taken to be constant in time). If we wish to maximise the cumulative reward, then we need to be able to predict these rewards by assuming their probability distribution and honing in on the necessary distribution parameters. There are some methods to do this, each with pros and cons that can be helpful for different system conditions. The simplest approaches can be summarised as follows:

8.1.1 The Value Learning Approach

Sample Average Method:

If we know that the rewards are going to be stationary in time, then the most common sense approach to finding their value is to calculate the mean reward for each action over time:

$$Q_t(a) = \frac{1}{N_t(a)} \sum_{i=1}^{N_t(a)} R_i(a) \tag{136}$$

where $N_t(a)$ is the number of times that action a is selected up to and including time t, and $R_i(a)$ is the reward received when a was selected for the i^{th} time. This expression is statistically the *right* way of calculating the reward, as it satisfies the necessary conditions - most importantly that $Q_t(a) \to q(a)$ as $t \to \infty$ by the law of large numbers. However, as discussed soon, it is often not the most *useful* way, with altered heuristic methods giving better results in certain situations. If we are to use this method however, Equation (136) can be calculated in a less computationally expensive way:

$$\begin{aligned} Q_{k+1}(a) &= \frac{1}{k}\sum_{i=1}^{k} R_i(a) = \frac{1}{k}\left[R_k(a) + \sum_{i=1}^{k-1} R_i(a)\right] \\ &= \frac{1}{k}\left[R_k(a) + (k-1)Q_k(a)\right] \\ &= Q_k(a) + \frac{1}{k}\left[R_k(a) - Q_k(a)\right] \end{aligned} \tag{137}$$

where $Q_{k+1}(a)$ is the updated value when a is selected for the $(k+1)^{th}$ time. Notice that as t increases and each action is selected many times (i.e. k is large), the influence of each reward on the average becomes less significant, which is fine for stationary systems, but a problem if the true values change with time, in which case the most recent rewards must be most significant.

With a set of estimate values for each action, the case of choosing the optimal action A_t for each time-step is simple:

$$A_t = \arg\max_a Q_t(a) \tag{138}$$

The above equation is called ***greedy***, because it always chooses what it thinks (based off past evidence) is the best option. This means that the average reward is guaranteed to increase with time (for positive rewards), but there is little to no exploration of other available actions, the same action will continue to be chosen if it has historically been the best expected performer. If there were more exploration then there is quite likely another path through *action-space* which yields a better cumulative reward, even if initially it sacrifices short term yield. However too much exploration can over-sacrifice the safe rewards of greedy options. This is the ***exploration vs exploitation*** trade-off. Often improvements in average received reward can be made by occasionally choosing the non-optimal action, in order to explore different avenues that may have better eventual prospects. The simplest way to do this is to select non-optimal actions randomly at a chosen probability, and is called ϵ-***greedy***:

$$\begin{gathered} r \sim \text{Uniform}([0,1]) \\ A_t = \begin{cases} \arg\max_a Q_t(a) & \text{if } r \geq \epsilon \\ \text{Uniform}(\{a\}) & \text{if } r < \epsilon \end{cases} \end{gathered} \tag{139}$$

Constant Reward Weighting:

The last line of Equation (137) effectively summarises as *New Estimate=Old Estimate+(Step Size)·(Prediction Difference).* Therefore we can deviate from the mathematically correct statement of Equation (137) and make a heuristic difference of making the step size constant:

$$Q_{k+1}(a) = Q_k(a) + \alpha[R_k(a) - Q_k(a)] \tag{140}$$

This now gives constant weighting to all rewards, and does not play down recent evidence in the way that the sample average method did. From this simple form, it is also possible to analytically expand the form for the updates of the value estimates:

$$\begin{aligned} Q_{k+1}(a) &= Q_k(a) + \alpha\left[R_k(a) - Q_k(a)\right] = \alpha R_k(a) + (1-\alpha)Q_k(a) \\ &= \alpha R_k(a) + (1-\alpha)\left[\alpha R_{k-1}(a) + (1-\alpha)Q_{k-1}(a)\right] = ... \\ &= (1-\alpha)^k Q_1(a) + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} R_i(\alpha) \end{aligned} \tag{141}$$

Therefore, once we select α, we need only to store the current reward for each time-step and action. Note that this series no longer converges, and so is mathematically not a correct way of converging in on the true action rewards, but that is often either not a problem for stationary systems, or even desired for non-stationary systems as the true reward for each action changes with time anyway.

Aside: Selecting $Q_1(a)$:

Both this method and the sample average method above have a requirement to select the initial values $Q_1(a)$ at the start of the execution. We could of course go for the simplest option and set it to 0 for all a, but we can take different values to push exploration. If, say, the true rewards follow a dsitribution of $q(a) \sim \mathcal{N}(0,1)$, and we select $Q_1(a) = 5\ \forall\ a$, then the first choice of any action will (almost certainly) yield a lot less of a reward than we have told the algorithm to expect. This means that the algorithm is less likely to choose that action next time and choose a different one instead (where the same outcome likely arises for this next action). Effectively then this is pushing an early exploration approach, where many actions are initially trialled at the expense of early reward, but after enough time-steps the algorithm corrects for the overexaggerated intial values and goes back to a greedier selection. This can produce temporary improvements in the yield of stationary systems.

Upper-confidence-bound action selection:

This in another heuristic approach for selecting actions that encourages exploration and can improve yield for simple and stationary systems. It adds a form of uncertainty to the expected reward of each action for each time-step that is dependent both on the time that the algorithm has been running for, and the number of time that action has been selected. Mathematically it reads:

$$A_t = \arg\max_a \left\{ Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right\} \tag{142}$$

where c is a constant dictating the extent of exploration. In this form, the uncertainty is similar to the square root of a variance, where the heuristic variance increases with time if that action is not selected any more times (i.e. if $N_t(a)$ does not increase). Similarly, if that action is selected often, then the number of times selected increases faster than $\ln t$ due to the logarithmic nature, and thus the uncertainty of that action's value quickly decreases. What is more, exploration is encouraged more for smaller t due to the decreasing gradient of a logarithm. This means that as time increases and the algorithm has more experience from selecting various options there is less emphasis on the need to explore.

An important advantage of this method is that every action is selected an infinite number of times (but with increasing time intervals in-between) as $t \to \infty$ (for $c > 0$), and thus the agent develops much more evidence to give strong accuracy of the estimated rewards to the true rewards $q(a)$.

8.1.2 The Policy Learning Approach

In contrast to the value learning approach above, where an action is chosen with an extent of greediness as the action of best reward, the policy learning approach provides a state to action mapping by constructing a probability distribution of actions based on the current state (for the n-armed bandit, this state is constant with time). From the nature of state distributions, we must ensure that the probability of all states sums to 1, and one common distribution that follows this and maximises the information is the Boltzmann distribution:

$$P(A_t = a) = \frac{\exp H_t(a)}{\sum_{k\in\{a\}} \exp H_t(k)} := \pi_t(a) \\ A_t \sim \pi_t(a) \tag{143}$$

where $H_t(a)$ is the *preference* towards action a (analogous to negative energy in statistical physics). NB: similarly to initial values, the intial preference is set prior to execution, as $H_1(a) = 0 \ \forall \ a$. A feature of this definition is that the probabilities are independent of linear translations in the preferences, i.e.:

$$\begin{aligned} \tilde{H}_t(a) &= H_t(a) + \gamma \ \forall \ a \\ \tilde{\pi}_t(a) &= \frac{\exp\gamma \exp H_t(a)}{\exp\gamma \sum_{k\in\{a\}} \exp H_t(k)} = \pi_t(a) \end{aligned} \tag{144}$$

As before, we now seek to maximise the reward over a series of time-steps by learning the true preferences of the system, and the selecting actions according to Equation (143). With this formulation we can do that quite simply, and update the preferences at each time-step via a gradient ascent method.

$$\begin{aligned} H_{t+1}(a) &= H_t(a) + \alpha \frac{\partial E[R_t]}{\partial H_t(a)} \\ &= H_t(a) + \alpha \frac{\partial}{\partial H_t(a)} \sum_{k\in\{a\}} \pi_t(k) q_t(k) \\ &= H_t(a) + \alpha \sum_{k\in\{a\}} \frac{\partial \pi_t(k)}{\partial H_t(a)} q_t(k) \end{aligned} \tag{145}$$

due to the $q_t(a)$ being dependent on t (if non-stationary) and a only. The next trick relies on the fact that $\pi_t(a)$ is a state distribution, and thus sums to 1 over all states. Therefore all the *changes* to it must eventually cancel to 0 - if it decreases w.r.t. one state, then it must increase w.r.t. another such that the sum is constantly 1 (i.e. the derivative is a Lagrange multiplier). Therefore we have broken no rules to add in the constraint:

$$\begin{aligned} H_{t+1}(a) &= H_t(a) + \alpha \sum_{k\in\{a\}} \frac{\partial \pi_t(k)}{\partial H_t(a)} q_t(k) - \lambda \sum_{k\in\{a\}} \frac{\partial \pi_t(k)}{\partial H_t(a)} \\ &= H_t(a) + \alpha \sum_{k\in\{a\}} \pi_t(k) \frac{\partial \pi_t(k)}{\partial H_t(a)} \frac{[q_t(k) - \lambda']}{\pi_t(k)} \end{aligned} \tag{146}$$

where $\alpha\lambda' = \lambda$. The bottom line of the above equation is now in the form of a statistical expectation w.r.t. the distribution $\pi_t(k)$, and can so be written as:

$$\begin{aligned} H_{t+1}(a) &= H_t(a) + \alpha E_\pi \left[\frac{\partial \pi_t(A_t)}{\partial H_t(a)} \frac{[q_t(A_t) - \lambda']}{\pi_t(A_t)} \right] \\ &= H_t(a) + \alpha E_\pi \left[\frac{\partial \pi_t(A_t)}{\partial H_t(a)} \frac{[R_t - \overline{R}_t]}{\pi_t(A_t)} \right] \end{aligned} \tag{147}$$

We have used here the definition $E[R_t] = q_t(A_t) = E[q_t(A_t)]$, and have redefined $\lambda' = \overline{R}_t$, a parameter representing the average of all rewards up to and including time t that improves learning speed (by using the difference between this and the current reward, rewards that are close together provide little improvements to the preference updates, but rewards that deviate greatly from each other suggest poor precision in the preference values and provide greater magnitude updates, thus speeding learning).

The complicated form of Equation (147) can now be simplified to:

$$\begin{aligned} H_{t+1}(a) &= H_t(a) + \alpha E_\pi \left[(\delta_{aA_t} - \pi_t(a))(R_t - \overline{R}_t) \right] \\ &\approx H_t(a) + \alpha (\delta_{aA_t} - \pi_t(a))(R_t - \overline{R}_t) \end{aligned} \tag{148}$$

The removal of the expectation makes the update more easily computed, and would be exact if we were working with non-stochastic systems. The fact that the systems are stochastic means this is a Monte Carlo like approximation however.

Proof:

First we can use the quotient rule to re-express the derivative:

$$\begin{aligned}
\frac{\partial \pi_t(A_t)}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)}\left[\frac{\exp H_t(A_t)}{\exp \sum_k H_t(k)}\right] \\
&= \left[\frac{1}{\exp \sum_k H_t(k)}\right]^2 \left\{\frac{\partial}{\partial H_t(a)}\left[\exp H_t(A_t)\right]\sum_k \exp H_t(k) - \exp H_t(A_t)\frac{\partial}{\partial H_t(a)}\sum_k \exp H_t(k)\right\} \\
&= \left[\frac{1}{\exp \sum_k H_t(k)}\right]^2 \left\{\delta_{aA_t}\exp H_t(A_t)\sum_k \exp H_t(k) - \exp H_t(A_t)\exp H_t(a)\right\} \\
&= \delta_{aA_t}\pi_t(A_t) - \pi_t(A_t)\pi_t(a) = \pi_t(A_t)\left[\delta_{aA_t} - \pi_t(a)\right]
\end{aligned} \tag{149}$$

Now, plugging that into Equation (147):

$$H_{t+1} = H_t(a) + \alpha E_\pi\left[(R_t - \overline{R}_t)\left(\delta_{aA_t} - \pi_t(a)\right)\right], \text{ Q.E.D.} \tag{150}$$

8.2 Associative Search (*Contextual Bandits*)

With the$n$-armed bandits problem we assume a stationary set-up in which the rewards from the actions do not change with time. In a full RL problem however, the immediate rewards will be constantly changing with time and dependent the current state. As an intermediary between these to problems, we can imagine a set-up in which there are a finite set of states $\{s\}$, and each has a different set of *true* action rewards $\{q_t(a,s)\}$. Now we can recreate the stationary problem as described above once we know which state we are in, or more realistically, we estimate probabilistically what state we are in and choose the optimal actions to take from there. This is the *contextual bandits* problem, and is a good starting point for more complex systems.

9 Computational Appendix

9.1 Hidden Markov Models

Evaluating:

1. Defining $\alpha_t(k)$ as the joint distribution of the hidden state being k and the data up to t. This definition is naive, as the joint distribution suffers numerical underflow as the joint distribution becomes over too many observations (see Section 9.2).

$$\begin{aligned} \vec{\alpha}_t \quad &= (A^T \vec{\alpha}_{t-1}) \circ \vec{B} \text{ for } \vec{B} = P(O_t|\vec{S}_t) \\ P(\{O_t\}_{t=1}^T) &= \sum_{i=1}^{K} \alpha_T(i) \end{aligned} \tag{151}$$

2. Defining $f_t(k)$ as the conditional distribution of the hidden state being k given the data up to t ($P(S_t = k|\{O_{t'}\}_{t'=1}^t)$). This approach requires a normalisation factor Z_t which can then be used to calculate the joint observations probability. Notice also that this method does not suffer the effects of numerical underflow compared to the first method as logarithms can be taken of last line of Equation (154), and the conditional distribution does not approach 0 with time (see the example in Section 9.2).

$$\begin{aligned} \vec{f}_t \quad &= \frac{(A^T \vec{f}_{t-1}) \circ \vec{B}}{(A^T \vec{f}_{t-1}) \cdot \vec{B}} \left(= \frac{P(\vec{S}_t, O_t|\{O_{t'}\}_{t'=1}^{t-1})}{P(O_t|\{O_{t'}\}_{t'=1}^{t-1})} \right) \\ &= \frac{(A^T \vec{f}_{t-1}) \circ \vec{B}}{Z_t} \text{ for } \vec{B} = P(O_t|\vec{S}_t) \\ P(\{O_t\}_{t=1}^T) &= \prod_{t=1}^{T} P(O_t|\{O_{t'}\}_{t'=1}^{t-1}) = \prod_{t=1}^{T} Z_t \end{aligned} \tag{152}$$

Filtering:

1. Using Equation (151), we use Bayes' rule to go from the joint distribution of hidden state with past observations to the conditional distribution:

$$\vec{f}_t = \frac{\vec{\alpha}_t}{\sum_{i=1}^K \alpha_T(i)} \text{ for } \vec{B} = P(O_t|\vec{S}_t) \tag{153}$$

2. In the same way as Equation (154) above, we can define the probability of each hidden state given the observations up to and including that time step using the forward algorithm. This is also a non-naive implementation, and even simpler than the evaluation, as the set $\{f_t\}_{t=1}^T$ is the desired result, no sum needed.

$$\vec{f}_t = \frac{(A^T \vec{f}_{t-1}) \circ \vec{B}}{(A^T \vec{f}_{t-1}) \cdot \vec{B}} \text{ for } \vec{B} = P(O_t|\vec{S}_t) \tag{154}$$

Smoothing:

1. In addition to the forwards algorithm producing the α values, for smoothing we also need the backwards algorithm to produce the β values. For each time-step t, a vectorised method to produce these are as follows (for $\beta_T(k)$ initialised to 1 for each k):

$$\vec{\beta}_t = A(\vec{B} \circ \vec{\beta}_{t+1}) \text{ for } \vec{B} = P(O_{t+1}|\vec{S}_{t+1}) \tag{155}$$

$$P(\vec{S}_t|\{O_t\}_{t=1}^T) = \frac{\vec{\alpha}_t \circ \vec{\beta}_t}{\vec{\alpha}_t \cdot \vec{\beta}_t} \tag{156}$$

2. The above method suffers numerical underflow as both $\alpha_t(k)$ and $\beta_t(k)$ approach 0 as $t \to T$ and $t \to 0$ respectively. Therefore a less naive approach is to use the conditional filter distribution of Equation (154):

$$\begin{aligned} P(S_t = k|\{O_t\}_1^T) &= \frac{P(S_t = k, \{O_{t'}\}_{t+1}^T|\{O_{t'}\}_1^t)}{P(\{O_{t'}\}_{t+1}^T|\{O_{t'}\}_1^t)} \\ &= \frac{P(\{O_{t'}\}_{t+1}^T|S_t = k, \cancel{\{O_{t'}\}_1^t})P(S_t = k|\{O_{t'}\}_1^t)}{P(\{O_{t'}\}_{t+1}^T|\{O_{t'}\}_1^t)} \end{aligned} \tag{157}$$

or in terms of the vector parameters (where $\vec{\beta}_t$ is defined as before):

$$P(\vec{S}_t|\{O_t\}_1^T) = \frac{\vec{\beta}_t \circ \vec{f}_t}{P(\{O_{t'}\}_{t+1}^T|\{O_{t'}\}_1^t)} \tag{158}$$

Note how the distribution above is conditional, and hence sums over all states to 1. This therefore requires:

$$\begin{aligned}\sum_{k=1}^{K} \vec{\beta}_t \circ \vec{f}_t = \vec{\beta}_t \cdot \vec{f}_t = P(\{O_{t'}\}_{t+1}^T|\{O_{t'}\}_1^t) \\ P(\vec{S}_t|\{O_t\}_1^T) = \frac{\vec{\beta}_t \circ \vec{f}_t}{\vec{\beta}_t \cdot \vec{f}_t}\end{aligned} \tag{159}$$

Notice how we still have the issue that $\vec{\beta}_t$ still underflows for $t \to 0$, but this is an improvement. For a full fix, see Example 2 in 9.2.

9.2 Numerical Underflow and the Log Domain

All of the algorithms we have seen so far in this document are *naive*, which is to say that they are direct mathematical solutions that give correct answers. However, they do not account for the actual computational implementation of the algorithms, and the errors that arise due to time complexity or memory storage, for example.

One such problem is *numerical underflow*, where the long sequential multiplication of probabilities produces values that are so small that their precision cannot be stored on a computer, and so they are rounded down to 0. This produces many problems when it comes to normalisation or choosing argmax for example, and so must be avoided. One way to avoid this issue is through working in the log domain, where a very small decimal less than one can be stored safely as a very negative number, and rewritten in absolute space as an exponential if necessary.

The way to prevent numerical underflow depends on the situation, but a few ideas are to work with conditional distributions rather than joint distributions (so that the sum of the probabilities for each time-step is 1, meaning they don't decrease with time to 0); to shift parameters into the log domain as described above; or an extension of this, to use the *log-sum-exp* rule if needed.

The log-sum-exp method:

Imagine a probability distribution $P(Y) = \sum_i P(Y, X = i)$ s.t. $P(Y)$ is so small it cannot be stored in computer memory. We can take logarithms of both sides to give:

$$\ln[P(Y)] = \ln\left[\sum_i P(Y, X = i)\right] = \ln\left[\sum_i P(Y|X = i)P(X = i)\right] \tag{160}$$

where the second equality is using Bayes' rule. Equation (160) can be expanded even further as:

$$\begin{aligned}\ln[P(Y)] = \ln\left[\sum_i \exp\left[\ln\left(P(Y|X = i)\right) + \ln\left(P(X = i)\right)\right]\right] := \ln\left[\sum_i \exp(\chi_i)\right] \\ \chi_i := \ln\left(P(Y|X = i)\right) + \ln\left(P(X = i)\right)\end{aligned} \tag{161}$$

Equation (161) is why this method is given the name log-sum-exp. Notice how we are now adding logarithms of probabilities rather than multiplying, so we do not see the dramatic decreases in value that we did before. However, if the probabilities are very small, then the χ_is will be very negative, and so the exponential again very small - so far we haven't actually *done* anything about the size issue. We can resolve this however by noticing an interesting relation and factoring out a constant from the exponentials. The relation is to do with maximums, and is:

$$\ln\left[\sum_i \exp(\chi_i)\right] \approx \max_i \chi_i \tag{162}$$

where the approximation is better the bigger the difference between the maximum χ_i and the rest. This is because exponentials increase very fast, so $\exp(\chi_{\max}) >> \exp(\chi_{\text{others}})$, meaning:

$$\ln\left[\sum_i \exp(\chi_i)\right] \approx \ln\left[\exp(\chi_{\max})\right] = \max_i \chi_i, \text{ Q.E.D.} \tag{163}$$

With this in mind, we can factor out the maximum χ_i from all of them, and rewrite Equation (161) as:

$$\ln[P(Y)] = \ln\left[\sum_i \exp\left(\chi_i - \max_i \chi_i\right)\right] + \max_i \chi_i \tag{164}$$

where every term both in the logarithms and addition is now of an order storable in computer memory. We can consequently store and carry out computations with $\ln[P(Y)]$ instead of $P(Y)$ regardless of how small the probabilities are, and return the absolute probabilities through:

$$P(Y) = \exp\left(\max_i \chi_i(Y)\right)\left[\sum_i \exp\left(\chi_i - \max_i \chi_i\right)\right] \tag{165}$$

Example 1: Logarithmic HMM evaluation and filtering

Approach 1 - joint distribution:

One approach is to use the sequentially decreasing joint distribution definition of $\alpha_t(k)$ and the log-sum-exp trick discussed above.

$$\begin{aligned}
\alpha_t(k) &= \sum_i P(O_t|S_t = k)A_{ik}\alpha_{t-1}(i) \\
\ln\left[\alpha_t(k)\right] &= \ln\left[\sum_i \exp\left(\chi_i(k)\right)\right] \\
\chi_i(k) &:= \ln\left[P(O_t|S_t = k)\right] + \ln\left[A_{ik}\right] + \ln\left[\alpha_{t-1}(i)\right] \\
\therefore\ \ln\left[\alpha_t(k)\right] &= \ln\left[\sum_i \exp\left(\chi_i(k) - \max_i \chi_i(k)\right)\right] + \max_i \chi_i(k)
\end{aligned} \tag{166}$$

Although this gives us a means of storing $\alpha_t(k)$ as its logarithm for very small probabilities, this is not very useful for evaluation as the sum over states within α_t is needed, and there is no method of expressing a sum over states using Equation (166) without leaving logarithm domain and getting numerical errors.

Approach 2 - conditional distribution:

Therefore, we can try a more clever approach, using a newly defined parameter that can be expressed as a product (and hence is easy to use in the log domain as a sum). With this in mind, and from using the probability chain rule, we need to find a parameter Z_t such that:

$$P(\{O_t\}_{t=1}^{T}) = \prod_{t=1}^{T} P(O_t|\{O_{t'}\}_{t'=1}^{t-1}) = \prod_{t=1}^{T} Z_t \tag{167}$$

It happens to be the case that $P(O_t|\{O_{t'}\}_{t'=1}^{t-1})$ is the denominator in Bayes' rule of the conditional distribution:

$$\begin{aligned}
P(S_t = k|\{O_{t'}\}_{t'=1}^{t}) = P(S_t = k|O_t, \{O_{t'}\}_{t'=1}^{t-1}) \\
&= \frac{P(S_t = k, O_t|\{O_{t'}\}_{t'=1}^{t-1})}{P(O_t|\{O_{t'}\}_{t'=1}^{t-1})} \\
&= \frac{P(O_t|S_t = k, \{O_{t'}\}_{t'=1}^{t-1})P(S_t = k|\{O_{t'}\}_{t'=1}^{t-1})}{P(O_t|\{O_{t'}\}_{t'=1}^{t-1})} \\
&= \frac{q_k^{O_t}\sum_i A_{ik}f_{t-1}(i)}{P(O_t|\{O_{t'}\}_{t'=1}^{t-1})}
\end{aligned} \tag{168}$$

Therefore, if we can work out that denominator, we can workout Z_t and hence express the product form of Equation (167). Well we can see that as the equation above is a *conditional* distribution, so if we sum over the hidden states k we better get 1 - i.e.:

$$\begin{aligned}\sum_k \frac{q_k^{O_t}\sum_i A_{ik}f_{t-1}(i)}{P(O_t|\{O_{t'}\}_{t'=1}^{t-1})} &= 1\\ \sum_k q_k^{O_t}\sum_i A_{ik}f_{t-1}(i) = P(O_t|\{O_{t'}\}_{t'=1}^{t-1})\end{aligned} \tag{169}$$

or in vector notation:

$$Z_t = \sum_{k=1}^{K}\left[(A^T\vec{f}_{t-1})\circ\vec{B}\right]_k \tag{170}$$

We still have a sequential product of probabilities, but now with the difference that no probability in Equation (170) decreases dramatically with time, as they are all either static, or conditional distributions. Therefore there is no numerical underflow issue in calculating Z_t and we do not need to use the log-sum-exp rule.

Finally we can resort to Equation (167). This time there *is* numerical underflow, as each Z_t is less than one, so the product strictly approaches 0 with time. But as we have used a product, the complex log-sum-exp rule isn't necessary, we can take logarithms directly:

$$\ln\left[P(\{O_t\}_{t=1}^{T})\right] = \ln\left[\prod_{t=1}^{T} Z_t\right] = \sum_{t=1}^{T}\ln\left[Z_t\right] \tag{171}$$

Example 2: HMM Logarithmic smoothing

Recall how we defined a new method for smoothing in Section 9.1, using the conditional filter distribution and the $\beta_t(k)$ parameter. This new method doesn't suffer underflow from $f_t(k)$ any more, but still does from $\beta_t(k)$, and therefore we need some more involved methods to fix this. The sole issue is with the sequentially smaller values of $\beta_t(k)$, and so in order to work with its logarithms rather than absolute values, we can take the natural log of the smoothed conditional distribution:

$$\begin{aligned}\ln P(S_t = k|\{O_t\}_1^T) &= \ln\beta_t(k) + \ln f_t(k) - \ln\left[\sum_{k=1}^{K}\beta_t(k)f_t(k)\right]\\ &= \ln\beta_t(k) + \ln f_t(k) - \ln\left[\sum_{k=1}^{K}\exp\left(\ln\beta_t(k) + \ln f_t(k)\right)\right]\\ &= \chi_t(k) - \ln\left[\sum_{k=1}^{K}\exp\left(\chi_t(k)\right)\right]\\ \chi_t(k) :&= \ln\beta_t(k) + \ln f_t(k)\end{aligned} \tag{172}$$

In the expression above, $\chi_t(k)$ is very negative for very small $\beta_t(k)$, and so taking the exponential will just reproduce the errors. Therefore we now employ the log-sum-exp rule as a fix, to make every part of the expression well defined for all t:

$$\ln P(S_t = k|\{O_t\}_1^T) = \chi_t(k) - \ln\left[\sum_{k=1}^{K}\exp\left(\chi_t(k) - \max_k\chi_t(k)\right)\right] - \max_k\chi_t(k) \tag{173}$$

Therefore we can see how to reduce the problem to simply finding the logarithms of $f_t(k)$ and $\beta_t(k)$. It is trivial to find $\ln f_t(k)$, as $f_t(k)$ is itself well defined for all t, and so we only need take the logarithm of the already found values. However, our original issue is that $\beta_t(k)$ *isn't* well defined for all t due to underflow, and so we must search for a relation between $\ln\beta_t(k)$ and $\ln\beta_{t+1}(k)$ instead. Going back to the original parameter definition given in Section 7.2.2, we can shift to the log domain:

$$\begin{aligned}
\ln \beta_t(k) &= \ln \left[\sum_i A_{ik} q_i^{O_t} \beta_{t+1}(k) \right] \\
&= \ln \left[\sum_i \exp\left(\gamma_{ki}(t)\right) \right] \\
&= \ln \left[\sum_i \exp\left(\gamma_{ki}(t) - \max_i \gamma_{ki}(t) \right) \right] + \max_i \gamma_{ki}(t) \\
\gamma_{ki}(t) :&= \ln A_{ik} + \ln q_i^{O_t} + \ln \beta_{t+1}(k)
\end{aligned} \tag{174}$$

where we have used the log-sum-exp *again* to account for $\gamma_{ki}(t)$ being very negative due to $\ln \beta_{t+1}(k)$. And so there is our solution, we have found a means of needing to use only the well-defined $\ln \beta_t(k)$ without needing to know $\beta_t(k)$ first, *and* we have found a method to compute the smoothed probabilities directly from these logarithmic values. In vector form, the algorithm can be summarised as below:

1. Initialise, for $t = T$:

$$\begin{aligned}
&\ln \vec{\beta}_T = \vec{0} \\
&\vec{\chi}_T = \ln \vec{f}_T + \ln \vec{\beta}_T \\
&\ln P(\vec{S}_T | \{O_t\}_1^T) = \vec{\chi}_T - \max_k \chi_T(k) - \ln \left[\sum_k \exp\left(\chi_T(k) - \max_k \chi_T(k) \right) \right]
\end{aligned} \tag{175}$$

2. Compute for $t = T - 1...1$ and $\vec{B} = P(O_t | \vec{S}_t)$:

$$\begin{aligned}
&\gamma_{ik} = \ln A_{ki} + \ln \vec{B}_i + \ln \beta_{t+1}(i) \text{ for each } i, k \\
&\ln \beta_t(k) = \ln \left[\sum_i \exp\left(\gamma_{ik} - \max_i \gamma_{ik} \right) \right] + \max_i \gamma_{ik} \\
&\vec{\chi}_t = \ln \vec{f}_t + \ln \vec{\beta}_t \\
&\ln P(\vec{S}_t | \{O_t\}_1^T) = \vec{\chi}_t - \max_k \chi_t(k) - \ln \left[\sum_k \exp\left(\chi_t(k) - \max_k \chi_t(k) \right) \right]
\end{aligned} \tag{176}$$

3. Terminate after $t = T$, and reform the smoothing distribution (note that we can simply exponentiate as this is a conditional matrix and thus not at risk of underflow):

$$\{P(\vec{S}_t | \{O_t\}_1^T)\}_1^T = \exp\left(\left\{ \ln P(\vec{S}_t | \{O_t\}_1^T) \right\}_1^T \right) \tag{177}$$

10 Areas of Current Research

10.1 A Biologically Plausible ANN

Unsupervised learning by competing hidden units. Dmitry Krotov, John J. Hopfield. Proceedings of the National Academy of Sciences Apr 2019, 116 (16) 7723-7731; DOI: 10.1073/pnas.1820458116

Classical artificial networks which update through the backpropagation algorithm are not inline with how biological networks update synapse strengths due to locality - in the brain, the synapses in the lower layers of the network cannot have information about the states of the higher layers or the correct labels due to no physical connection to provide the information and no sufficient central memory. Therefore, the biological analogue to artificial supervised learning is a lot more unsupervised by necessity, the lower layers of the network have to just learn general features of the inputs without directly associating those features to the labels, as would be the case for the fully supervised case.

It is believed that biological synapses update their weights based upon the potentials that pass through them. In this light, this paper builds a strategy for updating the weights in a competitive sense, where the hidden units compete to increase relevance for the task in hand relative to each other. It combats the issue of locality by ensuring that all weights update are defined as a function of the form:

$$\Delta W_{\mu i} = f(I_\mu, I_i) \tag{178}$$

where $\Delta W_{\mu i}$ is the weights update between neurons μ and i in two connected layers ν and h, I_μ and I_i are the output currents of those neurons, and f is some function. The structure of a network with one hidden layer is shown below:

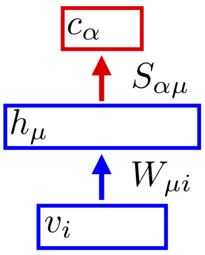


Figure 7: Diagrammatic structure of a shallow network with the weights between the lower layers being dependent only on the potentials of the connecting neurons, and the highest layer having a weight that learns in a traditional supervised manner due to its direct connection to the label layer. Taken from the paper.

Synaptic Rules:

The activations are a non-linear function of a linear sum of inputs, as in traditional ANNs, with the non-linearities given by:

$$\begin{aligned} h_\mu &= f(W_{\mu i}\nu_i) \text{ for } f(x) = \begin{cases} x^n & x \geq 0 \\ 0 & x \leq 0 \end{cases} \\ c_\alpha &= \tanh(\beta S_{\alpha\mu} h_\mu) \end{aligned} \tag{179}$$

where n and β are hyperparameters for tuning, and $n = 1$ is equivalent to a ReLu. The weights update is then importantly dependent on only local neurons, and is derived based from *Oja's rule* ($\Delta W_i = \eta(x_i y - w_i y^2)$, $y = \sum_i W_i x_i$) of neuroscience, which is a wrights-regularisation adaption of *Hebbian rule* ($\Delta W_i = \eta x_i y$) for synaptic weights update. Therefore, for 1 hidden unit ($W \rightarrow \vec{W}$), it is given by:

$$\begin{aligned} \tau_L \frac{\partial W_i}{\partial t} &= g(Q)\left[R^p \nu_i - \langle W, \nu \rangle W_i\right] \\ \langle X, Y \rangle &:= \sum_{ij} \delta_{ij} X_i Y_j |W_i|^{p-2} \\ Q &:= \frac{\langle W, \nu \rangle}{\langle W, W \rangle^{\frac{p-1}{p}}} \end{aligned} \tag{180}$$

where they use two new parameters, τ_L (which controls how fast the weights are learnt), and R, a constant. $g(Q)$ is a non-linear function, and p the Lebesgue norm. $g(Q)$ is selected such that it takes positive or negative values dependent on the post-synaptic potential characteristics Q, and therefore can cause both *Hebbian* and *anti-Hebbian* learning. The one-hidden-unit rule was shown above to demonstrate how it is an adaptation of the Oja rule (recovered if $g(Q) = Q$ and $p = 2$), but for the case of a neural network, it is more helpful to extend the number of hidden units and return to rank 2 tensors for the weight:

$$\begin{aligned} \tau_L \frac{\partial W_{\mu i}}{\partial t} &= g(Q) \left[R^p \nu_i - \langle \vec{W}_\mu, \nu \rangle_\mu W_{\mu i} \right] \\ \langle X, Y \rangle_\mu &:= \sum_{ij} \delta_{ij} X_i Y_j |W_{\mu i}|^{p-2} \end{aligned} \tag{181}$$

The inclusion of R ensures that each of the hidden units' weights $\vec{W}_\mu$ will eventually converge to a vector of length R, and therefore that they all lie on a hypersphere of radius R is weight-space.

Proof: that R is the converged radius of the weights vectors

First we need to know what to look for, and so we can see that from the definition of the inner product under the p norm, the inner product of W_μ with itself is:

$$\langle W_\mu, W_\mu \rangle_\mu = \sum_{ij} W_{\mu i} W_{\mu j} \delta_{ij} |W_{\mu i}|^{p-2} = \sum_i (W_{\mu i})^2 |W_{ij}|^{p-2} = \sum_i |W_{\mu i}|^p = |W_\mu|^p \tag{182}$$

And so, by making use of Equation (181) (but in vector form rather than index form - i.e. dropping i), we get:

$$\begin{aligned} \tau_L \frac{\partial}{\partial t} \langle W_\mu, W_\mu \rangle = \tau_L \frac{\partial}{\partial t} |W_\mu|^p &= p \tau_L \langle W_\mu, \frac{\partial}{\partial t} W_\mu \rangle \\ &= p g(Q) \langle W_\mu, \nu \rangle \left[R^p - \langle W_\mu, W_\mu \rangle \right] \end{aligned} \tag{183}$$

This equation says that, *for as long as $g(Q)$ and $\langle W_\mu, \nu \rangle$ are greater than 0*, the length of the W_μ vector will increase with time if and only if it is less than R^p (where the derivative is positive), and will decrease with time if and only if it is greater than R^p. This trade-off will continue as the lenght approaches R^p from both sides, until it reaches equilibrium when $\langle W_\mu, W_\mu \rangle = R^p$, and the derivative becomes 0 (and the length stops changing with time). This condition breaks down if $g(Q)\langle W_\mu, \nu \rangle \leq 0$, but fortunately it turns out that for a small violation ($-\Delta$ for a small Δ), the hidden units' weights still converge to R^p.

Learning Algorithm:

To reflect the biological nature of the method, firing rates are used over activation magnitudes, where the weights dictate whether a certain firing rate produces an excitatory or inhibitory potential. Higher weights reflect a more excitatory neuron, and the firing rate increases for that neuron. The threshold for whether a firing rate is treated as inhibitory or excitatory is h_*. Specifically, the (unsupervised) algorithm is as follows:

1. A set of inputs $\{\nu_i\}$ are converted into input currents $\{I_\mu\}$ via:

$$I_\mu = \langle W_\mu, \nu \rangle_\mu \tag{184}$$

2. The dynamics of the individual hidden neuron firing rates are then governed by the differential equation:

$$\tau \frac{\partial h_\mu}{\partial t} = I_\mu - w_{inh} \sum_{\nu \neq \mu} r(h_\nu) - h_\mu \tag{185}$$

   where $\tau << \tau_L$ is a constant defining the neuron-specific dynamical timescale, w_{inh} describes the strength of global inhibition that gives the competition between neurons (the higher it is the more they compete to be excitatory as the harder it is for a neurons firing rate to increase with time relative to other neurons), and $r(h) = \max(h, 0)$ is a ReLU function ensuring comparative firing rates are non-negative.

3. The firing rates are then fed into the non-linear activation function, with $g(Q) \to q(h_\mu)$ in Equation (181):

$$g(h) = \begin{cases} 0 & h < 0 \\ -\Delta & 0 \leq h < h_* \\ 1 & h \geq h_* \end{cases} \tag{186}$$

   which ensures that negative firing rates do not contribute to learning, while positive firing rates under the threshold become inhibitory neurons, and select neurons' firing rates above the threshold represent excitatory neurons.

The inhibition behind this is that if a hidden unit is very driven by a certain input feature, the learning algorithm pushes it towards recognising this feature in the future, whilst less activated units are pushed away form the feature in the future. This creates the specificity between input features and individual unit activations that is required of a neural network (particularly for classification), and is inspired by BCM theory in neuroscience and *winner-takes-all* learning.

The real-world drawback of this method is its slow time complexity due to the requirement to to solve Equation (185) numerically to find the firing rates, and the lack of mini-batches. Therefore a couple of computational tweaks were made to improve efficiency whilst retaining adequate performance from the original algorithm.

Computational Alterations:

1. Instead of solving numerically the differential Equation (185), the currents are used as a proxy for the *ranking* of the hidden units, with higher current giving higher ranking (i.e. the absolute firing rates are approximated as unnecessary and only their relative ranking, approximated through their relative input currents, is deemed important).

2. This requires a new $g(h)$ that cares not for absolute firing rates, just for relative size. This is accomplished through:
$$g(i) = \begin{cases} 1 & i = K \\ -\Delta & i = K - k \\ 0 & \text{otherwise} \end{cases} \tag{187}$$
where $i = K$ is the index for the highest ranked firing rate, and k is an index representing the k^{th} highest ranked index (set by the user, i.e. $k = 2$ makes the second highest ranked firing rate inhibitory).

3. Mini-batches can subsequently be introduced, if the ranking of input currents is performed for each input and propagated through Equation (181), but the weights update then averaged over the entire mini-batch. This reduces over-fitting of specific input features to hidden neurons.

With these adaptations, the computational complexity is reduced to $O(Kk)$x(mini-batch size).

11 (Cambridge) Inference Models

11.1 Bayesian Regression

11.1.1 Offline Learning

The usual method for linear regression uses least squares fitting to minimise the absolute distance (squared) between the observed data and a linear relationship. This seems intuitive, but also arbitrary - why use the modulus squared and not the modulus, and how can uncertainties be inferred in addition to mean fit? The fix for this is to treat regression in a probabilistic sense using Bayesian inference (and it turns out that the least squared minimisation is equivalent to maximum likelihood estimation with Gaussian errors).

Linear fitting:

Let a one-dimensional linear fit be given by:

$$y_n = w_0 + w_1 x_n + \epsilon_n,\ \epsilon_n \sim \mathcal{N}(0, \sigma_y^2) \tag{188}$$

We can directly infer from the model that:

$$p(y_n | x_n, \vec{w}) = \mathcal{N}(w_0 + w_1 x_n, \sigma_y^2) \tag{189}$$

and thus we use Maximum Likelihood Estimation in the usual way (accounting for all observations, hence the joint distribution over observations) to find that:

$$\begin{aligned}\mathcal{L}(\vec{w}) &= \ln p(\vec{y}|\vec{x}, \vec{w}) \\ &= -\frac{1}{2\sigma_y^2}(\vec{y} - X\vec{w})^T(\vec{y} - X\vec{w}) - \frac{N}{2}\ln 2\pi\sigma_y^2\end{aligned} \tag{190}$$

where for convenience we have adapted notation such that $\vec{w}^T = (w_0, w_1)$ and $X = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \cdots & \cdots \\ 1 & x_N \end{bmatrix}$.

Minimising (via differentiation) the expression in Equation (190) yields the familiar result:

$$\vec{w}_{ML} = (X^T X)^{-1} X^T \vec{y} \tag{191}$$

Non-linear fitting:

A simple way to extend this formulation to non-linear fits of a one dimensional input is to replace the linear x_n with some non-linear set of basis functions $\{\phi_d(x)\}_{d=1}^D$ such that:

$$\begin{aligned}\vec{y} &= w_0 + w_1\phi_1(\vec{x}) + ... + w_D\phi_D(\vec{x}) \\ &= \Phi_D \vec{w}\end{aligned} \tag{192}$$

where $\vec{x}$ and $\vec{y}$ are vector representations of the N input and output pairs, and $\Phi_D = \begin{bmatrix} 1 & \phi_1(x_1) & ... & \phi_D(x_1) \\ 1 & \phi_1(x_2) & ... & \phi_D(x_2) \\ \cdots & \cdots & \cdots & \cdots \\ 1 & \phi_1(x_N) & ... & \phi_D(x_N) \end{bmatrix}$.

From here we can follow the same procedure as above for the linear case (simply noticing that this form is equivalent to before with now a non-linear basis function set), and write:

$$\vec{w}_{ML} = (\Phi_D^T \Phi_D)^{-1} \Phi_D^T \vec{y} \tag{193}$$

11.1.2 Weight Regularisation

The above formulation for non-linear regression suffers from weights blow-up and over-fitting for high D. This is because when there are a large number of basis functions there is enough flexibility for the weights to be adjusted in such a way that the estimating function can go through every data point in the training dataset, often by setting the weights to very large values to cause near vertical rises and drops between points. Therefore we can regularise the weights to penalise high weights and increasing dimensions so that when we add basis functions we do not over-fit in the same way.

Intuitively, if we want to penalise the increase of weights then it makes sense to increase the cost by a term proportional to $|\vec{w}|^2$ in the cost function for linear regression. It turns out that this is also derivable through a more thorough derivation in Bayesian inference, and generalised into non-linear regression also, by introducing a weights distribution. If we desire the weights to be small (before knowing anything about the training data) and able to take any real value, then it makes sense to make the prior a uniform distribution $\vec{w} \sim \mathcal{N}(0, \sigma_w^2 \hat{I})$. From here we find the posterior over weights as:

$$\begin{aligned} p(\vec{w}|\vec{y}, \Phi_D) &\propto p(\vec{y}|\vec{w}, \Phi_D)p(\vec{w}) \\ &= \mathcal{N}(\Phi_D \vec{w}, \sigma_y^2 \hat{I})\mathcal{N}(0, \sigma_w^2 \hat{I}) \\ &\propto \exp\left[-\frac{\vec{w}^T}{2}\left(\frac{\Phi_D^T\Phi_D}{\sigma_y^2} + \frac{\hat{I}}{\sigma_w^2}\right)\vec{w} + \frac{\vec{w}^T\Phi_D^T\vec{y}}{\sigma_y^2}\right] \end{aligned} \tag{194}$$

where we notice that the product of two Gaussians is also Gaussian, and therefore the above expression can be expressed as $p(\vec{w}|\vec{y}, \Phi_D) = \mathcal{N}(\vec{\mu}_{w|x,y}, \Sigma_{w|x,y})$, with:

$$\begin{aligned} \Sigma_{w|x,y} &= \left(\frac{\Phi_D^T\Phi_D}{\sigma_y^2} + \frac{\hat{I}}{\sigma_w^2}\right)^{-1} \\ \vec{\mu}_{w|x,y} &= \frac{\Sigma_{w|x,y}\Phi_D^T\vec{y}}{\sigma_y^2} = \left(\Phi_D^T\Phi_D + \frac{\sigma_y^2}{\sigma_w^2}\hat{I}\right)^{-1}\Phi_D^T\vec{y} \end{aligned} \tag{195}$$

As the a posteriori distribution is Gaussian, then the *maximum a posteriori* weight is also the mean of the distribution, and thus we conclude that for Gaussian weight priors we see that the result is equivalent to least squares minimisation (see Equation (191)) with a regularisation term of the ratio of variances. If $\sigma_w^2 \to \infty$ then the weights prior is a uniform distribution, and the regularisation term vanishes (as it would be if there was no regularisation).

11.1.3 Uncertainty Extrapolation and Online Learning

In order to get a line of best fit. we can create a uniformly spaced set of inputs, $\{x^*\}$, and predict using the learned weight distribution (i.e. after training) what the outputs will be. As we are now using a weights *distribution* now rather than point-estimates (as was the case for maximum a posteriori and maximum likelihood models), we can give improved estimates as the expected value of $\{y^*\}$ when integrating over the learned weights distribution, and we can take it one step further and find the variance of $\{y^*\}$ also, which gives us a measure of the uncertainty. Mathematically this materialises as:

$$\begin{aligned} \mu_{y^*} = \mu_{y^*|x^*,y,x} = E[y^*|x^*, \vec{y}, \Phi_D] &= \int y^* p(y^*|x^*, \vec{y}, \Phi_D) dy^* \\ &= \int y^* \left\{\int p(y^*|\vec{w}, x^*)p(\vec{w}|\vec{y}, \Phi_D)d\vec{w}\right\} dy^* \\ &= \vec{\phi}^{*T}\vec{w}_{MAP} = \vec{\phi}^{*T}\vec{\mu}_{w|x,y} \end{aligned} \tag{196}$$

This is because the model for new data $\{y^*\}$ given $\{x^*\}$ is $y^* = \vec{\phi}(x^*)^T\vec{w}_{MAP} + \epsilon = \vec{\phi}^{*T}\vec{w}_{MAP} + \epsilon$ (i.e. non-linear regression using learned weights post-training), and the maximum a posteriori weights vector is equivalent to the mean of the a posteriori Gaussian weights distribution. Whilst it is certainly possible to use the second line's integral to derive the final expression long-hand, going from the second line to the last line of Equation (197) used a simplification inherent to Gaussian distributions that is explained in the **Aside** below.

In a similar way the variance of the predictive distribution can be calculated to give a measure of uncertainty in our Bayesian regression model:

$$\begin{aligned}\sigma^2_{y^*} = \sigma^2_{y^*|x^*,y,x} &= E[(y^* - \mu_{y^*})^2] \\ &= \vec{\phi}^{*T}\Sigma_{w|x,y}\vec{\phi}^* + \sigma^2_y\end{aligned} \tag{197}$$

Therefore, to summarise, the posterior predictive distribution, given Gaussian a priori weights have that been trained using training data $\vec{y}, \vec{x} \to \Phi_D$, is:

$$p(y^*|x^*, \vec{y}, \Phi_D) = \mathcal{N}(\vec{\phi}^{*T}\vec{\mu}_{w|x,y},\ \vec{\phi}^{*T}\Sigma_{w|x,y}\vec{\phi}^* + \sigma^2_y) \tag{198}$$

Aside: Obtaining the mean and variance of a linear Gaussian system

It can be shown that for a model $\vec{z} = A\vec{x} + B\vec{y}$ with A, B fixed matrices and $\vec{x}, \vec{y}$ Gaussianly distributed random variables, the distribution of $\vec{z}$ is given by:

$$\vec{z} \sim \mathcal{N}(A\vec{\mu}_x + B\vec{\mu}_y,\ A^T\Sigma_x A + B^T\Sigma_y B) \tag{199}$$

as:

$$\begin{aligned}\vec{\mu}_z &= E[A\vec{x} + B\vec{y}] = AE[\vec{x}] + BE[\vec{y}] \\ &= A\vec{\mu}_x + B\vec{\mu}_y \\ \Sigma_z &= \text{Cov}[A\vec{x} + B\vec{y}] = \text{Cov}[A\vec{x}] + \text{Cov}[B\vec{y}] \\ &= E[(A\vec{x} - A\vec{\mu}_x)(A\vec{x} - A\vec{\mu}_x)^T] + E[(B\vec{x} - B\vec{\mu}_y)(B\vec{y} - B\vec{\mu}_y)^T] \\ &= AE[(\vec{x} - \vec{\mu}_x)(\vec{x} - \vec{\mu}_x)^T]A^T + BE[(\vec{x} - \vec{\mu}_y)(\vec{y} - \vec{\mu}_y)^T]B^T \\ &= A\Sigma_x A^T + B\Sigma_y B^T\end{aligned} \tag{200}$$

This leads to the principle of *online* Bayesian learning, as the training data set is flexible in length and can thus be updated in real time. There are two approaches to do this, though in actuality they are equivalent ways to reach the same result.

The two sufficient statistics required to fully define the posterior weights distributions and thus complete the Bayesian predictive problem are $\Phi_D^T\Phi_D$ and $\Phi_D^T\vec{y}$. We can re-calculate these statistics for any length of training set, so if we ae in a scenario where data is coming in dynamically, we start with no 1 training point and then on arrival of the second, update the weights distribution by re-calculating these statistics. With these statistics it is possible to calculate the posterior predictive distribution in order to find the regression mean and uncertainty at any point, and thus the regression model fits to the data with time as more data arrives. The problem thus reduces to calculating the statistics in a dynamic programming kind of way.

By writing the matrix calculations in terms of index notation we see that:

$$\begin{aligned}(\Phi_D^T\Phi_D)^{(N)}_{ij} &= \sum_{n=1}^{N}\phi_i(x_n)\phi_j(x_n) = \sum_{n=1}^{N-1}\phi_i(x_n)\phi_j(x_n) + \phi_i(x_N)\phi_j(x_N) \\ &= (\Phi_D^T\Phi_D)^{(N-1)}_{ij} + \phi_i(x_N)\phi_j(x_N) \\ (\Phi_D^T\vec{y})^{(N)}_d &= \sum_{n=1}^{N}\phi_d(x_n)y_n = \sum_{n=1}^{N-1}\phi_d(x_n)y_n + \phi_d(x_N)y_N \\ &= (\Phi_D^T\vec{y})^{(N-1)}_d + \phi_d(x_N)y_N\end{aligned} \tag{201}$$

The above equations show that we only need to store the previous balues of the statistics also in order to calculate the next ones, and thus this algorithm is memory efficient.

11.2 Probabilistic Classification

11.2.1 Logistic Regression (binary classification)

For the simplified case of Binary clsssification we assume for the output values that $y_n \in \{0, 1\}$. We then define an activation which is a weighted sum of the input of dimension D, and then project the activations range (for all data points) onto the $[0,1]$ range to make valid probabilities. This is done using the *Sigmoid* function, as that fits all the necessary requirements (symmetric about the y-axis, strictly between 0 and 1, monotonic and differentiable for all activation inputs).

$$\begin{aligned} a_n &= \vec{w}^T \vec{x}_n \\ p(y_n = 1|\vec{x}_n, \vec{w}) &= \sigma(a_n) = \frac{1}{1+e^{-a_n}} \end{aligned} \tag{202}$$

In this way we are effectively performing linear transforms (translational shifts and steepness, and direction for > 1 dimensional inputs) of the sigmoid function in D dimensional space by adjusting the weights, and the training process amounts to finding the transformation that allows the sigmoid function to best classify the training data (of known inputs and labels). This is best imagined using a 1D input and training the weights of the function below to correctly classify the most outputs: (w_1 effects the steepness of the probability contour, and w_0 the translation along the input axis)

$$p(y_n = 1|\vec{x}_n, \vec{w}) = \frac{1}{1+\exp(-w_0 - w_1 x)} \tag{203}$$

Because the activations are linearly related to the inputs, this method is only able to effectively classify data with a linear decision boundary (i.e. it is possible to draw a straight line through the D dimensional space to optimally classify the data). In the oft-occurring case where the data cannot be separated linearly we therefore require an alternative formulation, and this will be discussed later.

Training the model:

In the usual way, finding the optimal weights consists of maximising the log-likelihood. A binomial distribution is used for the output probability distribution given the weights and inputs as the output labels are binary (if difficult to understand why, consider the case for a single n when $y_n = 0$ or $y_n = 1$ and see how it simplifies to the expression above (or 1 - the expression above if $y_n = 0$)). Also note the use of independence between data points below:

$$\begin{aligned} p(\{y_n\}_{n=1}^N|\{\vec{x_n}\}_{n=1}^N, \vec{w}) &= \prod_{n=1}^N \sigma(a_n)^{y_n}(1-\sigma(a_n))^{1-y_n} \\ \mathcal{L}(\vec{w}) &= \ln p(\{y_n\}_{n=1}^N|\{\vec{x_n}\}_{n=1}^N, \vec{w}) \\ &= -\sum_{n=1}^N [y_n \ln\sigma(a_n) + (1-y_n)\ln(1-\sigma(a_n))] \\ \frac{\partial\mathcal{L}}{\partial\vec{w}} &= \sum_{n=1}^N \left[y_n - \sigma(\vec{w}^T\vec{x}_n)\right]\vec{x}_n \end{aligned} \tag{204}$$

If we were able to, we would now want to set the derivative to 0 and solve to find the maximum likelihood weights. However there is no known analytic solution for the above, so we are required to iterate and optimise instead, using the procedure of *gradient ascent.* This means that for a given step-size parameter η, we can updates the weights as:

$$\vec{w}_{i+1} = \vec{w}_i + \eta\frac{\partial\mathcal{L}}{\partial\vec{w}_i} \tag{205}$$

The step-size must be chosen before training and therefore gives an element of parameter tuning needed. In general though we would want it to be small enough that the weights do not change too much as to overshoot the maximum, but not too small that convergence takes too long. Similarly to other parameter tuning cases in ML, methods like momentum can be used to give faster convergence.

Aside: Reward matrices for class switching probability boundaries

Let y be the binary prediction from the model, and $\hat{y}$ the true label. Then define $R(y,\hat{y})$ to be the reward (or cost if wrong) of predicting y when the true value is $\hat{y}$. We can then define a 2x2 matrix:

$$R = \begin{bmatrix} R(y=0,\hat{y}=0) & R(y=0,\hat{y}=1) \\ R(y=1,\hat{y}=0) & R(y=1,\hat{y}=1) \end{bmatrix} \tag{206}$$

Exactly on the probability threshold where we switch from predicting 0 to 1 or vice versa, we want the rewards of predicting either 0 or 1 to be the same, and therefore we marginalise over the (unknown) true values to give:

$$
\begin{aligned}
R(y=0) &= \sum_{\hat{y}\in\{0,1\}} R(y=0,\hat{y})p(\hat{y}|\vec{x},W) = \sum_{\hat{y}\in\{0,1\}} R(y=0,\hat{y})\rho \\
R(y=1) &= \sum_{\hat{y}\in\{0,1\}} R(y=1,\hat{y})p(\hat{y}|\vec{x},W) = \sum_{\hat{y}\in\{0,1\}} R(y=1,\hat{y})(1-\rho)
\end{aligned}
\tag{207}
$$

so that

$$
R(y=0) = R(y=1) \tag{208}
$$

$$
\rho = \frac{\sum_{\hat{y}\in\{0,1\}} R(y=1,\hat{y})}{\sum_{\hat{y}\in\{0,1\}} [R(y=1,\hat{y}) + R(y=0,\hat{y})]}
$$

is the probability of switching from 0 to 1 (note that if the rewards are equal, then this is 0.5 as assumed before).

11.2.2 Multi-Class Classification

In the case that we have more than two output classes the logistic regression method can be generalised by making use of *one-hot encoded* outputs (where each output is a vector of 0s with a 1 at the index k of that class). For similar reasoning to choosing the sigmoid function as the probability rage projection in logistic regression, a favourite for multiclass classification is the *softmax* function:

$$
p(\vec{y}_n = \vec{k}|\vec{x}_n, W) = \frac{\exp(a_{n,k})}{\sum_{k=1}^{\hat{K}} \exp(a_{n,k})} = \text{Softmax}(a_{n,k}) \tag{209}
$$

where $\vec{a}_n$ is an activation that is dictated by the weights and input. Note that the dimension of the outputs and inputs is, in general, not the same. The weights vector has also now been changed to a matrix to account for the fact that there is a different set vector of weights connecting the input dimensions to each class of the output. The activation for each n is also now a vector, with the k^{th} value being the activation for the k^{th} class. This dependency is most simply visualised as a shallow feed-forward network (no hidden layers) - see Figure 8. and in fact if we were to add hidden layers, this multiclass classification method becomes identical to that of a feedforward neural network.

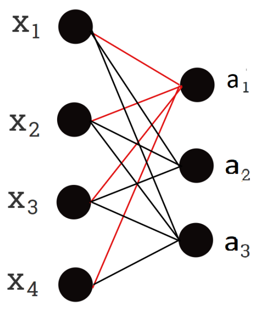


Figure 8: Visualisation of a multiclass classification for each n showing how each output class (if one-hot encoded) depends on all input vector dimensions through separate weights.

For linear multiclass classification the activations are calculated as a linear sum of the weighted input dimensions, and from Figure 8 it can be seen that this is done (in the case of 3 output classes and 4 input dimensions) as:

$$
\begin{aligned}
\vec{a}_n &= W\vec{x}_n \\
W &= \begin{bmatrix} w_1^{(1)} & w_2^{(1)} & w_3^{(1)} & w_4^{(1)} \\ w_1^{(2)} & w_2^{(2)} & w_3^{(2)} & w_4^{(2)} \\ w_1^{(3)} & w_2^{(3)} & w_3^{(3)} & w_4^{(3)} \end{bmatrix}
\end{aligned}
\tag{210}
$$

Combining Equations (209) and (210) to find the probability for a single data entry, and then taking the joint distribution over all outputs given the weights and inputs, we get: (note again that we can exploit the independence between all data points' a posteriori distributions)

$$
\begin{aligned}
p\left(\{\vec{y}_n\}_1^N | \{\vec{x}_n\}_1^N, W\right) &= \prod_{n=1}^{N} p(\vec{y}_n|\vec{x}_n, W) \\
&= \prod_{n=1}^{N} \left\{ \prod_{k=1}^{\hat{K}} p(\vec{y}_n = \vec{k}|\vec{x}_n, W)^{y_{n,k}} \right\} \\
&= \prod_{n=1}^{N} \left\{ \prod_{k=1}^{\hat{K}} \text{Softmax}(a_{n,k})^{y_{n,k}} \right\}
\end{aligned} \tag{211}
$$

If going from the first to second line of the equation above doesn't look overly clear, then substituting the one-hot encoded values of $y_{n,k}$: if $y_{n,k} = 0$, then exponentiating to the power 0 means that index contributes a product of 1, i.e. it does not contribute to the product as y_n is not that value, it only contributes if $y_{n,k} = 1$ and y_n is that class. For example if $N = 1$ and $\hat{K} = 4$ and $y_1 = 2$:

$$
\begin{aligned}
p(\vec{y}_1|\vec{x}_1, W) = \prod_{k=1}^{4} \text{Softmax}(a_{1,k})^{y_{1,k}} &= S(a_{1,1})^0 S(a_{1,2})^1 S(a_{1,3})^0 S(a_{1,4})^0 \\
&= S(a_{1,2}) = p(y_1 = 2|\vec{x}_1, W)
\end{aligned}
$$

In other words, the exponentiating is only a computational process to select the correct class probability.

Training the model:

First, as always, calculate and try to minimise log-likelihood:

$$
\begin{aligned}
\mathcal{L}(\vec{W}) &= \sum_{n=1}^{N} \left\{ \sum_{k=1}^{\hat{K}} y_{n,k} \ln S(a_{n,k}) \right\} \\
\frac{\partial \mathcal{L}(W)}{\partial \vec{w}_k} &= \sum_{n=1}^{N} \left[y_{n,k} - S(a_{n,k}) \right] \vec{x}_n \\
\text{If } W &= \begin{pmatrix} \vec{w}_1^T \\ \vec{w}_2^T \\ \dots \end{pmatrix}
\end{aligned} \tag{212}
$$

As with the case of logistic regression, this derivative doesn't have an analytic solution, so we will again have to iterate and optimise using:

$$
\vec{w}_k^{(i+1)} = \vec{w}_k^{(i)} + \eta \frac{\partial \mathcal{L}}{\partial \vec{w}_k^{(i)}} \tag{213}
$$

Aside: Softmax as an argmax approximation

As $|\vec{w}_k| \to \infty$, the softmax function approaches the argmax function because $\exp(a_{n,k})$ tends to infinity relative to all other $\exp(a_{n,j}),\ j \neq k$, so the normalised softmax function becomes a one-hot encoding of the argmax function. **Proof:**

$$
\begin{aligned}
S([W\vec{x}_n]_k) &= \frac{\exp(\vec{w}_k^T \vec{x}_n)}{\sum_{k'=1}^{\hat{K}} \exp(\vec{w}_{k'}^T \vec{x}_n)} \\
&= \frac{\exp([\vec{w}_k^T - \vec{w}_i^T]\vec{x}_n)}{\sum_{k'=1}^{\hat{K}} \exp([\vec{w}_{k'}^T - \vec{w}_i^T]\vec{x}_n)} \\
\lim_{|\vec{w}_i| \to \infty} S([W\vec{x}_n]_k) &= \lim_{|\vec{w}_i| \to \infty} \frac{\exp([\vec{w}_k^T - \vec{w}_i^T]\vec{x}_n)}{\sum_{k'=1}^{\hat{K}} \exp([\vec{w}_{k'}^T - \vec{w}_i^T]\vec{x}_n)} \\
&= \frac{\delta_{ik}}{\sum_{k'=1}^{\hat{K}} \delta_{ik'}} \\
&= \begin{cases} 1 & \text{if } i = k \\ 0 & \text{otherwise} \end{cases}
\end{aligned} \tag{214}
$$

Therefore if we pass through the vector form (over all k) of the input into the softmax function and take limit $|\vec{w}_i| \to \infty$, we get that the output is a one-hot encoded vector where the i^{th} element in 1 and the rest 0 - i.e. we get the one-hot encoding of the index that has the maximum weight:

$$\lim_{|\vec{w}_i| \to \infty} S(W\vec{x}_n) = \arg\max_i W\vec{x}_n, \text{ Q.E.D.}$$

Aside: Also if we make $\hat{K} = 2$, we can cancel out common terms in the softmax function to easily return the logistic regression expression of the binary classification case.

11.2.3 Non-Linear Classification

For generality we will continue working with multi-class labelled data. The change from linear to non-linear models is the same as in non-linear regression, and we assume a set of D basis functions $\{\phi_d(\vec{x})\}_{d=1}^{D}$ s.t.:

$$\begin{aligned} \vec{a}_n &= W\vec{\phi}(\vec{x}_n) \\ \vec{\phi}(\vec{x}_n) &= \begin{bmatrix} 1 & \phi_1(\vec{x}_n) & \cdots & \phi_D(\vec{x}_n) \end{bmatrix}^T \end{aligned} \tag{215}$$

The use of the softmax function for the probability projection is then the same except for the new definition of the activations, and the gradient ascent training is now:

$$\vec{w}_k^{(i+1)} = \vec{w}_k^{(i)} + \eta \sum_{n=1}^{N} \left[y_{n,k} - S(a_{n,k})\right] \vec{\phi}(\vec{x}_n) \tag{216}$$

<u>*Choosing bases:*</u>

The choice of basis function now becomes a major consideration in the method, and a very general choice is to choose functions that produce *tiling* - a set of functions with centres that are distributed (usually uniformly) throughout the input space. For example, the radial exponential decay basis:

$$\phi_d(\vec{x}_n) = \exp\left\{-\frac{1}{2l^2}|\vec{x}_n - \vec{\mu}_d|^2\right\} \tag{217}$$

Here $\vec{\mu}_d$ is fixed for each d, and l is a length parameter constant for all functions. The reason this algorithm works is that it decays exponentially, so only the basis functions with centres within l of $\vec{x}_n$ will contribute non-negligibly to $\vec{a}_n$, and thus that function's respective weight can be chosen depending on whether the $\vec{x}_n$'s within l from its centre should make $a_{n,k}$ more positive or more negative (and therefore make $y_{n,k}$ closer to 0 or 1 respectively).

<u>*Overfitting:*</u>

However, this makes it very easy to overfit when l is small (so each of the basis functions become unique to each training input data), and when D is large (so that the basis functions cover the whole input space). In this case the training process simply learns a sort-of one-to-one mapping of training inputs to functions, so that each individual weight is unique to each training input point. The model will then perform more poorly in testing, as the input points are different and so the model cannot map them as confidently (i.e. the probability contour is too steep around the training data inputs). This can be seen mathematically, as in training, the weights that are associated to basis functions that are *not* within l of a training point become negligible in magnitude compared to those that are - therefore:

$$\vec{a}_n = W\vec{\phi}(\vec{x}^*) \approx 0$$

and

$$p(\vec{y}^*|\vec{x}^*, W) = \text{Softmax}(W\vec{\phi}(\vec{x}^*)) \approx \frac{1}{\hat{K}}$$

representing a uniform distribution representing a lack of information even after training.

More case-specific bases can be chosen that are less prone to overfitting, and that do not have an exponentially increasing D with dimensions like the tiling method does.

11.3 Clustering

11.3.1 *k*-means clustering

The algorithm for k-means clustering is deterministic rathre than probabilistic, and is a good (for certain assumptions discussed below) approximation for the NP-hard problem of finding the optimal clusters of a set of data. The algorithm consists of allocating K means in the data, and then fitting the data to their nearest mean, before recalibrating the mean positions of the data points now belonging to that class.

Mathematically this can be executed ad follows: we first define an ownership matrix S s.t.

$$S_{n,k} := 1 \text{ if } \vec{x}_n \in \text{ cluster } k \tag{218}$$

for example, for 3 data points belonging to one of 2 clusters:

$$S = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

<u>*Cost Minimisation:*</u>

We now define a cost function that we wish to minimise, and this can be one of many cost functions, but the most intuitive is the absolute squared error:

$$C_2 := \sum_{n=1}^{N} \left\{ \sum_{k=1}^{K} S_{n,k} |\vec{x}_n - \vec{m}_k|^2 \right\} \tag{219}$$

where $\vec{m}_k$ is the mean position of the k^{th} cluster, and the sum over k only includes data points n for that k if they belong to that cluster due to the $S_{n,k}$ selection process.

This cost can be minimised in 2 steps as it consists of 2 independent factors:

1. Minimise C_2 w.r.t. $S_{n,k}$ whilst holding $\{\vec{m}_k\}$ constant by putting $\vec{x}_n$s in the clusters with the closest mean position:
$$S_{n,k} = \delta_{kk'} \text{ for } k' = \arg\min_k |\vec{x}_n - \vec{m}_k|^2 \tag{220}$$

2. Minimise C_2 w.r.t. $\{\vec{m}_k\}$ whilst holding S constant:
$$\begin{aligned} \frac{\partial C_2}{\partial \vec{m}_k} &= 2\sum_{n=1}^{N} S_{n,k}(\vec{x}_n - \vec{m}_k) = 0 \\ \vec{m}_k &= \frac{\sum_n S_{n,k}\vec{x}_n}{\sum_n S_{n,k}} = \frac{\sum_{\vec{x}_n \in k} \vec{x}_n}{N_k} \end{aligned} \tag{221}$$
Notice how this definition above of the optimal mean of the k^{th} cluster is simply the mean position of all of the data points belonging to that cluster.

3. Repeat the two steps above until convergence, which happens when $S^{(i+1)} - S^{(i)} = \underline{0}$.

There are however multiple drawbacks to this method. largely due to it being deterministic and overly simple. The most obvious are:

- The number of clusters K must be chosen before by the user, and this is not ideal for a supposedly unsupervised algorithm. There are methods to infer the number of clusters, such as the Dirichlet Process Mixture Model, but they are mathematically and computationally a lot more complex.

- The C_2 (and any C_p) cost function penalises radial distance from data points to the means, and therefore the points are always clustered in a circle (in 2D, sphere in 3D, etc...). This leads to large failures of the algorithm on non-circular or different shaped clusters. In order to change cluster shape, we need an idea of the covariance of the data points in the clusters, and therefore can use a *'soft'* probabilistic model (as discussed in the next subsection).

- Random initialisations of the means might make the cost function converge on a local minimum rather than a global one, and if the means are initialised very close together, this is likely to not be a very satisfactory minimum. This problem is fixed in part by the k-means ++ algorithm below.

Aside: We are always guaranteed to get convergence for C_2, even if not to a global minimum. This is because $C_2 \geq 0 \; \forall \; \vec{x}_n, S, \{\vec{m}_k\}$, and the minimisation algorithm must either decrease the cost function or keep it the same. Therefore we are guaranteed over an infinite number of iterations to converge on a minimum of C_2.

Initialisation - the k-means ++ algorithm:

The assumption of this algorithm is that an improvement (of competitiveness O(log k) compared to the NP-hard optimal solution) can be found by increasing the spread of the initial mean positions. The algorithm is different from random initialisation in that it uses the data points *themselves* as positions for the means, and is as follows:

1. Randomly and uniformly select a data point and set this as the first mean.
2. With all the other *non-selected* data points, define $\{D_{N-1}(\vec{x}_n)\}$ as the distances from the first mean to the n^{th} data point.
3. Create a weighting distribution proportional to $D_{N-1}(\vec{x})^2$ over the remaining $(N-1)$ points, and sample the next point to be chosen as the mean from this.
4. Repeat steps 2 and 3 for the remaining means to be found, by calculating $D_{N-2}(\vec{x})$ as the total distance from the point to mean 1 and mean 2, etc...

Example: Image encoding and compression

K-means clustering can be used as a means of *lossy* compression of images, i.e. encoding them in such a way that they can be stored with fewer bits of data, but cannot be decoded back to the same quality. Consider an image built up of N pixels, with each represented by a scalar (of grey-scale image intensity) or a 3D vector (of RGB values).

We can perform k-means clustering to this set of vectors for a set number of means that controls the quality of the decoded image:

1. Perform k-means clustering to obtain the means $\{\mu_k\}$ and their membership matrix S. These alone will be recorded for the compressed image.
2. To decode the image, set the intensity/colour of each pixel to the intensity/colour of the mean that it belongs to. In this way the image is broken down into *sections* of different intensities/colours based on their location in intensity (or RGB) space. The number of means is the number of different pixel types.

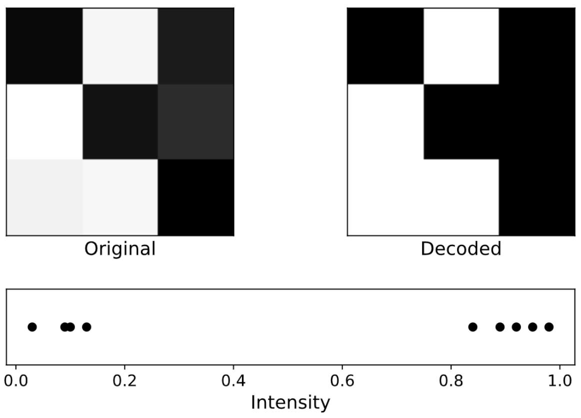


Figure 9: A simple visualisation of how 2 means can be used to encode a simple greyscale image. The clustering algorithm clusters the (1D) greyscale in intensity space, and the decoded image is then lower quality.

The memory gain of this method can be quantised, as the memory needed for a RGB image is $24N$ bits for N pixels (8 bits for each colour channel), and for the compressed image we need $(24k + N \log_2 k)$ bits, including memory for the centre and the membership information respectively.

11.3.2 Probabilistic Clustering - Gaussian Mixture Models as a Generative Model

This method employs a similar ideology to k-means clustering, but with the flexibility of non-strictly-radial covariance, and being probabilistic rather than deterministic. We set up k Gaussian distributions, each with mean and covariance $\vec{\mu}_k, \Sigma_k$, and determine which data points belong to each Gaussian. Mathematically then, for the input data $\{\vec{x}_n\}$ belonging to Gaussians $\{s_n\}$, we have:

$$\begin{aligned} p(\vec{x}_n|s_n = k) &= \mathcal{N}(\vec{\mu}_k, \Sigma_k) \\ p(s_n = k) &:= \pi_k^{(n)} \rightarrow \pi_k \end{aligned} \tag{222}$$

where we have dropped the (n) superscript for brevity. For ease of notation, let $\theta_k = \{\pi_k, \vec{\mu}_k, \Sigma_k\}$, and $\theta := \{\theta_k\}_{k=1}^K$. Using the membership matrix ($S_{n,k} = 1$ if $\vec{x}_n \in k$) as before, we can exploit input data independence to say:

$$\begin{aligned} p(\{\vec{x}_n\}_1^N|S, \{\vec{\mu}_k\}_1^K, \{\Sigma_k\}_1^K) &= \prod_{n=1}^N p(\vec{x}_n|S, \vec{\mu}_k, \Sigma_k) \\ &\propto \prod_{n=1}^N \left\{ \prod_{k=1}^K \left[\pi_k \exp\left(-\frac{1}{2}(\vec{x}_n - \vec{\mu}_k)^T \Sigma_k^{-1} (\vec{x}_n - \vec{\mu}_k) \right) \right]^{S_{n,k}} \right\} \end{aligned} \tag{223}$$

where the product over k acts as a selection criteria as before due to the $S_{n,k}$ exponent forcing the product to either contribute the probability distribution for the n^{th} data point in the k^{th} Gaussian if there is membership, or 1 otherwise.

We wish to maximise this likelihood by optimising both the membership of inputs to Gaussians, and by optimising the parameters of those Gaussians, much like with k-means clustering. Therefore we can move to the log domain and maximise the log-likelihood:

$$\hat{\theta} = \arg\max_\theta \mathcal{L}(\theta) = \arg\max_\theta \sum_n \sum_k \ln \left[\pi_k \exp\left(-\frac{1}{2}(\vec{x}_n - \vec{\mu}_k)^T \Sigma_k^{-1} (\vec{x}_n - \vec{\mu}_k) \right) \right]^{S_{n,k}} \tag{224}$$

Unfortunately, this expression is not analytically optimisable w.r.t. the parameters $\theta = \{\theta_k\}_1^K = \{\{\pi_k, \vec{\mu}_k, \Sigma_k\}\}_1^K$, and so we must use an iterative optimisation technique, similar to that of k-means clustering, but this time much more mathematically involved. This is the *Expectation Maximisation (EM)* algorithm.

Expectation-Maximisation algorithm summary:

(Please also see Section 5.4.1, where the EM algorithm was used to train Linear Gaussian Models).

The mathematical detail of the operation will be described below, but the outline of the method is as follows:

1. Pick an arbitrary distribution $q_n(s)$, and define a quantity $\mathcal{F}(q_n(s), \theta_n) \leq \mathcal{L}(\theta_n)$. ($s$ here is a membership state for an arbitrary data point that can take values $s \in [1, ..., K]$.)

2. We now hold θ_n constant, and maximise $\mathcal{F}$ w.r.t. $q_n(s)$, s.t. $\mathcal{F}(q_n(s), \theta_n) \leq \mathcal{F}(q_{n+1}(s), \theta_n) \leq \mathcal{L}(\theta_n)$. This is the **E-step**.

3. Now hold $q_{n+1}(s)$ constant and maximise w.r.t. (continuous variable) θ, s.t.:

   $$\theta_{n+1} = \arg\max_\theta \mathcal{F}(q_{n+1}(s), \theta)$$

   This is the **M-step**.

   NB: In order for the M-step to guarantee an improvement of the log-likelihood, we must have after the E-step that $\mathcal{F}(q_{n+1}(s), \theta_n) = \mathcal{L}(\theta_n)$. If this is not the case, and $\mathcal{F}$ is strictly less, then there is room for a possibility that optimising $\mathcal{F}$ in the M-step w.r.t. θ moves the parameters in a way that doesn't affect, or worse, negatively affects, $\mathcal{L}(\theta_{n+1})$. Therefore, we will always try to seek to choose and optimise the arbitrary distribution $q(s)$ s.t. $\mathcal{F}(q_{n+1}(s), \theta_n) = \mathcal{L}(\theta_n)$.

4. Repeat steps 1 and 2 until convergence (either in $\mathcal{L}$ or $\mathcal{F}$ or θ).

Mathematical Detail for the EM Algorithm:

For reasons outlines in point 3 above, we wish for $\mathcal{F}$ to be chosen so that when it is maximised w.r.t. $q(s)$ it is equal to the log-likelihood. Therefore a sensible form for it to take is:

$$\begin{aligned}\mathcal{F}_m(q(s),\theta) = \mathcal{F}(q(s_m),\theta) &= \mathcal{L}_m(\theta) - D_{KL}\left[q(s_m)||p(s_m|\vec{x}_m,\theta)\right] \\ &= \ln p(\vec{x}_m|\theta) - \sum_k q(s_m=k)\ln\left(\frac{q(s_m=k)}{p(s_m=k|\vec{x}_m,\theta)}\right)\end{aligned} \tag{225}$$

where we have included the subscript m to signify this is for a single data point, but will often drop this from here on (the log-likelihood is recovered by $\mathcal{L}(\theta) = \mathcal{L}_1(\theta) + \mathcal{L}_2(\theta) + ... + \mathcal{L}_N(\theta)$. We have also used the Kullback-Leibler divergence (see section 4.3 for more details) for its property of being strictly positive unless the two distributions are equal, in which case it is equal to 0 (proof of this concept is in 'Proof 1' below). The expression in Equation (225) is called the *Free Energy*, in relation to thermodynamics in physics.

E-step:

As discussed in point 3 above, for the E-step, we wish to maximise this free energy w.r.t. $q(s)$ in a way that makes $\mathcal{F}(q(s),\theta) = \mathcal{L}(\theta)$ so that for the M-step we are guaranteed to improve the log-likelihood. By looking at Equation (225) this can be done by setting $q(s) = p(s|\vec{x},\theta)$ so that the KL-divergence is 0. Therefore:

$$\begin{aligned}\hat{q}(s_n=k) &= \frac{p(\vec{x}_n|s_n=k,\theta)p(s_n=k|\theta)}{p(\vec{x}_n|\theta)} \\ &= \frac{1}{p(\vec{x}_n|\theta)}\frac{\pi_k}{(2\pi|\Sigma_k|)^{1/2}}\exp\left[-\frac{1}{2}(\vec{x}_n-\vec{\mu}_k)^T\Sigma_K^{-1}(\vec{x}_n-\vec{\mu}_k)\right] \\ &:= \frac{1}{p(\vec{x}_n|\theta)}u_{n,k} = \frac{u_{n,k}}{\sum_{k=1}^K u_{n,k}}\end{aligned} \tag{226}$$

where in the last line above we have used the fact that for any one data point, the sum over its membership state probabilities must sum to 1, and therefore we can normalise.

M-step:

We now wish to fix $\hat{q}(s_n)$ and maximise the free energy w.r.t. the parameters $\theta = \{\{\vec{\mu}_k, \Sigma_k, \pi_k\}\}_1^K$. After some rearranging (see 'Proof 2' below for details), we get:

$$\begin{aligned}\hat{\theta} &= \arg\max_\theta \mathcal{F}(\hat{q}(s),\theta) = \arg\max_\theta \sum_n \mathcal{F}_n(\hat{q}(s_n),\theta) \\ &= \arg\max_\theta \sum_n\sum_k \hat{q}(s_n=k)\ln\left[\frac{p(s_n=k,\vec{x}|\theta)}{\hat{q}(s_n=k)}\right] \\ &= \arg\max_\theta \sum_n\left\{\sum_k \hat{q}(s_n=k)\ln p(s_n=k,\vec{x}|\theta) - \sum_k \hat{q}(s_n=k)\ln\hat{q}(s_n=k)\right\} \\ &= \arg\max_\theta \sum_n\sum_k \hat{q}(s_n=k)\ln p(s_n=k,\vec{x}|\theta) \\ &= \arg\max_\theta \sum_n\sum_k \hat{q}(s_n=k)\ln\left[p(\vec{x}|s_n=k,\theta)p(s_n=k|\theta)\right] \\ &= \arg\max_\theta \sum_n\sum_k \hat{q}(s_n=k)\ln\left\{\frac{\pi_k}{(2\pi|\Sigma_k|)^{1/2}}\exp\left[-\frac{1}{2}(\vec{x}_n-\vec{\mu}_k)^T\Sigma_K^{-1}(\vec{x}_n-\vec{\mu}_k)\right]\right\} \\ &= \arg\max_\theta \sum_n\sum_k \hat{q}(s_n=k)\left[\ln\pi_k - \frac{1}{2}\ln|\Sigma_k| - \frac{1}{2}(\vec{x}_n-\vec{\mu}_k)^T\Sigma_k^{-1}(\vec{x}_n-\vec{\mu}_k)\right]\end{aligned} \tag{227}$$

where the argmax can be found for each parameter analytically by differentiating w.r.t. the parameters and finding the root. The solutions are given by (see 'Proof 3' below for proof):

$$\begin{aligned}
\hat{\vec{\mu}}_k &= \frac{1}{N_k}\sum_n \hat{q}(s_n = k)\vec{x}_n \\
\hat{\Sigma}_k &= \frac{1}{N_k}\sum_n \hat{q}(s_n = k)(\vec{x}_n - \vec{\mu}_k)^T(\vec{x}_n - \vec{\mu}_k) \\
\hat{\pi}_k &= \frac{N_k}{N}
\end{aligned} \tag{228}$$

for:

$$N_k = \sum_n \hat{q}(s_n = k) \tag{229}$$

$$\hat{q}(s_n = k) = \frac{u_{n,k}}{\sum_{k=1}^K u_{n,k}} \propto \frac{\pi_k}{(2\pi|\Sigma_k|)^{1/2}} \exp\left[-\frac{1}{2}(\vec{x}_n - \vec{\mu}_k)^T \Sigma_K^{-1}(\vec{x}_n - \vec{\mu}_k)\right]$$

Summary:

Therefore, to summarise, the EM algorithm computes the following:

1. Initialise $q(s_n = k)$ and $\theta_k = \{\pi_k, \vec{\mu}_k, \Sigma_k\}$ for each k.
2. For the E-step, compute:

$$\boxed{u_{n,k} = \frac{\pi_k}{(2\pi|\Sigma_k|)^{1/2}} \exp\left[-\frac{1}{2}(\vec{x}_n - \vec{\mu}_k)^T \Sigma_K^{-1}(\vec{x}_n - \vec{\mu}_k)\right]}$$

$$\boxed{\hat{q}(s_n = k) = \frac{u_{n,k}}{\sum_{k=1}^K u_{n,k}}}$$

3. For the M-step, compute:

$$\boxed{\hat{\vec{\mu}}_k = \frac{1}{N_k}\sum_n \hat{q}(s_n = k)\vec{x}_n}$$

$$\boxed{\hat{\Sigma}_k = \frac{1}{N_k}\sum_n \hat{q}(s_n = k)(\vec{x}_n - \vec{\mu}_k)(\vec{x}_n - \vec{\mu}_k)^T}$$

$$\boxed{\hat{\pi}_k = \frac{N_k}{N}}$$

4. Repeat steps 2 and 3 until convergence (of the free energy) or until a certain number of steps is reached.
5. The optimal q's are the (soft) optimised membership probabilities:

$$\hat{q}(s_n) = p(s_n|\vec{x}_n, \theta)$$

Proof 1: proving that the KL-divergence is greater than or equal to 0.

For this proof we will use the method of *Lagrange Multipliers* and show the conditions for an extremum give equal probability distributions and therefore 0 Kullback-Leibler divergence. We will then show that this extremum has to be a minimum always.

$$\begin{aligned}
D_{KL}(p(x)||q(x)) &= -\int p(x)\ln\left[\frac{q(x)}{p(x)}\right]dx \\
&= -\int p(x)\ln q(x)dx + \int p(x)\ln p(x)dx + \lambda\left\{1 - \int p(x)dx\right\} \\
\frac{\partial}{\partial p(x)}D_{KL}(p(x)||q(x)) &= -\frac{\partial}{\partial p(x)}\int p(x)\ln q(x)dx + \frac{\partial}{\partial p(x)}\int p(x)\ln p(x)dx - \lambda \\
&= -\ln q(x) + \ln p(x) + 1 - \lambda = 0
\end{aligned} \tag{230}$$

Therefore the extremum of KL-divergence is reached when $p(x) = q(x)\exp(\lambda - 1)$, and as both $p(x)$ and $q(x)$ have to integrate to 1, $\lambda = 1$ and $p(x) = q(x)$ Q.E.D.

Now to identify this extremum:

$$\frac{\partial^2}{\partial p(x)^2} D_{KL}(p(x)||q(x)) = \frac{1}{p(x)} > 0 \;\forall\; 0 \le p(x) \le 1 \tag{231}$$

Therefore, if the second derivative is strictly positive for all $p(x)$, then the extremum at $p(x) = q(x)$ must be a minimum. Therefore $D_{KL}(p(x)||q(x)) \ge 0$ always.

Proof 2: proving that the free energy can be rearranged as shown.

We start with the definition of the free energy from Equation (225):

$$\begin{aligned}
\mathcal{F}(\hat{q}(s), \theta) &= \sum_n \mathcal{F}_n(\hat{q}(s_n), \theta), \text{ where:} \\
\mathcal{F}_n(\hat{q}(s_n), \theta) &= \ln p(\vec{x}_n|\theta) - \sum_k \hat{q}(s_n = k) \ln \left[\frac{\hat{q}(s_n = k)}{p(s_n = k|\vec{x}_n, \theta)}\right] \\
&= \ln p(\vec{x}_n|\theta) + \sum_k \hat{q}(s_n = k) \ln \left[\frac{p(s_n = k, \vec{x}_n|\theta)}{p(\vec{x}_n|\theta)\hat{q}(s_n = k)}\right] \\
&= \ln p(\vec{x}_n|\theta) + \sum_k \hat{q}(s_n = k) \ln \left[\frac{p(s_n = k, \vec{x}_n|\theta)}{\hat{q}(s_n = k)}\right] - \sum_k \hat{q}(s_n = k) \ln p(\vec{x}_n|\theta) \\
&= \ln \left[\frac{p(\vec{x}_n|\theta)}{p(\vec{x}_n|\theta)}\right] + \sum_k \hat{q}(s_n = k) \ln \left[\frac{p(s_n = k, \vec{x}_n|\theta)}{\hat{q}(s_n = k)}\right] \\
&= \sum_k \hat{q}(s_n = k) \ln \left[\frac{p(s_n = k, \vec{x}_n|\theta)}{\hat{q}(s_n = k)}\right] \\
\therefore\; \mathcal{F}(\hat{q}(s), \theta) &= \sum_n \sum_k \hat{q}(s_n = k) \ln \left[\frac{p(s_n = k, \vec{x}_n|\theta)}{\hat{q}(s_n = k)}\right] \;\text{ Q.E.D.}
\end{aligned} \tag{232}$$

where we have used the E-step result of $\hat{q}(s_n = k) = p(s_n = k|\vec{x}, \theta)$ and therefore that it sums over k to 1 (note that $p(\vec{x}_n|\theta)$ is independent of k and therefore not in the sum).

Proof 3: proving that maximising free energy w.r.t. each parameter gives the states results.

$\hat{\pi}_{\mathbf{k}}$: Firstly for the optimal parameter of $\hat{\pi}_k$, we use the method of Lagrange multipliers once more for the restriction that $\sum_k \pi_k = 1$:

$$\begin{aligned}
\frac{\partial \mathcal{F}(\hat{q}(s), \theta)}{\partial \pi_k} &= \frac{\partial}{\partial \pi_k}\left[\mathcal{F} + \lambda\left(\sum_k \pi_k - 1\right)\right] \\
&= \frac{\partial}{\partial \pi_k}\left[\sum_n \sum_k \hat{q}(s_n = k)\left[\ln \pi_k - \frac{1}{2}\ln|\Sigma_k| - \frac{1}{2}(\vec{x}_n - \vec{\mu}_k)^T \Sigma_k^{-1}(\vec{x}_n - \vec{\mu}_k)\right] + \lambda\left(\sum_k \pi_k - 1\right)\right] \\
&= \frac{1}{\pi_k}\sum_n \hat{q}(s_n = k) + \lambda = 0
\end{aligned} \tag{233}$$

And so by rearranging and summing over states we get:

$$\begin{aligned}
-\lambda \pi_k &= \sum_n \hat{q}(s_n = k) \\
-\lambda \sum_k \pi_k = -\lambda &= \sum_k \sum_n \hat{q}(s_n = k) \\
\therefore\; \hat{\pi}_k &= \frac{\sum_n \hat{q}(s_n = k)}{\sum_n \sum_k \hat{q}(s_n = k)} = \frac{1}{N}\sum_n \hat{q}(s_n = k) \text{ Q.E.D.}
\end{aligned}$$

where we have used the E-step result of $\hat{q}(s_n = k) = p(s_n = k|\vec{x}_n, \theta)$ and that it sums to 1 over k space.

$\hat{\vec{\mu}}_{\mathbf{k}}$: Next for finding the optimal k^{th} cluster mean, $\vec{\mu}_k$:

$$\begin{aligned}\frac{\partial\mathcal{F}}{\partial\vec{\mu}_k} &= \frac{\partial}{\partial\vec{\mu}_k}\sum_n\sum_k \hat{q}(s_n=k)\left[\ln\pi_k - \frac{1}{2}\ln|\Sigma_k| - \frac{1}{2}(\vec{x}_n-\vec{\mu}_k)^T\Sigma_k^{-1}(\vec{x}_n-\vec{\mu}_k)\right]\\ &= \Sigma_k^{-1}\sum_n \hat{q}(s_n=k)(\vec{x}_n-\vec{\mu}_k) = 0\end{aligned} \tag{234}$$

Which leave us with:

$$\hat{\vec{\mu}}_k = \frac{\sum_n \hat{q}(s_n=k)\vec{x}_n}{\sum_n \hat{q}(s_n=k)} = \frac{1}{N_k}\sum_n \hat{q}(s_n=k)\vec{x}_n \text{ Q.E.D.} \tag{235}$$

$\hat{\mathbf{\Sigma}}_{\mathbf{k}}$: And finally, we can repeat a similar method for $\hat{\Sigma}_k$:

$$\begin{aligned}\frac{\partial\mathcal{F}}{\partial\Sigma_k} &= \frac{\partial}{\partial\Sigma_k}\sum_n\sum_k \hat{q}(s_n=k)\left[\ln\pi_k - \frac{1}{2}\ln|\Sigma_k| - \frac{1}{2}(\vec{x}_n-\vec{\mu}_k)^T\Sigma_k^{-1}(\vec{x}_n-\vec{\mu}_k)\right]\\ &= -\frac{1}{2}\sum_n \hat{q}(s_n=k)\frac{\partial}{\partial\Sigma_k}\ln|\Sigma_k| - \frac{1}{2}\sum_n \hat{q}(s_n=k)\frac{\partial}{\partial\Sigma_k}(\vec{x}_n-\vec{\mu}_k)^T\Sigma_k^{-1}(\vec{x}_n-\vec{\mu}_k)\end{aligned} \tag{236}$$

Unfortunately, the two derivates in the bottom line above are not particularly simple to differentiate, so for ease of reading they will be calculated separately and then substituted. For the first derivative to be considered: (here $\lambda_d,\ d=1,...,D$ are the eigenvalues of Σ_k)

$$\begin{aligned}\frac{\partial}{\partial\Sigma_k}\ln|\Sigma_k| &= \frac{\partial}{\partial\Sigma_k}\ln\prod_{d=1}^{D}\lambda_d = \frac{\partial}{\partial\Sigma_k}\sum_{d=1}^{D}\ln\lambda_d\\ &= \sum_{d=1}^{D}\frac{\partial\lambda_d}{\partial\Sigma_k}\frac{\partial}{\partial\lambda_d}\ln\lambda_d = \sum_d \frac{1}{\lambda_d}\frac{\partial\lambda_d}{\partial\Sigma_k}\end{aligned} \tag{237}$$

Now we notice that Σ_k is symmetric, and thus decomposable into $\Sigma_k = \sum_d \lambda_d\vec{u}_d\vec{u}_d^T$ and $\Sigma_k^{-1} = \sum_d \frac{1}{\lambda_d}\vec{u}_d\vec{u}_d^T$ (using projection into its own eigenspace), so that for any arbitrary x it is true that:

$$\begin{aligned}\Sigma_k^{-1}\frac{\partial\Sigma_k}{\partial x} &= \sum_{d'}\frac{1}{\lambda_{d'}}\vec{u}_{d'}\vec{u}_{d'}^T\sum_d\frac{\partial\lambda_d}{\partial x}\vec{u}_d\vec{u}_d^T\\ &= \sum_d\sum_{d'}|u_{d'}\rangle\langle u_{d'}|u_d\rangle\langle u_d|\frac{1}{\lambda_{d'}}\frac{\partial\lambda_d}{\partial x}\\ &= \sum_d\sum_{d'}|u_{d'}\rangle\delta_{dd'}\langle u_d|\frac{1}{\lambda_{d'}}\frac{\partial\lambda_d}{\partial x} = \sum_d |u_d\rangle\langle u_d|\frac{1}{\lambda_d}\frac{\partial\lambda_d}{\partial x}\\ \mathrm{Tr}\left(\Sigma_k^{-1}\frac{\partial\Sigma_k}{\partial x}\right) &= \mathrm{Tr}\left(\sum_d |u_d\rangle\langle u_d|\frac{1}{\lambda_d}\frac{\partial\lambda_d}{\partial x}\right) = \sum_d\frac{1}{\lambda_d}\frac{\partial\lambda_d}{\partial x}\end{aligned} \tag{238}$$

where we have switched in to bra-ket notation for ease of reading, and used the critical properties of $\vec{u}_d$ being **orthonormal** eigenvectors (hence the delta operator in the third line of the equation above). This final expression is exactly what we have in Equation (237) above, when $x=\Sigma_k$, so we conclude that:

$$\frac{\partial}{\partial\Sigma_k}\ln|\Sigma_k| = \mathrm{Tr}\left(\Sigma_k^{-1}\frac{\partial\Sigma_k}{\partial\Sigma_k}\right) \tag{239}$$

Next is to find an expression for the second of the derivatives we need for Equation (236). To do this we can use the following relation:

$$\begin{aligned}\frac{\partial}{\partial x}\Sigma_k\Sigma_k^{-1} &= \frac{\partial \hat{I}}{\partial x} = 0 \\ &= \Sigma_k\frac{\partial \Sigma_k^{-1}}{\partial x} + \frac{\partial \Sigma_k}{\partial x}\Sigma_k^{-1} \\ \therefore \;\; \frac{\partial \Sigma_k^{-1}}{\partial x} &= -\Sigma_k^{-1}\frac{\partial \Sigma_k}{\partial x}\Sigma_k^{-1}\end{aligned} \tag{240}$$

so now we can compare this to what is needed form Equation (236) to give:

$$\begin{aligned}\sum_n \hat{q}(s_n = k)\frac{\partial}{\partial \Sigma_k}(\vec{x}_n - \vec{\mu}_k)^T\Sigma_k^{-1}(\vec{x}_n - \vec{\mu}_k) &= \sum_n \hat{q}(s_n = k)(\vec{x}_n - \vec{\mu}_k)^T\frac{\partial \Sigma_k^{-1}}{\partial \Sigma_k}(\vec{x}_n - \vec{\mu}_k) \\ &= -\sum_n \hat{q}(s_n = k)(\vec{x}_n - \vec{\mu}_k)^T\Sigma_k^{-1}\frac{\partial \Sigma_k}{\partial \Sigma_k}\Sigma_k^{-1}(\vec{x}_n - \vec{\mu}_k) \\ &= \mathrm{Tr}\left(-\Sigma_k^{-1}\frac{\partial \Sigma_k}{\partial \Sigma_k}\Sigma_k^{-1}\sum_n \hat{q}(s_n = k)(\vec{x}_n - \vec{\mu}_k)(\vec{x}_n - \vec{\mu}_k)^T\right)\end{aligned} \tag{241}$$

With these two derivatives calculated, we now return to Equation (236) to show that minimising w.r.t. Σ_k is equivalent to saying (note the use of linearity of the trace in evaluating the sum over n):

$$\begin{aligned}\sum_n \hat{q}(s_n = k)\mathrm{Tr}\left(\Sigma_k^{-1}\frac{\partial \Sigma_k}{\partial \Sigma_k}\right) + \mathrm{Tr}\left(-\Sigma_k^{-1}\frac{\partial \Sigma_k}{\partial \Sigma_k}\Sigma_k^{-1}\sum_n \hat{q}(s_n = k)(\vec{x}_n - \vec{\mu}_k)(\vec{x}_n - \vec{\mu}_k)^T\right) &= 0 \\ \mathrm{Tr}\left(\sum_n \hat{q}(s_n = k)\Sigma_k^{-1}\frac{\partial \Sigma_k}{\partial \Sigma_k} - \Sigma_k^{-1}\frac{\partial \Sigma_k}{\partial \Sigma_k}\Sigma_k^{-1}\sum_n \hat{q}(s_n = k)(\vec{x}_n - \vec{\mu}_k)(\vec{x}_n - \vec{\mu}_k)^T\right) &= 0\end{aligned} \tag{242}$$

If the trace of a matrix is 0 in its own eigenspace, then that matrix must also be equal to 0, meaning that:

$$\begin{aligned}\sum_n \hat{q}(s_n = k)\Sigma_k^{-1}\cancel{\frac{\partial \Sigma_k}{\partial \Sigma_k}} &= \Sigma_k^{-1}\cancel{\frac{\partial \Sigma_k}{\partial \Sigma_k}}\Sigma_k^{-1}\sum_n \hat{q}(s_n = k)(\vec{x}_n - \vec{\mu}_k)(\vec{x}_n - \vec{\mu}_k)^T \\ N_k\Sigma_k^{-1} &= \left(\Sigma_k^{-1}\right)^2\sum_n \hat{q}(s_n = k)(\vec{x}_n - \vec{\mu}_k)(\vec{x}_n - \vec{\mu}_k)^T \\ N_k\Sigma_k &= \sum_n \hat{q}(s_n = k)(\vec{x}_n - \vec{\mu}_k)(\vec{x}_n - \vec{\mu}_k)^T \\ \hat{\Sigma}_k &= \frac{1}{N_k}\sum_n \hat{q}(s_n = k)(\vec{x}_n - \vec{\mu}_k)(\vec{x}_n - \vec{\mu}_k)^T \text{ Q.E.D.}\end{aligned} \tag{243}$$

Above we have used the prior definition of $N_k = \sum_n \hat{q}(s_n = k)$.

11.4 Gaussian Processes

Gaussian process regression is a *function-space* representation of the Bayesian regression shown in Section 11.1.3. Although it talks about a function space, we do not define any functions, as that would require parameters, whereas this is a *non-parametric* (∞*-parametric*) model. Instead, the intuition is that we take slices through the x-axis at a (theoretically infinite) set of values $\{x^*\}$, and sample where a function (which has to pass only once left to right) would cross each slice - this is the position f^*. If the positions are correlated with each other (i.e. a covariance function $K(x, x') \neq 0$), then the crossing value of one slice alters the distribution (which is assumed Gaussian for its consistency property) of all others, with the extent of control given by the magnitude of the covariance.

Mathematically this is denoted:

$$\vec{f}(x) \sim \mathcal{GP}\left(\vec{m}(x) = 0, K(x, x')\right) \tag{244}$$

where $\vec{m}$ and K are infinite and (infinite x infinite) dimension mean and covariances respectively. It can be thought of as an *infinite dimensional Gaussian, where any finite subset of* $\vec{f}$*'s is itself a finite multivariate Gaussian*: $\vec{f} \sim \mathcal{N}\left(\vec{m}(x) = 0, K(x, x')\right)$.

Noiseless (Conditional) Predictive Distribution

This is all well and good, but so far its very abstract and not very applicable in a practical setting, because we need at least one anchor point to control the correlation with other points. That is to say, we need at least one set of (x, f) values that we know for 'certain' (more about this shortly), so that we can then construct the f^* distributions for all the other infinite slices based upon this reference covariance. If we fix these points in certainty (or in actuality with an independent observation noise factor which will be introduced in the presence of observed data later), then we can form an improved conditional predictive distribution $p(\vec{f}^* | \vec{x}^*, \{\vec{f}\}, \{\vec{x}\})$ (where the vectors now represent multiple test tuples). This conditional predictive distribution is given, for one of the theoretically infinite number of test-points, by (see **Derivation 1** below for derivation):

$$\begin{aligned} p(\vec{f}^* | \vec{x}^*, \vec{f}, \vec{x}) &= N(\vec{f}^*; \vec{\mu}_{cp}, \Sigma_{cp}) \\ \vec{\mu}_{cp} &= K(\vec{x}, \vec{x}^*)^T K(\vec{x}, \vec{x})^{-1} \vec{f} \\ \Sigma_{cp} &= K(\vec{x}^*, \vec{x}^*) - K(\vec{x}, \vec{x}^*)^T K(\vec{x}, \vec{x})^{-1} K(\vec{x}, \vec{x}^*) \end{aligned} \tag{245}$$

where $K(\vec{x}, \vec{x})$ is a count($\vec{x}$) by count($\vec{x}$) matrix, $K(\vec{x}, x^*)$ is a count($\vec{x}$) by count($\vec{x}^*$) matrix, and $K(x^*, x^*)$ is a matrix also.

One thing which we have not done yet though is to include any *observations* - we assumed that our fixed points $\{(\vec{x}, \vec{f})\}$ have no error, which is equivalent to saying that we have observed them with absolute certainty - something which is of course non-practically true. To resolve this therefore, we need to use some Bayesian inference.

Noisy (Posterior) Predictive Distribution

More realistically, we have the situation that the observations are actually noisy, and follow an assumed Gaussian relationship:

$$y = f + \epsilon; \text{ for } \epsilon \sim \mathcal{N}(0, \sigma_n^2) \tag{246}$$

If we now, for simplicity (and physical accuracy if the noise is white noise) assume that the observation noise is i.i.d., we get the following covariance relationship:

$$\text{cov}(y_i, y_j) = K(x_i, x_j) + \sigma_n^2 \delta_{ij} = K(x_i, x_j) + \sigma_n^2 \hat{I} \tag{247}$$

So that when we switch out the noiseless observations $\vec{f}$ to the noisy ones $\vec{y}$, we now have (here $\vec{f}^*$ is in theory infinite dimensional):

$$\begin{bmatrix} \vec{y} \\ \vec{f}^* \end{bmatrix} = \mathcal{N}\left(0, \begin{bmatrix} K(\vec{x}, \vec{x}) + \sigma_n^2 \hat{I} & K(\vec{x}, \vec{x}^*) \\ K(\vec{x}, \vec{x}^*)^T & K(\vec{x}^*, \vec{x}^*) \end{bmatrix}\right) \tag{248}$$

and thus, simply we adjust Equation (245) to get:

$$
\begin{aligned}
p(\vec{f^*}|\vec{x}^*, \vec{y}, \vec{x}) &= N(\vec{f^*}; \vec{\mu}_{pp}, \Sigma_{pp}) \\
\vec{\mu}_{cp} &= K(\vec{x}, \vec{x}^*)^T \left[K(\vec{x}, \vec{x})^{-1} + \sigma_n^2 \hat{I} \right] \vec{f} \\
\Sigma_{cp} &= K(\vec{x}^*, \vec{x}^*) - K(\vec{x}, \vec{x}^*)^T \left[K(\vec{x}, \vec{x})^{-1} + \sigma_n^2 \hat{I} \right]^{-1} K(\vec{x}, \vec{x}^*)
\end{aligned}
\tag{249}
$$