# 1 Probability

- Given a joint distribution $\mathcal{P}(A, B)$, the **Product Rule**:

$$\mathcal{P}(A, B) = \mathcal{P}(A \wedge B) = \mathcal{P}(A|B)\mathcal{P}(B) \tag{1.1}$$

**Marginal distribution** or called the **sum rule**:

$$\mathcal{P}(A) = \sum_b \mathcal{P}(A, B) = \sum_b \mathcal{P}(A|B = b)\mathcal{P}(B = b) \tag{1.2}$$

**Conditional Probability**:

$$\mathcal{P}(A|B) = \frac{\mathcal{P}(A, B)}{\mathcal{P}(B)} \quad \text{if } \mathcal{P}(B) > 0 \tag{1.3}$$

**Bayes Rule//Bayes Theorem**:

$$\overbrace{\mathcal{P}(A = a|B = b)}^{\text{posterior}} = \frac{\mathcal{P}(A = a, B = b)}{\underbrace{\mathcal{P}(B = b)}_{\text{evidence}}} = \underbrace{\frac{\overbrace{\mathcal{P}(B = b|A = a)}^{\text{likelihood}}\overbrace{\mathcal{P}(A = a)}^{\text{prior}}}{\sum_{a'} \mathcal{P}(B = b|A = a')\mathcal{P}(A = a')}}_{\text{evidence / marginal likelihood}} \tag{1.4}$$

- **Monte Carlo approximation**: When you generate a new distribution by change of variables, monte carlo sampling can be used to approximate the new distribution. This would be approximated from the empirical distribution generated from the sampling (MLPP 2.7). $\sqrt{\frac{\hat{\sigma}^2}{S}}$ is called the empirical **standard error** and is an estimate of our uncertainty about our estimate of the mean from the empirical distribution. This shows the more the number of samples $S$, the lower the standard error (MLPP 2.7.3).
- **Entropy** of a random variable $X$ is the measure of uncertainty. Given a distribution with $K$ states, it measures the uncertainty given the distribution. Minimum value is 0 (no uncertainty), and maximum entropy is $\log_2 K$ (MLPP 2.8.1). Entropy is the average number of bits needed to encode a distribution.
- **Cross Entropy** is the average number of bits needed to encode data coming from a source with distribution $p$ and we encoded it with distribution $q$ rather than $p$ (MLPP 2.8.2).
- **Kullback-Leibler (KL) divergence** is the measure of the dissimilarity between two probability distributions. In other words its the average number of *extra* bits to encode data of distribution $p$ in distribution $q$ (MLPP 2.8.2). KL divergence is 0 iff $p \equiv q$.
- **Mutual Information** tells how much knowing one variable tells us about another variable, i.e. it is the reduction in uncertainty about $X$ after observing $Y$. This is done by seeing how similar the joint distribution $\mathcal{P}(X, Y)$ is to the factored distribution $\mathcal{P}(X)\mathcal{P}(Y)$. This ensures that mutual information is 0 iff the variables are independent (MLPP 2.8.3). This is unlike correlation coefficient which is 0 when variables are independent, but it can also be zero when variables are dependent.
- **Pointwise mutual information** is the amount we learn from updating the prior $\mathcal{P}(X)$ into the posterior $\mathcal{P}(X|Y)$. MI is the expected value of PMI (MLPP 2.8.3).
- Mutual information is usually defined for discrete distributions. For continuous variables you can discretize by histogramming them but that is easily perturbed by bin boundaries and bin sizes. A more robust way is to try different bin sizes and locations and find the configuration which maximizes MI. This statistic is called **Maximal Information Criterion (MIC)**. MIC can capture really non-linear and one-to-many dependence between variables which correlation coefficient can't (MLPP 2.8.3.1).

# 2 Generative Models for Discrete Data

- Psychological research has shown that people can **learn concepts** from positive examples alone (MLPP 3.2).
- The posterior prediction can be done by seeing on what set of **hypothesis spaces** of concepts, $\mathcal{H}$, does the data fit on. Hypothesis space is just a set of different hypothesis to fit a concept (MLPP 3.2). The subset of $\mathcal{H}$ which fits the data is called the **version space**.
- There might be multiple hypothesis in the version space, so the question is which hypothesis is the best one. You can apply the **size principle** which finds the simplest hypothesis to model the data (this is **Occam's razor**) (MLPP 3.2.1).
- There might be a un-natural/complicated hypothesis which produces a better likelihood to explain the data. In this case you can use a **Prior** to give lower probability to un-natural hypothesis. This gives you the ability to insert your subjective opinions or background knowledge for *Bayesian reasoning* (MLPP 3.2.2).
- Given that we want to find the best hypothesis $h$ which explains the data, we have the following bayesian reasoning:

$$\mathcal{P}(h|\mathcal{D}) = \frac{\mathcal{P}(h)\mathbb{I}(\mathcal{D} \in h)/|h|^N}{\sum_{h' \in \mathcal{H}} \mathcal{P}(\mathcal{D}, h')} \tag{2.1}$$

  When we have enough data, the posterior becomes peaked on a single concept, namely the **Maximum a Posteriori (MAP)** estimate:

$$\mathcal{P}(h|\mathcal{D}) = \delta_{\hat{h}^{\text{MAP}}}(h) \tag{2.2}$$

  where $\hat{h}^{\text{MAP}} = \arg\max_h \mathcal{P}(h|\mathcal{D})$ which is the posterior mode (MLPP 3.2.3).
- Since the likelihood term in equation 2.1 depends on $N$; increasing it would diminish the influence of the prior in a MAP estimate. Hence, a MAP estimate converges to the **Maximum Likelihood Estimate (MLE)**:

$$\hat{h}^{\text{MLE}} = \arg\max_h \log \mathcal{P}(\mathcal{D}|h) \tag{2.3}$$

  which says that the data overwhelms the prior (MLPP 3.2.3).
- **Sufficient Statistics** of a data are all the parameters needed to know the about the data to infer any parameters. For instance for 6 sided die rolled $N$ times, the sufficient statistics for inferring parameters for a Dirichlet-multinomial model would be $N_1, N_2, N_3, N_4, N_5$, and $N$ where $N_i$ is the number of times $i$ was rolled (MLPP 3.3.1).
- **Overfitting and the black swan paradox** happens when the sample size is small. When we just consider the MLE (ignore a prior - or have a uniform prior), the estimate might indicate for instance that a head is impossible because the 3 samples were all tails. This is called the **Zero count problem** or the **sparse data problem** (MLPP 3.3.4.1).
- When using a uniform prior (an uninformative prior) we always get the MLE solution.
- Naïve Bayes Classifier is a classifier which applies Bayes theorem with strong (naïve) independence assumptions (MLPP 3.5). It is a generative approach. Since we suppose independence, it requires us to specify the class conditional distribution ($\mathcal{P}(\mathbf{x}|y = c)$) for each feature separately. Hence, the class conditional density becomes:

$$\mathcal{P}(\mathbf{x}|y = c, \theta) = \prod_{j=1}^{D} \mathcal{P}(x_j|y = c, \theta_{jc}) \tag{2.4}$$

Since this model only requires $O(CD)$ parameters, where $C$ is the number of classes, and $D$ is the number of features, the classifier is relatively immune to overfitting. Hence, the posterior probability is just:

$$\mathcal{P}(y = c|\mathbf{x}, \theta) \propto \mathcal{P}(y = c) \prod_{j=1}^{D} \mathcal{P}(x_j|y = c, \theta_{jc}) \tag{2.5}$$

(wikipedia `http://en.wikipedia.org/wiki/Naive_Bayes_classifier`) "In simple terms, a naive Bayes classifier assumes that the presence or absence of a particular feature is unrelated to the presence or absence of any other feature, given the class variable. For example, a fruit may be considered to be an apple if it is red, round, and about 3" in diameter. A naive Bayes classifier considers each of these features to contribute independently to the probability that this fruit is an apple, regardless of the presence or absence of the other features."

# 3 Gaussian Models

- The **Multi-variate Gaussian/Normal (MVN)** is given by the exponent of the mahalanobis distance between point $\mathbf{x}$ and $\boldsymbol{\mu}$:

$$\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\mathsf{T}\boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})\right] \tag{3.1}$$

$$= \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left[-\frac{1}{2}\sum_{i=1}^{D}\frac{y_i^2}{\lambda_i}\right] \tag{3.2}$$

  where $y_i \equiv \mathbf{u}_i^\mathsf{T}(\mathbf{x} - \boldsymbol{\mu})$ where $\mathbf{u}_i$ is the $i$th column of the orthonormal matrix $\mathbf{U}$ from the eigen-decomposition $\boldsymbol{\Sigma} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^\mathsf{T}$, and $\lambda_i$ is the $i$th eigenvalue of $\boldsymbol{\Sigma}$. $\boldsymbol{\mu}$ decides the center of the ellipse, eigenvectors determines the orientation of the ellipse, and the eigenvalues determine the elongation in each direction (MLPP 4.1.2).

- The MLE for estimating the parameter of an MVN is just the empirical mean and empirical variance (MLPP 4.1.3).

- It can be proved that the MVN is the distribution with the maximum entropy having specified the mean and the variance (the first two moments). This is useful because usually given some data, the only thing that can be reliably estimated are the first two moments. Everything else must be left as uncertain as possible when creating the distribution - hence giving one reason for the wide use of MVN (MLPP 4.1.4).

- **Gaussian discriminant analysis (GDA)** is a generative classifier where class conditional densities are defined by a MVN i.e. $\mathcal{P}(\mathbf{x}|y = c, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c)$. When $\boldsymbol{\Sigma}_c$ is diagonal, it is a naïve Bayes classifier (MLPP 4.2). Types of GDA:

  - **Quadratic discriminant analysis (QDA)** is the simple GDA where the posterior is given by the MVN class conditional likelihood and the prior is just a scalar weighting of each class (MLPP 4.2.1).
  - **Linear discriminant analysis (LDA)** is an even simpler form of GDA where the class conditional covariance is the same across all classes i.e. $\boldsymbol{\Sigma}_c \equiv \boldsymbol{\Sigma}$. If you do the math, $\mathbf{x}^\mathsf{T}\boldsymbol{\Sigma}^{-1}\mathbf{x}$ can be factored out from the posterior since its independent of $c$. This leaves us with a **softmax function** (a function which at high values gives the most probable $\arg\max_c$ class, and at low values samples classes uniformly: $\mathcal{S}\left(\boldsymbol{\lambda} \triangleq [\lambda_1, \ldots, \lambda_N]\right)_c = \frac{e^{\lambda_c}}{\sum_{i=1}^{N} e^{\lambda_i}}$, If $\mathcal{S}(\boldsymbol{\lambda}/T)$ then with higher values of $T$, the probability would be more uniform). Essentially, we end up with linear decision boundaries between two classes (where the decision boundary is defined by where the maximum posterior between classes switches from one class to another) (MLPP 4.2.2).

- **Two-class LDA** is quite a simple model and $\mathcal{P}(y = 1|\mathbf{x}, \boldsymbol{\theta})$ boils down to a sigmoid function. Let's consider a point $\mathbf{x}_0$ which lies on the line between the two means $\boldsymbol{\mu}_0, \boldsymbol{\mu}_1$. If $\pi_0 = \pi_1$ i.e. the two priors are equal, then $\mathbf{x}_0 = \frac{1}{2}(\boldsymbol{\mu}_0 + \boldsymbol{\mu}_1)$. If $\pi_0 > \pi_1$ then $\mathbf{x}_0$ shifts more toward $\boldsymbol{\mu}_1$, making the line segment $\boldsymbol{\mu}_0, \mathbf{x}_0$ larger. Another parameter $\mathbf{w}$ defines the steepness of the logisitic function and depends on how separated the means relative to the variance: in other words $\mathbf{w} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)$ is the line in the direction $\boldsymbol{\mu}_0$ to $\boldsymbol{\mu}_1$. So the posterior is sigmoid $\text{sigm}(\mathbf{w}^\mathsf{T}(\mathbf{x} - \mathbf{x}_0))$ (MLPP 4.2.3).
- To make things even simpler, we can work with not only having the same covariance across all classes $\boldsymbol{\Sigma}_c \equiv \boldsymbol{\Sigma}$, but also enforcing diagonal covariance. This simple model is called the **Diagonal LDA** which is better suited for a high-dimensional setting (MLPP 4.2.7).
- One drawback of LDA is that it blindly considers all features regardless of their discriminability. **Nearest shrunken centroids classifier** (MLPP 4.2.8) is one such fix where we first find the class-independent mean for each feature $m_j$ and then see how much each class's mean for that feature deviates from it, $\Delta_{cj}$. If $\Delta_{cj} = 0$ for all $c$, that means the feature is uninformative. We can do this during MAP estimation by encouraging zeros by increasing the regularization parameter $\lambda$ for the prior - this essentially reduces the number of features our model would consider for posterior estimation.

- Even though using MLE for GDA sounds attractive due its simplicity it might pose problems because the full covariance matrix is singular if $N_c < D$ i.e. the number of examples for a particular class is less than the number of features. Even if $N_c > D$, $\boldsymbol{\Sigma}$ could be close to singular. Possible fixes at (MLPP 4.2.5).

- If $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2)$ is jointly gaussian with parameters:

$$\boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{pmatrix}, \quad \boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1} = \begin{pmatrix} \boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12} \\ \boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22} \end{pmatrix} \tag{3.3}$$

  The marginals are simply:

$$\mathcal{P}(\mathbf{x}_1) = \mathcal{N}(\mathbf{x}_1|\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_{11}) \tag{3.4}$$
$$\mathcal{P}(\mathbf{x}_2) = \mathcal{N}(\mathbf{x}_2|\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_{22}) \tag{3.5}$$

  and the conditionals are (MLPP 4.3.1):

$$\mathcal{P}(\mathbf{x}_1|\mathbf{x}_2) = \mathcal{N}(\mathbf{x}_1|\boldsymbol{\mu}_{1|2}, \boldsymbol{\Sigma}_{1|2}) \tag{3.6}$$
$$\boldsymbol{\mu}_{1|2} = \boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\mathbf{x}_2 - \boldsymbol{\mu}_2) \tag{3.7}$$
$$= \boldsymbol{\mu}_1 - \boldsymbol{\Lambda}_{11}^{-1}\boldsymbol{\Lambda}_{12}(\mathbf{x}_2 - \boldsymbol{\mu}_2) \tag{3.8}$$
$$\boldsymbol{\Sigma}_{1|2} = \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21} \tag{3.9}$$
$$= \boldsymbol{\Lambda}_{11}^{-1} \tag{3.10}$$

- You can write the MVN in the canonical/natural parameters rather than the moment parameters (mean and variance) (MLPP 4.3.3):

$$\boldsymbol{\Lambda} \equiv \boldsymbol{\Sigma}^{-1}, \quad \boldsymbol{\xi} \equiv \boldsymbol{\Sigma}^{-1}\boldsymbol{\mu} \tag{3.11}$$

  We can convert back to moment parameters by:

$$\boldsymbol{\mu} = \boldsymbol{\Lambda}^{-1}\boldsymbol{\xi}, \quad \boldsymbol{\Sigma} = \boldsymbol{\Lambda}^{-1} \tag{3.12}$$

  The distribution can now be written as:

$$\mathcal{N}_\text{N}(\mathbf{x}|\boldsymbol{\xi}, \boldsymbol{\Lambda}) = (2\pi)^{-D/2}|\boldsymbol{\Lambda}|^{1/2} \exp\left[-\frac{1}{2}\left(\boldsymbol{x}^\mathsf{T}\boldsymbol{\Lambda}\boldsymbol{x} + \boldsymbol{\xi}^\mathsf{T}\boldsymbol{\Lambda}^{-1}\boldsymbol{\xi} - 2\boldsymbol{x}^\mathsf{T}\boldsymbol{\xi}\right)\right] \tag{3.13}$$

Writing the conditional is easier in the canonical form:

$$\mathcal{P}(\boldsymbol{x}_1|\boldsymbol{x}_2) = \mathcal{N}_{\mathrm{N}}(\boldsymbol{x}_1|\boldsymbol{\xi}_1 - \boldsymbol{\Lambda}_{12}\boldsymbol{x}_2\boldsymbol{\Lambda}_{11}) \tag{3.14}$$

Multiplying two Gaussians:

$$\mathcal{N}_{\mathrm{N}}(\xi_f, \lambda_f)\mathcal{N}_{\mathrm{N}}(\xi_g, \lambda_g) = \mathcal{N}_{\mathrm{N}}(\xi_f + \xi_g, \lambda_f + \lambda_g) \tag{3.15}$$

- If $\mathbf{x} \in \mathbb{R}^{D_x}$ is a hidden variable and $\mathbf{y} \in \mathbb{R}^{D_y}$ is a noisy observation of $\mathbf{x}$. We can create a **Linear Gaussian System** to relate them (MLPP 4.4):

$$\mathcal{P}(\mathbf{x}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_x, \boldsymbol{\Sigma}_x) \tag{3.16}$$
$$\mathcal{P}(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}|\mathbf{A}\mathbf{x} + \mathbf{b}, \boldsymbol{\Sigma}_y) \tag{3.17}$$

The posterior is given by:

$$\mathcal{P}(\mathbf{x}|\mathbf{y}) = \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_{x|y}, \boldsymbol{\Sigma}_{x|y}) \tag{3.18}$$
$$\boldsymbol{\Sigma}_{x|y}^{-1} = \boldsymbol{\Sigma}_x^{-1} + \mathbf{A}^\mathsf{T}\boldsymbol{\Sigma}_y^{-1}\mathbf{A} \tag{3.19}$$
$$\boldsymbol{\mu}_{x|y} = \boldsymbol{\Sigma}_{x|y}\left[\mathbf{A}^\mathsf{T}\boldsymbol{\Sigma}_y^{-1}(\mathbf{y} - \mathbf{b}) + \boldsymbol{\Sigma}_x^{-1}\boldsymbol{\mu}_x\right] \tag{3.20}$$

- Inferring parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ for the MVN (MLPP 4.6).

# 4 Bayesian Statistics

- The use of priors and Bayes rule leads to Bayesian Statistics.
- We can easily compute a point estimate by looking at the posterior mean, median and mode (the MAP estimate). We most often use mode/MAP because it boils to an optimization problem, and furthermore MAP estimate can be seen in non-Bayesian terms by thinking of the log prior as a regularizer (MLPP Sect 5.2.1).
- **Drawbacks of MAP** (MLPP Sect 5.2.1): (1) gives no indication of uncertainty (like other point estimates); (2) having no idea of uncertainty leads us to over-confidence in our predictions (note we are most interested in predictive accuracy rather than parameter estimation for practical purposes); (3) the mode usually doesn't give a good idea of the distribution - because the largest peak can occur anywhere in an arbitrary distribution. Mean and median take the volume of the distribution into consideration. The way to work around this is to use a loss function. For instance a 0-1 loss function is $L(\boldsymbol{\theta}, \hat{\boldsymbol{\theta}}) = \mathbb{I}(\boldsymbol{\theta} \neq \hat{\boldsymbol{\theta}})$, which means that you get no partial credit for a wrong answer - hence this would return the optimal point estimate for the distribution. Loss functions:

| Loss method | Function | ... returns |
|---|---|---|
| 0-1 loss | $L(\boldsymbol{\theta}, \hat{\boldsymbol{\theta}}) = \mathbb{I}(\boldsymbol{\theta} \neq \hat{\boldsymbol{\theta}})$ | Optimal estimate for posterior mode |
| Squared error loss | $L(\boldsymbol{\theta}, \hat{\boldsymbol{\theta}}) = (\boldsymbol{\theta} - \hat{\boldsymbol{\theta}})^2$ | posterior mean |
| Robust loss function | $L(\boldsymbol{\theta}, \hat{\boldsymbol{\theta}}) = \lvert\boldsymbol{\theta} - \hat{\boldsymbol{\theta}}\rvert$ | posterior median |

; (4) MAP estimate is not invariant to reparameterization: changing from one representation to another equivalent representation doesn't transform the mode where it should be in the new representation. The MAP estimate depends on this parameterization. The MLE does not suffer from this since likelihood is a function not a probability density. Bayesian inference also does not suffer from this problem since the change of measure is taken into account while integrating over the parameter space.

- **Credible Intervals** (MLPP Sect 5.2.2): To get a measure of confidence, credible intervals gives you a way to give the width of the posterior distribution. This is a contiguous region which contains $1-\alpha$ of the posterior probability mass:

$$C_\alpha(\mathcal{D}) \;=\; (l,u) \;:\; \mathcal{P}(l \le \theta \le u|\mathcal{D}) \;=\; 1-\alpha \tag{4.1}$$

  where $l, u$ are the bounds. Because there might be many such intervals, we select the one such that there $(1-\alpha)/2$ mass in each tail; this is called the **central interval**. If we know the CDF $F$ of the posterior, $l = F^{-1}(\alpha/2)$ and $l = F^{-1}(1-\alpha/2)$. If we don't know $F$, we can draw samples from the posterior, and then use monte carlo approximation to the posterior quantiles: we draw $S$ samples and find the sample which occurs at $\alpha/S$ over the sorted list. We will get the true quantile when $S \to \infty$.

- **Highest posterior density regions** (MLPP Sect 5.2.2.1): rather than finding a region which has some amount of probability mass like in central interval, we might want to select all mass which is above a certain probability. We do this in a way to get $1-\alpha$ of the probability mass:

$$1-\alpha = \int_{\theta:\mathcal{P}(\theta,\mathcal{D})>p^*} \mathcal{P}(\theta,\mathcal{D})d\theta \tag{4.2}$$

  where $p^*$ is the probability threshold. Hence we get:

$$C_\alpha(\mathcal{D}) = \{\theta : \mathcal{P}(\theta,\mathcal{D}) > p^*\} \tag{4.3}$$

  Note that for a multimodal distribution this might be multiple connected regions in the posterior density.

- At times we have multiple parameters, and we are interested in computing the posterior distribution of some function of the parameters. For instance if there are two sellers on Amazon selling a thing for the same price, but seller 1 has 90 positive reviews, and 10 negative ones; and seller 2 has 2 positive reviews and 0 negative ones. We can write the problem if $\theta_1$ and $\theta_2$ are unknown reliabilities of the sellers, we want to find $\mathcal{P}(\theta_1 > \theta_2|\mathcal{D}) \equiv \mathcal{P}(\delta = \theta_1 - \theta_2 > 0|\mathcal{D})$. We can solve this analytically, or by monte carlo sampling which is easy because $\theta_1$ and $\theta_2$ are independent in the posterior.

- **Model Selection** (MLPP Sect 5.3): Another question in Bayesian statistics is selecting the right model which does not overfit or underfit the data - finding the right answer is model selection. One way to do this is to do cross-validation on each model $m$ and pick the model with the least error. A more efficient approach is to compute the posterior over the models:

$$\mathcal{P}(m|\mathcal{D}) = \frac{\mathcal{P}(\mathcal{D}|m)\mathcal{P}(m)}{\sum_{m\in\mathcal{M}} \mathcal{P}(m,\mathcal{D})} \tag{4.4}$$

  and then select the MAP model $\hat{m} = \arg\max \mathcal{P}(m|\mathcal{D})$, which is called the **Bayesian model selection**. If the priors are uniform, it amounts to finding the model $m$ which maximizes the likelihood $\mathcal{P}(\mathcal{D}|m) = \int \mathcal{P}(\mathcal{D}|\boldsymbol{\theta})\mathcal{P}(\boldsymbol{\theta}|m)d\boldsymbol{\theta}$.

  - **Bayesian Occam's Razor** (MLPP Sect 5.3.1): one might think that selecting the model based on $\mathcal{P}(\mathcal{D}|m)$ would select a complex model (using a lot of parameters). This would be true if we use $\mathcal{P}(\mathcal{D}|\hat{\boldsymbol{\theta}}_m)$ to select models where $\hat{\boldsymbol{\theta}}_m$ is the MLE or MAP estimate of parameters for model $m$, because models with more parameters would ofcourse fit the data better and give a higher likelihood. Howerever since $\mathcal{P}(\mathcal{D}|m)$ is integrating out the parameters rather than maximizing them we are automatically protected from overfitting - models with larger number of parameters not necessarily have a higher marginal likelihood. This is **Bayesian Occam's Razor** effect. Another way of seeing it is that $\sum_{\mathcal{D}'} \mathcal{P}(\mathcal{D}'|m) = 1$ so more complex models need to spread the probability mass more thinly over a larger area, whereas less complex models cover lesser data space so they might miss the data collection instance you are working with.

- In Parameter inference:

$$\mathcal{P}(\boldsymbol{\theta}|\mathcal{D}, m) = \frac{\mathcal{P}(\mathcal{D}|\boldsymbol{\theta}, m)\mathcal{P}(\boldsymbol{\theta}|m)}{\mathcal{P}(\mathcal{D}|m)} \tag{4.5}$$

  we didn't have to worry about the evidence/marginal likelihood $\mathcal{P}(\mathcal{D}|m)$ because that is constant w.r.t $\boldsymbol{\theta}$. However when comparing models $m$, we need to compute the evidence (MLPP Sect 5.3.2). This can be found if we know the normalization factors individually $Z_0, Z_\ell, Z_N$ i.e. the prior, likelihood, and posterior respectively - then $\mathcal{P}(\mathcal{D}) = Z_N/(Z_0 Z_\ell)$.

- The BIC approximation to log marginal likelihood takes the form where a penalty is applied according to the complexity of the model (MLPP Sect 5.3.2.4). BIC is very closely related to **minimum descriptor length** (MDL) which characterizes the score for a model in terms of how well it fits the data minus how complex the model is. Also similar is the **Akaike information criterion** (AIC) which is derived from the frequentist framework, and usually gives more complex models than BIC (because the penalize complexity of the model with less intensity) - but it usually results in better predicitive accuracy.
- The effect of a prior might not be that significant for posterior inference (because with larger data the likelihood overwhelms the prior), but in the case of computing the evidence/marginal likelihood, it plays a much more significant role, since we are averaging the likelihood over all possible parameter configurations, as weighted by the prior (MLPP 5.3.2.5). If the prior is of the form $\mathcal{N}(\mathbf{0}, \alpha^{-1}\mathbf{I})$, so if $\alpha$ is large, the effect of the prior is small hence we can use a complex model with many small parameters. When the prior is unknown, the right Bayesian thing to do is to have a prior over a prior. It is alright to give an uninformative hyper-prior - yet it is always better to go up in this prior to prior hierarchy.
- Suppose that we have choice of two models $M_0, M_1$. What model should we select. For this we define the **Bayes Factor**, which is just the likelihood ratio except that we integrate out the parameters which allows us to compare models of different complexity:

$$BF_{1,0} = \frac{\mathcal{P}(\mathcal{D}, M_1)\mathcal{P}(M_0)}{\mathcal{P}(\mathcal{D}, M_0)\mathcal{P}(M_1)} \tag{4.6}$$

  If $BF_{1,0} > 1$, then we prefer model 1. There is Jeffrey's scale on $BF_{1,0}$ which is the bayesian counterpart to the frequentist p-value (MLPP 5.3.3).

- **Improper Prior** are ones which don't sum to 1, which can be problematic for model selection/hypothesis testing (MLPP 5.3.4). Plus, using proper, but very vague, priors can cause similar problems. In particular Bayes factor will always favor a simpler model, since the probability of the observed data under a complex model with a very diffuse prior will always be very small. This is called the **Jeffreys-Lindley paradox**.

- The most controversial aspect of Bayesian statistics is the reliance on priors. Bayesians argue this is unavoidable since all inference must be done conditional on certain assumptions about the world. But, you can minimize the reliance on priors (MLPP sect 5.4):
  - **Uninformative Priors**: which is "let the data speak for itself."
  - **Jeffreys Priors**: are more principled way of producing uninformative priors. The key observation is that if $\mathcal{P}(\phi)$ is non-informative then any re-parameterization of the prior such as $\theta = \mathcal{P}(\phi)$ for some function $h$, should also be non-informative (MLPP sect 5.4.2). It is set as proportional to the square root of the determinant of the Fisher information: $\mathcal{P}_\theta(\theta) \propto \sqrt{|\mathcal{I}(\phi)|}$.
  - **Robust Priors** (MLPP sect 5.4.3): Try not to be too confident on the prior by having heavy tails which avoids forcing things toward the prior mean.
  - Since robust priors can be expensive to compute, a good compromise is to use **mixtures of conjugate priors** (MLPP sect 5.4.4). They can be used to approximate many kinds of priors.

- **Hierarchical Bayes** if we want to compute the posterior $\mathcal{P}(\boldsymbol{\theta}|\mathcal{D})$ we need to know the prior $\mathcal{P}(\boldsymbol{\theta}|\eta)$. But if we don't know how to set the hyperparameters $\eta$, we need to have prior over priors. This is called hierarchical bayes since there are multiple levels of unknown quantities (MLPP sect 5.5). For instance in a two level model:

$$\mathcal{P}(\boldsymbol{\eta}, \boldsymbol{\theta}|\mathcal{D}) \propto \mathcal{P}(\mathcal{D}|\boldsymbol{\theta})\mathcal{P}(\boldsymbol{\theta}|\boldsymbol{\eta})\mathcal{P}(\boldsymbol{\eta}) \tag{4.7}$$

- **Empirical Bayes** (MLPP sect 5.6): sometimes its better to marginalize out $\boldsymbol{\theta}$ from Equation 4.7; which would leave us with a simpler model of computing $\mathcal{P}(\boldsymbol{\eta}|\mathcal{D})$. Rather than estimating the full distribution we can just get a point estimate for $\boldsymbol{\eta}$, which is typically smaller in dimensionality than $\boldsymbol{\theta}$, and hence is less prone to overfitting, and we can safely use a uniform prior. Of course, empirical bayes violates the principle that the prior should be chosen independent of the data. But one can view this as a cheap approximation to inference in a hierarchical Bayesian model, just as MAP estimation is an approximation to inference on one level model $\boldsymbol{\theta} \rightarrow \mathcal{D}$. In short, the more integrals one performs, the more Bayesian you become:

| **Method** | **Definition** |
|---|---|
| Maximum Likelihood | $\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \mathcal{P}(\mathcal{D}|\boldsymbol{\theta})$ |
| MAP Estimation | $\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \mathcal{P}(\mathcal{D}|\boldsymbol{\theta})\mathcal{P}(\boldsymbol{\theta}|\boldsymbol{\eta})$ |
| Empirical Bayes (ML-II) | $\hat{\boldsymbol{\eta}} = \arg\max_{\boldsymbol{\eta}} \int \mathcal{P}(\mathcal{D}|\boldsymbol{\theta})\mathcal{P}(\boldsymbol{\theta}|\boldsymbol{\eta})d\boldsymbol{\theta} = \arg\max_{\boldsymbol{\eta}} \mathcal{P}(\mathcal{D}|\boldsymbol{\eta})$ |
| MAP-II | $\hat{\boldsymbol{\eta}} = \arg\max_{\boldsymbol{\eta}} \int \mathcal{P}(\mathcal{D}|\boldsymbol{\theta})\mathcal{P}(\boldsymbol{\theta}|\boldsymbol{\eta})\mathcal{P}(\boldsymbol{\eta})d\boldsymbol{\theta} = \arg\max_{\boldsymbol{\eta}} \mathcal{P}(\mathcal{D}|\boldsymbol{\eta})\mathcal{P}(\boldsymbol{\eta})$ |
| Hierarchical Bayes (Full Bayes) | $\mathcal{P}(\boldsymbol{\eta}, \boldsymbol{\theta}|\mathcal{D}) \propto \mathcal{P}(\mathcal{D}|\boldsymbol{\theta})\mathcal{P}(\boldsymbol{\theta}|\boldsymbol{\eta})\mathcal{P}(\boldsymbol{\eta})$ |

- **Bayesian Decision Theory** (MLPP sect 5.7): Nature produces a set of labels $y \in \mathcal{Y}$ and an observation with each label $\mathbf{x} \in \mathcal{X}$, and we need to take an action $a$ from the action space $\mathcal{A}$. We incur a **loss** $L(y, a)$, which measures how compatible our action $a$ was to the hidden state $y$. We gave some loss functions in this table 4. Our goal would be to devise a **decision procedure** or **policy**, $\delta : \mathcal{X} \rightarrow \mathcal{A}$ in a way that it minimizes the expected loss:

$$\delta(\mathbf{x}) = \arg\min_{a \in \mathcal{A}} \mathbb{E}[L(y, a)] \tag{4.8}$$

Following this leads to **rational behavior**. By saying "expected" in the Bayesian sense we mean the expected value of $y$ given the data we have seen so far - this would be the **posterior expected loss**:

$$\rho(a|\mathbf{x}) \triangleq \mathbb{E}_{\mathcal{P}(y|\mathbf{x})}[L(y, a)] = \int_y L(y, a)\mathcal{P}(y|\mathbf{x})dy \tag{4.9}$$

where the **Bayes Estimator/Bayes Decision Rule** is given by:

$$\delta(\mathbf{x}) = \arg\min_{a \in \mathcal{A}} \rho(a|\mathbf{x}) \tag{4.10}$$

For instance, for the 0-1 loss function (MLPP sect 5.7.1.1) - where $y$ is the true class label and one is penalized when $y \neq a$, the posterior expected loss is $\rho(a|\mathbf{x}) = \mathcal{P}(a \neq q|\mathbf{x}) = 1 - \mathcal{P}(y|\mathbf{x})$, and hence the Bayes estimator becomes the posterior mode: $y^*(\mathbf{x}) = \arg\max_{y \in \mathcal{Y}} \mathcal{P}(y|\mathbf{x})$.

- When the classifier is really unsure about its decision, it can refuse to classify that data point. This is a **Reject Option** (MLPP sect 5.7.1.2). The loss function with reject action can be written as:

$$L(y = j, a = i) = \begin{cases} 0 & \text{if } i = j \text{ and } i, j \in \{1, \ldots, C\} \\ \lambda_r & \text{if } i = C + 1 \text{ i.e. refuse to classify} \\ \lambda_s & \text{classified incorrectly} \end{cases} \tag{4.11}$$

We can show that the optimal action is to pick the reject action if the most probably class has a probability below $1 - \frac{\lambda_r}{\lambda_s}$, otherwise pick the most probable class.

- Rather than having a 0-1 loss, we can have a different loss on false negative and false positives:

| | $\hat{y}=1$ | $\hat{y}=0$ |
|---|---|---|
| $y=1$ | 0 | $L_{FN}$ |
| $y=0$ | $L_{FP}$ | 0 |

The posterior expected loss for the two possible actions is given by:

$$\rho(\hat{y}=0|\mathbf{x}) = L_{FN} \cdot \mathcal{P}(y=1|\mathbf{x}) \tag{4.12}$$
$$\rho(\hat{y}=1|\mathbf{x}) = L_{FP} \cdot \mathcal{P}(y=0|\mathbf{x}) \tag{4.13}$$

Ofcourse, we should pick $\hat{y}=1$ iff $\rho(\hat{y}=0|\mathbf{x}) > \rho(\hat{y}=1|\mathbf{x})$ i.e. $\mathcal{P}(y=1|\mathbf{x})/\mathcal{P}(y=0|\mathbf{x}) > L_{FP}/L_{FN}$.

- These quantities are important for ROC and PR curves:

| | | |
|---|---|---|
| True Positive Rate / Sensitivity / Recall | $TPR = TP/(TP+FN) \approx \mathcal{P}(\hat{y}=1\|y=1)$ | Ratio of correctly classified $y=1$ |
| False Positive Rate / False Alarm Rate | $FPR = FP/(FP+TN) \approx \mathcal{P}(\hat{y}=1\|y=0)$ | Ratio of incorrectly classified $y=0$ |
| Precision | $P = TP/(TP+FP)$ | Ratio of correctly classified $\hat{y}=1$ decisions |

**ROC** is plot of TPR against FPR. Given a classifier $\delta(\mathbf{x}) = \mathbb{I}(f(\mathbf{x}) > \tau)$ i.e. the classifier will classify positive $\hat{y}=1$ if $f(\mathbf{x}) > \tau$. If we start from threshold $\tau$ of 1, there are no TP and FP (because the classifier never gives $\hat{y}=1$). Hence both $TPR = FPR = 0$. When we decrease the threshold, you can increase TPR and FPR, because TP and FP increase by decreasing $\tau$. Hence, the TPR and FPR functions parameterized by $1-\tau$ are both monotonically increasing functions. Since $dTPR/dFPR > 0$, the ROC curve is always monotonically increasing. The **area under curve (AUC)** gives a single error metric from ROC curves. The other metric is **equal error rate (EER)** which is the value where $FPR = 1 - TPR$, in other words, it is the value on the line going from (0,1) to (1,0). Lower the EER the better.

- **PR curve** is precision against recall. Precision measure what fraction of $\hat{y}=1$ were correct classifications, whereas recall measures what fraction of $y=1$ were correctly classified. In other words, PR curves don't care about TN's. For a single score we can use the mean precision (averaging over all recall values). Alternatively we can give the precision for fixed recall value. The **F score / F1 score** is the harmonic mean of precision recall:

$$F_1 \triangleq \frac{2PR}{R+P} \tag{4.14}$$

As stated before recall/TPR is 0 when $\tau = 1$. On the other hand precision is usually undefined at $\tau = 1$, because there are usually no TP or FP. When decreasing $\tau$, the curve is defined as soon as we make the first classification $\hat{y}=1$. Moreover, when $\tau = 0$, the recall is usually 1 because FN=0. On the other hand the precision at $\tau = 0$ is usually non-zero because it is $\approx \mathcal{P}(y=1)/(\mathcal{P}(y=1) + \mathcal{P}(y=0))$, whenever $FN = TN = 0$ i.e. no $\hat{y}=0$ classification is made. One important thing to note is that PR curve is not a necessarily a monotonically decreasing function with decreasing $\tau$. The reason being is that precision can decrease or increase with increasing recall (decreasing $\tau$). This happens whenever you add more TP without adding any FP (more accurately $\alpha/A > \beta/B$ where $\alpha, \beta$ are the set of new/added TP and FP, whereas $A, B$ are the number of TP and FP respectively at higher $\tau$).

# 5 Frequentist Statistics

- By trying to deal with parameters not like random variables (which Bayesians do) we come up with frequentist statistics. Instead of being based on posterior distribution, they are based on sampling distribution. This distribution is built by sampling multiple distributions from the true latent distribution. In Bayesian approach there are no repeated trials.

- (MLPP sect 6.2) A parameter is estimated by an estimator $\delta$ to some data $\mathcal{D}$, so $\hat{\boldsymbol{\theta}} = \delta(\mathcal{D})$. The parameter is viewed as fixed whereas the data is random, which is exact opposite of the Bayesian view. The uncertainty in the parameter estimate can be measured by computing the sampling distribution of $\delta$. The way it is done is that lets say you have different datasets $\mathcal{D}^{(s)} = \{\mathbf{x}_i^{(s)}\}_{i=1}^N$ for some true model $\mathcal{P}(\cdot|\boldsymbol{\theta}^*)$, where $N$ is the number of samples in each dataset for $s = [1, S]$. If the estimate $\hat{\delta}$ gives some estimate of $\theta$, then the distribution of our estimate would be $\{\hat{\theta}(\mathcal{D}^{(s)})\}$. As we let $S \to \infty$, the distribution induced is the sampling distribution of the estimator. Bootstrap sampling is one way to get this sampling distribution:
- **Bootstrap** (MLPP sect 6.2.1) is a simple Monte Carlo technique to approximate the sampling distribution. For **parametric bootstrap** is to generate the samples using $\hat{\theta}(\mathcal{D})$. An alternative, is the **non-parametric bootstrap**, is to sample the $x_i^s$ (with replacement) from the original data $\mathcal{D}$, and then compute the induced distribution as before.
- In **frequentist decision theory** (MLPP sect 6.3) there is a loss function and likelihood, but no posterior or posterior expected loss. We are free to choose any estimator or decision procedure. Our expected loss or risk is:

$$R(\theta^*, \delta) \triangleq E_{\mathcal{P}(\tilde{\mathcal{D}}|\theta^*)}\left[L(\theta^*, \delta(\tilde{\mathcal{D}}))\right] = \int L(\theta, \delta(\tilde{\mathcal{D}}))\mathcal{P}(\tilde{\mathcal{D}}|\theta^*)d\tilde{\mathcal{D}} \tag{5.1}$$

  A basic problem with frequentist decision theory is that it relies on knowing the true distribution $\mathcal{P}(\cdot|\theta^*)$ to evaluate risk. As compared to the Bayesian approach in Equation 4.9, which averages over $\theta$ (which is unknown) and conditions on $\mathcal{D}$ (which is known), whereas the frequentist approach averages over $\tilde{\mathcal{D}}$ (thus ignoring the observed data), and conditions on $\theta$ (which is still unknown). Note that the risk cannot even be computed because $\theta^*$ is unknown - and consequently we cannot compare different estimators in terms of their frequentist risk. Following are some solutions:

  - We need to convert $R(\theta^*, \delta)$ into a single measure of quality $R(\delta)$, hence dropping the reliance on $\theta^*$. One approach is to put a prior on $\theta^*$ and then define **Bayes risk** of an estimator as:

$$R_B(\delta) \triangleq E_{\mathcal{P}(\theta^*)}[R(\theta^*, \delta)] = \int R(\theta^*, \delta)\mathcal{P}(\theta^*)d\theta^* \tag{5.2}$$

    and now like in the Bayesian case, we get the Bayes estimator which minimizes the expected risk: $\delta_B \triangleq \arg\min_\delta R_B(\delta)$. *A Bayes estimator can be obtained by minimizing the posterior expected loss for each* $\mathbf{x}$. (see MLPP Theorem 6.3.1). This shows that by picking the optimal action on a case-by-case basis (as in the Bayesian approach) is optimal on average (as in the frequentist approach).
  - (MLPP sect 6.3.2) Rather than having to define a prior for the estimator, an alternative approach is to define **maximum risk** of an estimator: $R_{\max}(\delta) \triangleq \max_{\boldsymbol{\theta}^*} R(\boldsymbol{\theta}^*, \delta)$. **minmax rule** is one which minimizes the maximum risk:

$$\delta_{MM} \triangleq \arg\min_\delta R_{\max}(\delta) \tag{5.3}$$

    i.e. which decision procedure has the lowest worst case risk. Minmax estimators are overly conservative.
  - Even though frequentist decision theory relies on knowing the true distribution $\mathcal{P}(\cdot|\theta^*)$ in order to evaluate risk, some estimators might be worse than others regardless of the value of $\theta^*$ i.e. if $R(\theta, \delta_1) \leq R(\theta, \delta_2) : \forall\theta \in \Theta$, then we can say $\delta_1$ dominates $\delta_2$. An estimator is **admissible** if it is not *strictly* dominated by any other estimator (MLPP sect 6.3.3).
  - The **James-Stein estimator** is a biased shrinkage estimator of the mean of Gaussian random vectors: $\hat{\theta}_i = B\bar{x} + (1 - B)x_i$ where $0 < B < 1$ is a tuning constant. It can be shown that the shrinkage estimator has lower frequentist risk than the MLE (sample mean) for $N \geq 4$. This is called the **Stein's paradox** (MLPP sect 6.3.3.2).

- An estimator is considered **consistent** if it can eventually recover the parameters from the data i.e. $s \to \infty : \hat{\theta}(\mathcal{D}) \to \theta^*$.
- The **bias of an estimator** is:

$$\text{bias}(\hat{\theta}(\cdot)) = \mathbb{E}_{\mathcal{P}(\mathcal{D}|\theta^*)}\left[\hat{\theta}(\mathcal{D}) - \theta^*\right] \tag{5.4}$$

*The MLE for a Gaussian variance, $\hat{\sigma}^2$ is not an unbiased estimator of $\sigma^2$.* Let's say you are trying to compute the variance $\sigma^2$ from the whole population i.e our $\theta^* : \sigma^2 = \frac{\sum_{i=1}^{N}(x_i-\mu)^2}{N}$. When you sample and try to estimate the statistic of the variance $s^2 = \frac{\sum_{i=1}^{n}(x_i-\bar{x})^2}{n}$, we can provably get a smaller estimate of the variance which is biased. To get an unbiased estimate you need the denominator to be $n-1$ (to make the statistic larger).

- **Cramer-Rao lower bound** provides a lower bound on the variance of any unbiased estimator (MLPP sect 6.4.3). This is important to see because just seeing that the estimator is unbiased is not enough - we also need to see its variance.
- **Bias-variance tradeoff** (MLPP sect 6.4.4) We can prove that the expected value of the variance of the estimated parameter is given by (supposed we use quadratic loss):

$$\mathbb{E}\left[(\hat{\theta} - \theta^*)^2\right] = \text{var}\left[\hat{\theta}\right] + \text{bias}^2(\hat{\theta}) \tag{5.5}$$

$$\text{MSE} = \text{variance} + \text{bias}^2 \tag{5.6}$$

($\hat{\theta}$ is the estimate, $\theta^*$ is the true value) This means that it might be useful to use an biased estimator as long as it reduces our variance, considering our goal is to minimize squared loss. (see MLPP Figure 6.5)

- (MLPP sect 6.4.4.3) It can be shown for 0-1 loss instead of squared error that risk is no longer experssible as squared bias plus variance. So for **classification, the bias variance tradeoff** works a such: if you are on the right side of the decision boundary then the bias is negative and decreasing the variance will decrease the misclassification rate. But, if your estimate is on the wrong side of the decision boundary, then the bias is positive, and it pays to have a larger variance.
- **Empirical Risk Minimization** (MLPP sect 6.5): To overcome the problem of not being able to compute the risk function (since we don't know the true distribution), we can look at a loss function of the form $L(y, \delta(\mathbf{x}))$, where $y$ is true but unknown response and $\delta(\mathbf{x})$ is our prediction given the input $\mathbf{x}$. In this case the frequentist risk is:

$$R(p_*, \delta) \triangleq \mathbb{E}_{(\mathbf{x},y)\sim p_*}\left[L(y, \delta(\mathbf{x}))\right] = \sum_{\mathbf{x}}\sum_{y} L(y, \delta(\mathbf{x}))p_*(\mathbf{x}, y) \tag{5.7}$$

where $p_*$ is the nature's distribution, which can be empirically estimated $p_{\text{emp}}(\mathbf{x}, y|\mathcal{D}) \triangleq \frac{1}{N}\sum_{i=1}^{N}\delta_{\mathbf{x}_i}(\mathbf{x})\delta_{y_i}(y)$, which makes the empirical risk as follows:

$$R_{\text{emp}}(\mathcal{D}, \delta) \triangleq R(p_{\text{emp}}(\cdot|\mathcal{D}), \delta) = \frac{1}{N}\sum_{i=1}^{N} L(y_i, \delta(\mathbf{x}_i)) \tag{5.8}$$

In the case of 0-1 loss $L(y, \delta(\mathbf{x})) = \mathbb{I}(y \neq \delta(\mathbf{x}))$, this becomes the **misclassification rate**. In the case of squared error loss, $L(y, \delta(\mathbf{x})) = (y - \delta(\mathbf{x}))^2$ this becomes **mean squared error**.

- Usually minimizing the empirical risk will result in overfitting. It is therefore necessary to add complexity penalty to the objective function (in comparison to Equation 5.8):

$$R'(\mathcal{D}, \delta) = R_{\text{emp}}(\mathcal{D}, \delta) + \lambda C(\delta) \tag{5.9}$$

where $C(\delta)$ is a measure of complexity of the prediction function and $\lambda$ controls the strength of the complexity penalty. This is called **Regularized risk minimization** (MLPP 6.5.1). We can now pick an estimator by $\hat{\delta}_\lambda = \arg\min_\delta [R'(\mathcal{D}, \delta)]$, where following the **structural risk minimization** principle $\hat{\lambda} = \arg\min_\lambda \hat{R}(\hat{\delta}_\lambda)$, where $\hat{R}(\delta)$ is an estimate of risk.

- We can estimate the risk of an estimator by using a validation set (or cross validation - use **stratified CV** to approximately equal the proportion of labels in each fold). The risk when doing CV is this:

$$R(m, \mathcal{D}, K) = \frac{1}{N} \sum_{k=1}^{K} \sum i \in \mathcal{D}_k L(y_i, f_m^k(\mathbf{x}_i)) \tag{5.10}$$

  where $m$ is a discrete index such as the degree of a polynomial, or a continuous index such as the strength of a regularizer. $K$ is the number of folds, $\mathcal{D}_k$ is the set of points in fold $k$, and $f_m^k(\mathbf{x})$ is the function that was trained on all the data except the test data for fold $k$.
- **Leave one out cross validation (LOOCV)** is one where the model is trained and tested $N$ times by leaving a sample out each time. When we analytically remove the effect of the $i$th training sample, it is called **generalized cross validation**.
- The standard frequentist measure of uncertainty of an estimate is the standard error of the mean:

$$se = \frac{\hat{\sigma}}{\sqrt{N}} = \sqrt{\frac{\hat{\sigma}^2}{N}}, \qquad \hat{\sigma}^2 = \frac{1}{N} \sum_{i=1}^{N} (L_i - \bar{L})^2, \quad L_i = L(y_i, f_m^{k(i)}(\mathbf{x}_i)), \quad \bar{L} = \frac{1}{N} \sum_{i=1}^{N} L_i \tag{5.11}$$

  Note that $\sigma$ measures the intrinsic variability of $L_i$ across samples, whereas $se$ measures our uncertainty about the mean $\bar{L}$ (MLPP sect 6.5.3.2).
- A common heuristic for picking a model from these noisy estimates is to pick the value which corresponds to the simplest model whose risk is no more than one standard error above the risk of the best model; this is called the **one-standard error rule**.
- (MLPP sect 6.5.4) Since CV is slow, there is motivation to find analytical approximations or bounds to the generalization error. **Statistical Learning Theory (SLT)** tries to bound the risk $R(p_*, h)$ for any data distribution $p_*$ and hypothesis $h \in \mathcal{H}$ in terms of the empirical risk $R_{emp}(\mathcal{D}, h)$, the sample size $N = |\mathcal{D}|$, and the size of the hypothesis space $\mathcal{H}$. The bound you can derive tells us that the optimism of the training error (i.e. risk always seems low when the model is viewed from the training set) increases $\dim(\mathcal{H})$ but decreases with $N = |\mathcal{D}|$, as is expected. If the hypothesis space $\mathcal{H}$ is infinite (e.g. we have real valued parameters) we cannot use $\dim(\mathcal{H}) = |\mathcal{H}|$, and instead we need to use **Vapnik-Chervonenkis** or **VC dimension** of the hypothesis class.
- **Log loss**: $L_{\text{nll}}(y, \eta) = -\log \mathcal{P}(y|\mathbf{x}, \mathbf{w}) = \log(1 + e^{-y\eta})$.

- In Bayesian statistics, we condition on what is known namely the observed data, $\mathcal{D}$ and average over what is not known, namely the parameter $\theta$. In frequentist statistics, we do exactly the opposite: we condition on what is unknown namely the true parameter value $\theta$ and average over hypothetical future data sets $\tilde{\mathcal{D}}$.
- p-value (MLPP sect 6.6.2)
- (MLPP sect 6.6.3) The fundamental reason for many of frequentist pathologies is that frequentist inference violates the likelihood principle, which says that inference should be based on the likelihood of the observed data, not based on hypothetical future data that you have not observed. The main arguments for it are the **sufficiency principle** (all data to estimate an unknown parameter is in the sufficient statistic), and **weak conditionality** (which says inference should be based on information that has happened not on what might happen).

# 6 Linear Regression

- It is used to fit models in supervised machine learning. When augmented with kernels or other basis functions, it can even model non-linear relationships.

- (MLPP sect 7.2) Linear regression model is of this form:

$$\mathcal{P}(y|\mathbf{x},\boldsymbol{\theta}) = \mathcal{N}(y|\mathbf{w}^\mathsf{T}\mathbf{x},\sigma^2) \tag{6.1}$$

For non-linear relationships, we can do **basis function expansion** by putting a non-linear function over $\mathbf{x}$, $\phi(\mathbf{x})$:

$$\mathcal{P}(y|\mathbf{x},\boldsymbol{\theta}) = \mathcal{N}(y|\mathbf{w}^\mathsf{T}\phi(\mathbf{x}),\sigma^2) \tag{6.2}$$

Note that even now the model is linear in the parameters $\mathbf{w}$, hence it is still linear regression. If $\mathbf{x}$ was just a scalar, you could use a polynomial basis function $\phi(x) = [1, x, x^2, \ldots, x^d]$. Increasing $d$ would increase the complexity of the model. The expectation of the linear model can be written as:

$$\mathbb{E}(y|\mathbf{x}) = w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n = \mathbf{w}^\mathsf{T}\mathbf{x} \tag{6.3}$$

where the first value of $\mathbf{x}$ vector is always 1, and $w_0$ is called the intercept. You can make this model arbitrarily complex as long as it is linear in the parameter space, for instance:

$$\mathbb{E}(y|\mathbf{x}) = w_0 + w_1x_1 + w_2x_2 + w_3x_1x_2 + w_4x_1^2 + w_5x_2^2 \tag{6.4}$$

- (MLPP sect 7.3) You can compute the **maximum likelihood estimate** or what is called the **least squares solution**: $\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \log \mathcal{P}(\mathcal{D}|\boldsymbol{\theta})$. If the training examples are **independent and identically distributed (iid)**, then we can write the log likelihood as $\ell(\boldsymbol{\theta}) \triangleq \mathcal{P}(\mathcal{D}|\boldsymbol{\theta}) = \sum_{i=1}^N \log \mathcal{P}(y^{(i)}|\mathbf{x}^{(i)},\boldsymbol{\theta})$. Instead of maximizing the log likelihood, we can minimize the negative log likelihood (NLL). It turns out the MLE minimizes the **residual sum of squares / sum of square errors**:

$$\text{RSS}(\mathbf{w}) \triangleq \sum_{i=1}^N (y^{(i)} - \mathbf{w}^\mathsf{T}\mathbf{x}^{(i)})^2 = \|\epsilon\|_2^2 \tag{6.5}$$

(RSS/$N$ is called the **mean squared error (MSE)**). Note the negative log likelihood is convex with a unique minimum. Importantly, this is true even if we use basis function expansion, such as polynomials, because the negative log likelihood is still linear in the parameters $\mathbf{w}$, even if it is not linear in the inputs $\mathbf{x}$. The derivation of the MLE leads to the **normal equation** - where the solution $\hat{\mathbf{w}}$ to this linear system of equations is called the **ordinary least squares (OLS)**:

$$\hat{w}_{\text{OLS}} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y} \tag{6.6}$$

where the design matrix $\mathbf{X}$ which contains all the training data, where each row is a training sample, and each column is a feature. Note, the first column in $\mathbf{X}$ will be all 1s since $x_0^{(i)} = 1$. We will also have a vector $\mathbf{y}$ containing the output values for each training sample.

- For a geometrical interpretation of linear regression see MLPP sect 7.3.2.
- (MLPP sect 7.3.3) For a scalar parameter function a **convex function** is one where $\frac{d^2}{d\theta^2}f(\theta) > 0$. For multivariate linear regression, a twice continuously differentiable function is convex iff its Hessian is positive definite for all $\boldsymbol{\theta}$ (where Hessian is the matrix of second partial derivatives, defined by $H_{jk} = \frac{\partial f^2(\boldsymbol{\theta})}{\partial\theta_j\partial\theta_k}$)
- If there are outliers in the data, linear regression with $\ell_2$ squared loss will be severely affected. This is because squared loss penalizes deviations quadratically, so points far away from the fit have more effect than the ones near. To do **robust linear regression** (MLPP sect 7.3.2) you can replace the Gaussian distribution for the response variable with a distribution with heavy tails - such a distribution will assign higher likelihood to outliers without having to perturb the model. One possibility is laplace distribution: $\mathcal{P}(y|\mathbf{x},\mathbf{w},b) \propto \exp\left(-\frac{1}{b}|y - \mathbf{w}^\mathsf{T}\mathbf{x}|\right)$. Another is a **huber loss** function which is quadratic before the loss is $\delta$, and become like $\ell_1$ after that:

$$L_H(r,\delta) = \begin{cases} r/2 & \text{if } |r| \le \delta \\ \delta|r| - \delta^2/2 & \text{if } |r| > \delta \end{cases} \tag{6.7}$$

- **Ridge Regression** (MLPP sect 7.5): To avoid overfitting, we can use a gaussian prior with a gaussian likelihood (you can use a robust function for the likelihood as discussed before).

$$\arg\max_{\mathbf{w}} \sum_{i=1}^{N} \log \mathcal{N}(y^{(i)}|w_0 + \mathbf{w}^\mathsf{T}\mathbf{x}^{(i)}, \sigma^2) + \sum_{j=1}^{D} \log \mathcal{N}(w_j|0, \tau^2) \tag{6.8}$$

the second part is the zero mean Gaussian prior where $1/\tau^2$ controls the strength. Note, now we have separated out the intercept because we don't want to regularize it. This is equivalent to minimizing the cost function:

$$J(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - (w_0 + \mathbf{w}^\mathsf{T}\mathbf{x}^{(i)}))^2 + \lambda\|\mathbf{w}\|_2^2 \tag{6.9}$$

where $\lambda \triangleq \sigma^2/\tau^2$. Note, that the first term is still the MSE/NLL and the second term is the complexity penalty. The corresponding normal equation solution is:

$$\hat{w}_{\text{ridge}} = \left(\lambda \mathbf{I}_D + \mathbf{X}^\mathsf{T}\mathbf{X}\right)^{-1}\mathbf{X}^\mathsf{T}\mathbf{y} \tag{6.10}$$

This is called **ridge regression** or **penalized least squares**. This method encourages the parameters to be small (by $\ell_2$ **regularization / weight decay**).

- For performing numerically stable linear regression normal equation see MLPP sect 7.5.2.
- For connection between PCA and linear regression see MLPP sect 7.5.3. In short the directions in which we are most uncertain about $\mathbf{w}$ are determined by the eigenvectors of the matrix $\mathbf{X}$ with the smallest eigenvalues. Hence small singular values $\sigma_j$ (where $\sigma_j^2$ are eigenvalues of $\mathbf{X}^\mathsf{T}\mathbf{X}$) correspond to directions with high posterior variance. These are the directions which the ridge shrinks the most.
- (MLPP sect 7.5.4) When you are modeling i.e. training a system from training data, the model eventually has three kinds of errors (1) **noise floor** which is an irreducible component that all models incur due to the intrinsic variability of the generating process; (2) **structural error** is the discrepancy between the generating process and the model; (3) **approximation error** is the discrepancy between the parameters that are estimated and the best parameters that can be estimated for a given model. If your model has enough degrees of freedom, and the size of the training data was just enough, you would create a model which has no structural error i.e. it will hit the noise floor. Typically this would happen much quicker (as a function of the amount of training data) for simpler models. Same is true for approximation error.
- **Bayesian Linear Regression** (MLPP sect 7.6) So ridge or OLS regression gives us a point estimate of our parameters - but if we wanted to compute the full posterior over $\mathbf{w}$ and $\sigma^2$ you need a bayesian approach. So this will allow us a full posterior from which we can not only know the MAP solution (which is the mean/mode of the gaussian) but we will also be able to sample models from this distribution.

# 7 Logistic Regression

- Logistic Regression is a machine learning method for classification. There are both **binary classification** problems and **multi-class classification** problems. Linear regression can be adjusted to perform binary classification - where we would fit a line to our labels $y \in \{0, 1\}$ given $\mathbf{x}$. This $\mathbf{x}^*$ will give a threshold for giving labels to our testing data.
- One way to build a probabilistic classifier is to create a joint model of the form $\mathcal{P}(y, \mathbf{x})$ and then to condition on x, thereby deriving $\mathcal{P}(y|\mathbf{x})$. This is called the **generative approach**. An alternative approach is to fit a model of the form $\mathcal{P}(y|\mathbf{x})$ directly. This is called the **discriminative approach**. (MLPP sect 8.1)

- (MLPP sect 8.2) Logistic regression is a discriminative approach to classification, which is linear in its parameters. It is modeled as:

$$\mathcal{P}(y|\mathbf{x},\mathbf{w}) = \text{Ber}(y|\text{sigm}(\mathbf{w}^\mathsf{T}\mathbf{x})), \qquad \text{sigm}(\mathbf{w}^\mathsf{T}\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^\mathsf{T}\mathbf{x})} \tag{7.1}$$

The latter is called the **sigmoid function / logistic function**. If we threshold the probability at 0.5, we induce a linear decision boundary whose normal is given by $\mathbf{w}$. In other words we say that $y = 1$ when $\mathbf{w}^\mathsf{T}\mathbf{x} \geq 0$ (this is what defines the decision boundary/plane). If we suppose $y^{(i)} \in \{-1, +1\}$, the negative log likelihood is:

$$\ell_{\text{-}}(\mathbf{w}) = \sum_{i=1}^{N} \log\left(1 + \exp(-y^{(i)}\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)})\right) \tag{7.2}$$

Where $\ell_{\text{-}}(\cdot)$ is the symbol for the negative log likelihood (NLL). In a more typical case $y^{(i)} \in \{0, 1\}$, we can write the cost function as:

$$\ell_{\text{-}}(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N} y^{(i)} \log\left(\text{sigm}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)})\right) + \left(1 - y^{(i)}\right) \log\left(1 - \text{sigm}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)})\right) \tag{7.3}$$

Let's take a look at this negative log likelihood cost function more carefully. If $y^{(i)} = 0$ you would want to get $\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} \leq 0$, or pay a penalty otherwise. If $\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} \gg 0$, then $\text{sigm}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)}) = 1$, hence setting the second part of the equation to 0. Hence, for this point the cost function would be 0. On the other hand if the function was doing the right thing i.e. $\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)} \ll 0$, then $\text{sigm}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)}) = 0$, and this would reduce the cost function by $\frac{1}{N}$ (i.e. added $-\frac{1}{N}$). Therefore, the more correct decisions our parameter makes, the more negative NLL would be. Hence, minimizing this function would be the right thing to do. Since there is no closed form for this equation, you need to do gradient descent to reach the minima:

$$\mathbf{w}^{(m+1)} := \mathbf{w}^{(m)} - \eta\nabla\ell_{\text{-}}(\mathbf{w}^{(m)}), \qquad \nabla\ell_{\text{-}}(\mathbf{w}) = \begin{bmatrix} \frac{\partial}{\partial \mathbf{w}_1}\ell_{\text{-}}(\mathbf{w}) \\ \vdots \\ \frac{\partial}{\partial \mathbf{w}_K}\ell_{\text{-}}(\mathbf{w}) \end{bmatrix} \tag{7.4}$$

Here, $\eta$ is called the learning rate, whose setting is critical to the algorithm. If its set low, it can have very slow convergence, whereas setting it too high would have a ping ping effect where there is a chance of overshooting the target. We can show that the derivative would be of this form:

$$\frac{\partial}{\partial \mathbf{w}_j}\ell_{\text{-}}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}\left(\text{sigm}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(i)}) - \mathbf{y}^{(i)}\right)\mathbf{x}_j^{(i)} \tag{7.5}$$

Which can be turned into:

$$\nabla\ell_{\text{-}}(\mathbf{w}) = \frac{1}{N}\mathbf{X}^\mathsf{T}\begin{bmatrix} \text{sigm}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(1)}) - \mathbf{y}^{(1)} \\ \vdots \\ \text{sigm}(\mathbf{w}^\mathsf{T}\mathbf{x}^{(N)}) - \mathbf{y}^{(N)} \end{bmatrix} \tag{7.6}$$

- (MLPP sect 8.3.2) One problem with this algorithm is that how to set the learning rate $\eta$. One way to do this is to travel in the direction of the gradient till the gradient becomes 0 - this would ensure we have reached the minimum in that direction. From that point we would need to find the gradient again (which would be orthogonal to the previous direction - hence exhibiting zig-zag behavior), and then travel in that direction. This essentially gives a way to set $\eta$ in each iteration. This algorithm is called **Steepest descent**, where the learning rate is set by:

$$\eta^{(m)} = \underset{\eta \geq 0}{\arg\min}\, \ell_{\text{-}}\left(\mathbf{w} - \eta\nabla\ell_{\text{-}}(\mathbf{w})\right) = \underset{\eta \geq 0}{\arg\min}\, \phi_m(\eta) \tag{7.7}$$

This is called **line minimization** or **line search**. You can also add a momentum term $\text{sigm}((\mathbf{w}^{(m)}-\mathbf{w}^{(m-1)})^\mathsf{T}\mathbf{x})$ to reduce the zig-zag effect.

- **Newton's Method** is a method to do the same optimization which takes the curvature of the space (i.e. the Hessian) into account.

**Algorithm 1:** Newton's method

1. Initialize $\mathbf{w}^{(0)}$
2. **for** $m = 0, 1, \ldots$ *until convergence* **do**
3. Evaluate $\nabla \ell_\text{-}(\mathbf{w}^{(m-1)})$
4. Evaluate hessian $H_\ell(\mathbf{w}^{(m)}) = \nabla^2 \ell_\text{-}(\mathbf{w}^{(m)})$ where $H_\ell(\mathbf{w}) = \begin{bmatrix} \frac{\partial^2}{\partial \mathbf{w}_1^2}\ell_\text{-}(\mathbf{w}) & \cdots & \frac{\partial^2}{\partial \mathbf{w}_1 \mathbf{w}_K}\ell_\text{-}(\mathbf{w}) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2}{\partial \mathbf{w}_1 \mathbf{w}_K}\ell_\text{-}(\mathbf{w}) & \cdots & \frac{\partial^2}{\partial \mathbf{w}_K^2}\ell_\text{-}(\mathbf{w}) \end{bmatrix}$
5. Compute $\frac{d\phi_m(\eta)}{d\eta} = 0$ to find $\eta^{(m)}$, where $\phi_m(\eta) = \ell_\text{-}\left(\mathbf{w}^{(m)} - \eta H_\ell(\mathbf{w}^{(m)})^{-1}\nabla \ell_\text{-}(\mathbf{w}^{(m)})\right)$
6. $\mathbf{w}^{(m+1)} := \mathbf{w}^{(m)} - \eta^{(m)} H_\ell(\mathbf{w}^{(m)})^{-1}\nabla \ell_\text{-}(\mathbf{w}^{(m)})$

Note step 5 where you want to compute the roots of an arbitrary equation can be done by Newton-Rhapson method, where $\theta^{(m+1)} = \theta^{(m)} - \frac{f(\theta^{(m)})}{f'(\theta^{(m)})}$ is computed until convergence. The newton's method works if $H_\ell(\mathbf{w}^{(m)})$ is strictly positive definite. We can use the **Levenberg-Marquardt** modification which aims to keep the hessian positive definite. This becomes essentially a compromise between Newton's method and steepest descent. **Iteratively reweighted least squares (IRLS)** which is very similar is explained in MLPP sect 8.3.4.

- Since it might be expensive to compute the Hessian, **Quasi-Newton methods** try to iteratively approximate the Hessian using information gleaned from the gradient vector at each step. One of them is BFGS which can be thought of as a diagonal plus low rank approximation to the hessian.

- Just like in ridge regression, we can use $\ell_2$ regularization, where the new functions are of the form:

$$\ell_\text{-}^R(\mathbf{w}) = \ell_\text{-}^R(\mathbf{w}) + \lambda \mathbf{w}^\mathsf{T}\mathbf{w} \tag{7.8}$$

$$\nabla \ell_\text{-}^R(\mathbf{w}) = \nabla \ell_\text{-}(\mathbf{w}) + \lambda \mathbf{w} \tag{7.9}$$

$$H_\ell^R(\mathbf{w}) = H_\ell(\mathbf{w}) + \lambda \mathbf{I} \tag{7.10}$$

- For multi-class classification, you can use **multinomial logit regression** which has the following form (MLPP sect 8.3.7):

$$\mathcal{P}(y = c|\mathbf{x}, \mathbf{W}) = \frac{\exp(\mathbf{w}_c^\mathsf{T}\mathbf{x})}{\sum_{c'=1}^C \exp(\mathbf{w}_{c'}^\mathsf{T}\mathbf{x})} \tag{7.11}$$

The NLL is, given $y_{ic} = \mathbb{I}(y_i = c)$:

$$\ell_\text{-}(\mathbf{W}) = -\sum_{i=1}^N \left[\left(\sum_{c=1}^C y_{ic}\mathbf{w}_c^\mathsf{T}\mathbf{x}_i\right) - \log\left(\sum_{c'=1}^C \exp(\mathbf{w}_{c'}^\mathsf{T}\mathbf{x}_i\right)\right] \tag{7.12}$$

- **Bayesian Logistic Regression** (MLPP sect 8.4): We cannot compute the full posterior exactly because there is no appropriate conjugate prior.

- **Laplace approximation** (MLPP sect 8.4.1) where we use an approximation to the marginal likelihood with the help of taylor expansion. This is also called **Gaussian approximation** because the posterior becomes Gaussian like as the sample size increases.

- If the data is linearly separable, then there might be many parameter settings that correspond to the lines that separate the training data. Hence, the MLE is not well defined in such a case. Since large regression weights make the sigmoid very steep, turning it into a step function.
- The **Posterior Predictive** (MLPP 8.4.4) where we predict the class given the data:

$$\mathcal{P}(y|\mathbf{x},\mathcal{D}) = \int \mathcal{P}(y|\mathbf{x},\mathbf{w})\mathcal{P}(\mathbf{w}|\mathcal{D})d\mathbf{w} \tag{7.13}$$

  Since this is untractable, the simplest approximation in the binary case is (where $\mathbb{E}[\mathbf{w}]$ is the posterior mean):

$$\mathcal{P}(y=1|\mathbf{w},\mathcal{D}) \approx \mathcal{P}(y=1|\mathbf{x},\mathbb{E}[\mathbf{w}]) \tag{7.14}$$

  A better approximation is **Monte carlo approximation**:

$$\mathcal{P}(y=1|\mathbf{w},\mathcal{D}) \approx \frac{1}{S}\sum_{i=1}^{S} \text{sigm}((\mathbf{w}^s)^\mathsf{T}\mathbf{w}) \tag{7.15}$$

  where $\mathbf{w}^s \sim \mathcal{P}(\mathbf{w}|\mathcal{D})$ are just samples from the distribution. Even though the decision boundary is linear, the posterior predictive density is not linear. By averaging over multiple predictions, we see that the uncertainty in the decision boundary "splays out" as we move further from the training data. Note also that the posterior mean decision boundary is roughly equally far from both classes.
- **Online Learning** (MLPP 8.5): Suppose that at each step, "nature" presents a sample $\mathbf{z}_k$ and the "learner" must respond with a parameter estimate $\boldsymbol{\theta}_k$. The objective is to reduce **regret**, which is averaged loss incurred relative to the best we could have gotten in hindsight:

$$\text{regret}_k \triangleq \frac{1}{k}\sum_{t=1}^{k} f(\boldsymbol{\theta}_t, \mathbf{z}_t) - \min_{\boldsymbol{\theta}^* \in \Theta} \frac{1}{k}\sum_{t=1}^{k} f(\boldsymbol{\theta}_*, \mathbf{z}_t) \tag{7.16}$$

  One simple way to do is online gradient descenet: $\boldsymbol{\theta}_{k+1} = \text{proj}_\Theta\left(\boldsymbol{\theta}_k - \eta_k \nabla f(\boldsymbol{\theta}_k, \mathbf{z}_k)\right)$ where $\eta_k$ is the step size. Rather than minimizing regret, we can also minimize expected loss $f(\boldsymbol{\theta}) = \mathbb{E}\left[f(\boldsymbol{\theta}, \mathbf{z}\right]$ where the expectation is taken over future data. Since some variables in the objective are random, this is called **stochastic optimization**. One way is to compute $\boldsymbol{\theta}_{k+1}$ at each step - this is called **Stochastic Gradient Descent (SGD)**. The learning rate can be $\eta_k = 1/k$; or $\eta_k = (\tau_0 + k)^{-\kappa}$ where $\tau_0 \geq 0$ slows down early iterations of the algorithm and $\kappa \in (0.5, 1]$ controls the rate of forgetting old values.
- One disadvantage of SGD is that it uses the same step size for all parameter values. An alternative approach is **adagrad** (adaptive gradient) which is similar in spirit as a diagonal Hessian approximation. In particular, if $\theta_i(k)$ is parameter $i$ at time $k$, and $g_i(k)$ is its gradient, then we can update as follows:

$$\theta_i(k+1) := \theta_i(k) - \eta \frac{g_i(k)}{\tau_0 + \sqrt{s_i(k)}}, \qquad s_i(k) := s_i(k-1) + g_i(k)^2 \tag{7.17}$$

  This results in a per-parameter step-size which adapts to the curvature of the loss function.
- If we dont have an infinite data stream, we can simulate one by sampling data points at random from our

training set:

**Algorithm 2:** Stochastic gradient descent

```
Initialize θ, η
repeat
    Randomly permute data
    for i = 1 : N do
        g = ∇f(θ, z_i)
        θ ← proj_Θ(θ − ηg)
        Update η
until converged
```

In addition to enhanced speed, SGD is often less prone to getting stuck in shallow local minima, because it adds a certain amount of "noise", and spends more cycles searching through the parameter space, rather than exhaustively computing the gradient of the loss function.

- **Least Mean Squares (LMS)** algorithm is a way to do online linear regression. Here the online gradient is $\mathbf{g}_k = \mathbf{x}_k(\boldsymbol{\theta}_k^\mathsf{T}\mathbf{x}_k - y_k)$. After computing the gradient, we take a step in the direction $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k(\hat{y}_k - y_k)\mathbf{x}_k$, where $y_k$ is true response, and $\hat{y}_k = \boldsymbol{\theta}_k^\mathsf{T}\mathbf{x}_k$. Note that LMS may require multiple passes through the data to find the optimum. By contrast, the recursive least squares algorithm, which is based on the Kalman filter and which uses second-order information, finds the optimum in a single pass

- **Perceptron Algorithm** (MLPP sect 8.5.4) is online logistic regression. The batch gradient was given in Equation 7.5. In the online case, the weight update is:

$$\boldsymbol{\theta}_k = \boldsymbol{\theta}_{\boldsymbol{k-1}} - \eta_k \mathbf{g}_i, \qquad \mathbf{g}_i \triangleq (\mu_i - y_i)\mathbf{x}_i \tag{7.18}$$

  where $\mu_i = \mathcal{P}(y_i = 1|\mathbf{x}_i, \boldsymbol{\theta}_k) = \mathbb{E}[y_i|\mathbf{x}_i, \boldsymbol{\theta}_k]$. This is exactly the same form as LMS. Now if we consider the case where $\hat{y}_i = \arg\max_{y\in\{0,1\}} \mathcal{P}(y|\mathbf{x}_i, \boldsymbol{\theta})$ which represents the most probable class. If we consider $y = \{-1, +1\}$ rather than $y = \{0, +1\}$, then $\hat{y}_i y_i = -1$ would be an error and $\hat{y}_i y_i = +1$ would be a correct estimate. Hence, we can set $\hat{y}_i = \text{sign}(\boldsymbol{\theta}^\mathsf{T}\mathbf{x}_i)$. At each step, we update the weight vector by adding on the gradient. The key observation is that, if we predicted correctly, then $\hat{y}_i = y_i$ , so the (approximate) gradient is zero and we do not change the weight vector. The perceptron algorithm would converge if the data is linearly separable.

- The Bayesian way to do online learning would be:

$$\mathcal{P}(\boldsymbol{\theta}|\mathcal{D}_{1:k}) = \mathcal{P}(\mathcal{D}_k|\boldsymbol{\theta})\mathcal{P}(\boldsymbol{\theta}|\mathcal{D}_{1:k-1}) \tag{7.19}$$

  This has the obvious advantage of returning a posterior instead of just a point estimate. It also allows for the online adaptation of hyper-parameters, which is important since cross-validation cannot be used in an online setting. Finally, it has the (less obvious) advantage that it can be quicker than SGD (because effectively we associate a different learning rate to each parameter).

- **Generative vs Discriminative** (MLPP 8.6.1) (also see properties of different methods in Table 8.1):
  - Usually the assumptions in discriminative models are stronger than generative models.
  - In training discriminative models we usually maximize the conditional log likelihood, whereas in generative we maximize the joint log likelihood.
  - Since the discriminative models do not need to model the distribution of features, discriminative models might require less data if the model is correct.
  - Usually building generative models is easy. Plus since we estimate parameters for each class independently, adding classes doesn't require re-training - which is not the case in discriminative.

- Dealing with missing data in generative classifiers is much easier. We can model the reason for missing data which can help in inference (MLPP 8.6.2). There is also the case of missing-ness in training data and testing data. The former is harder to deal than latter. When the class label is sometimes missing during training time, it is called **semi-supervised** learning. If some feature is missing during test time, we can marginalize it out (MLPP 8.6.2.1). For instance naïve Bayes classifier which deals with features independetly, we can skip missing features by just not multiplying/ignoring it.
- Semi-supervised learning is much easier with generative classifiers.
- Generative classifier model the joint, and hence treat the input $\mathbf{x}$ and output $y$ symmetrically. So given one, you can always infer the other.
- It is much easier to deal with preprocessing (like using a basis function $\psi(\mathbf{x})$, rather than $\mathbf{x}$) in discriminative models. It is hard in generative models because then the features are correlated in complex ways.
- Some generative models make strong independence assumptions (like naïve Bayes) which might not be true. This results in extreme posterior class probabilities (very near 0 or 1).

- **Fisher Linear Discriminant Analysis (FLDA)** (MLPP 8.6.3): Since discriminant (generative model for classification) analysis requires fitting a MVN to the features, it can be problematic in high dimensions. One approach would be to reduce the dimensions to $\mathbf{z} \in \mathbb{R}^L$ and then fit an MVN over it. You can always construct a $\mathbf{W} \in \mathbb{R}^{L \times D}$ through PCA such that $\mathbf{z} = \mathbf{W}\mathbf{x}$. Since, PCA is an unsupervised approach (i.e. it is agnostic to class labels), the low dimensional $\mathbf{z}$ might not result in good separability. To deal with this we have FLDA in which $\mathbf{W}$ has the assumption that the low-dimensional data can be classified as well as possible using a Gaussian class-conditional density model. The drawback of this technique is that it is restricted $L \leq C - 1$, regardless of $D$. For a two class case, the idea behind FLDA would be to find a line to project the data on such that the distance between the means $m_k$ is maximized, while the ensuring the projected clusters are tight i.e. the variance $s_k$ is small:

$$J(\mathbf{w}) = \frac{(m_2 - m_1)^2}{s_1^2 + s_2^2} = \frac{\mathbf{w}^\mathsf{T}\mathbf{S}_B\mathbf{w}}{\mathbf{w}^\mathsf{T}\mathbf{S}_W\mathbf{w}} \tag{7.20}$$

where $\mathbf{S}_B$ is the inter-class scatter matrix, and $\mathbf{S}_W$ is the intra-class scatter matrix.

# 8 Generalized Linear Models and the Exponential Family

- **Exponential family** (MLPP 9.1) is a group of distributions. Any member if this family can be used as class-conditional density in order to make a generative classifier. It can also be used as a response variable in discriminative models, where the mean is a linear function of the inputs (**Generalized Linear Model**).
- Why the exponential family (MLPP 9.2)? Under certain regularity conditions it can be shown that the exponential family has finite-sized sufficient statistics (**Pitman-Koopman-Darmois theorem**). It is also the only family for which conjugate priors exist. It can also be shown that it makes the least set of assumptions given user constraints.
- (MLPP 9.2.1) The exponential family has a pdf or pmf $\mathcal{P}(\mathbf{x}|\boldsymbol{\theta})$ where $\mathbf{x} \in \mathcal{X}^m$ and $\boldsymbol{\theta} \in \Theta \subset \mathbb{R}^d$:

$$\mathcal{P}(\mathbf{x}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} h(\mathbf{x}) \exp\left[\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x})\right] \tag{8.1}$$

$$= h(\mathbf{x}) \exp\left[\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}) - A(\boldsymbol{\theta})\right] \tag{8.2}$$

where

$$Z(\boldsymbol{\theta}) = \int_{\mathcal{X}^m} h(\mathbf{x}) \exp\left[\boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x})\right] d\mathbf{x} \tag{8.3}$$

$$A(\boldsymbol{\theta}) = \log Z(\boldsymbol{\theta}) \tag{8.4}$$

| Symbol | Meaning |
|---|---|
| $\boldsymbol{\theta}$ | natural parameters / canonical parameters |
| $\boldsymbol{\phi}(\mathbf{x}) \in \mathbb{R}^d$ | vector of sufficient statistics. If $\boldsymbol{\phi}(\mathbf{x}) = \mathbf{x}$ then its a natural exponential family |
| $Z(\boldsymbol{\theta})$ | partition function |
| $A(\boldsymbol{\theta})$ | log partition function / cumulant function |
| $h(\mathbf{x})$ | scaling constant, often equal to 1 |

In Equation 8.1, $\boldsymbol{\theta}$ can be replaced with $\eta(\boldsymbol{\theta}) \triangleq \boldsymbol{\eta}$, i.e. $\eta(\cdot)$ maps parameters to canonical parameters $\boldsymbol{\eta}$. If $\dim(\boldsymbol{\theta}) < \dim(\boldsymbol{\eta})$ it is called the curved exponential family, which means we have more sufficient statistics than parameters. If $\eta(\boldsymbol{\theta}) = \boldsymbol{\theta}$, the model is said to be in canonical form.

- For examples on how bernoulli, multinoulli, univariate gaussian are from the exponential family (MLPP 9.2.2).
- (MLPP 9.2.3) The log partition function $A(\boldsymbol{\theta})$ is important because its derivatives can be used to generate **cumulants** of the sufficient statistics. The cumulants of a random variable are defined by the logarithm of the moment-generating function. For a single parameter case:

$$\frac{dA(\theta)}{d\theta} = \mathbb{E}\left[\phi(x)\right], \qquad \frac{d^2A(\theta)}{d\theta^2} = \mathbb{E}\left[\phi^2(x)\right] - \mathbb{E}\left[\phi(x)\right]^2 = \operatorname{var}\left[\phi(x)\right] \tag{8.5}$$

  For the multivariate case, $\nabla^2 A(\boldsymbol{\theta}) = \operatorname{cov}\left[\boldsymbol{\phi}(\mathbf{x})\right]$. Since the covariance is positive definite, we see that $A(\boldsymbol{\theta})$ is a convex function.

- (MLPP 9.2.4) The likelihood of an exponential family model has the form:

$$\mathcal{P}(\mathcal{D}|\boldsymbol{\theta}) = \left[\prod_{i=1}^{N} h(\mathbf{x}^{(i)})\right] g(\boldsymbol{\theta})^N \exp\left(\boldsymbol{\eta}(\boldsymbol{\theta})^\mathsf{T} \left[\sum_{i=1}^{N} \boldsymbol{\phi}(\mathbf{x}^{(i)})\right]\right) \tag{8.6}$$

  And the sufficient statistics are $N$ and:

$$\boldsymbol{\theta}(\mathcal{D}) = \left[\sum_{i=1}^{N} \phi_1(\mathbf{x}^{(i)}), \ldots, \sum_{i=1}^{N} \phi_K(\mathbf{x}^{(i)})\right] \tag{8.7}$$

  For instance, the univariate Gaussian has the sufficient statistics $\boldsymbol{\theta}(\mathcal{D}) = \left[\sum_i \mathbf{x}^{(i)}, \sum_i \mathbf{x}^{(i)^2}\right]$ and $N$. One of the conditions on the exponential family is that support of the distribution is not dependent on the parameter. For instance, the uniform distribution is not exponential family because its support set, $\mathcal{X}$, depends on the parameters.

- The MLE, the empirical average of the sufficient statistics must equal the model's theoretical expected sufficient statistics, i.e., $\hat{\boldsymbol{\theta}}$ must satisfy:

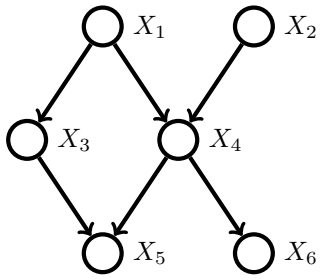
$$\mathbb{E}\left[\boldsymbol{\phi}(\mathbf{X})\right] = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{\phi}(\mathbf{x}^{(i)}) \tag{8.8}$$

- If the prior is conjugate to the likelihood, it means that the prior $\mathcal{P}(\boldsymbol{\theta}|\boldsymbol{\tau})$ has the same form as the likelihood $\mathcal{P}(\mathcal{D}|\boldsymbol{\theta})$. For this to make sense, we require that the likelihood have finite sufficient statistics, so that we can write $\mathcal{P}(\mathcal{D}|\boldsymbol{\theta}) = \mathcal{P}(s(\mathcal{D})|\boldsymbol{\theta})$.
- See the form of the likelihood, prior, posterior, and posterior predictive in (MLPP 9.2.5).
- The principle of maximum entropy (MLPP 9.2.6) says we should pick the distribution with maximum entropy (i.e. as close to uniform as possible), subject to the constraints that the moments of the distribution match the empirical moments of the specified functions. This would result in an exponential distribution - showing that this is the family of distributions which makes the least number of assumptions.

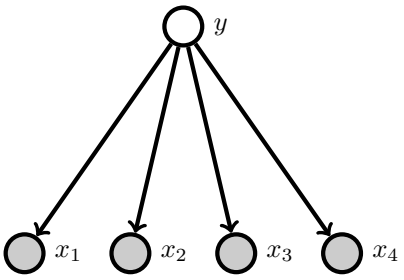
- Linear and logistic regression are examples of **Generalized Linear Models (GLMs)** (MLPP 9.3). These are models in which the output density is in the exponential family, and in which the mean parameters are a linear combination of the inputs, passed through a possibly nonlinear function, such as the logistic function.
- Logistic regression is done with a model of the form $\mathcal{P}(y = 1|\mathbf{x}_i, \mathbf{w}) = \text{sigm}(\mathbf{w}^T\mathbf{x}_i)$. In general any function $g^{-1}$ ($\mathcal{P}(y = 1|\mathbf{x}_i, \mathbf{w}) = g^{-1}(\mathbf{w}^T\mathbf{x}_i)$) that maps $[-\infty, \infty]$ to $[0, 1]$ is valid for regression. In **Probit Regression** (MLPP 9.4), $g^{-1}(\eta) = \Phi(\eta)$, where $\Phi(\eta)$ is the cdf of the standard normal. This function is very similar to the logistic (sigmoid) function. One advantage of probit regression is that it can be used for classification where the labels are ordinal (ranked) - this is called ordinal regression.
- At times we want to fit many related classification or regression models. It is often reasonable to assume that the input-output mapping is similar across these different models, so it might give better performance by fitting all the parameters at the same time. This is called **multi-task learning**, **transfer learning**, or **learning to learn**. Given different $J$ groups, each with $N$ items, we have $(\mathbf{x}_{1,1}, y_{1,1}), \ldots, (\mathbf{x}_{N_1,1}, y_{N_1,1}), \ldots, (\mathbf{x}_{N_J,J}, y_{N_j,J})$. The goal is to fit models $\mathcal{P}(y_j|\mathbf{x}_j)$ for all $j$ i.e. groups. Even though we want to fit individual models across different groups, there might not be enough data to fit on a group. As a compromise we fit a model on each group, but encourage the model parameters to be similar across groups. More precisely, suppose $\mathbb{E}[y_{ij}|\mathbf{x}_{ij}] = g(\mathbf{x}_{ij}^\mathsf{T}\boldsymbol{\beta}_j)$, where $g$ is the link function for the GLM. Furthermore, suppose $\boldsymbol{\beta}_j \sim \mathcal{N}(\boldsymbol{\beta}_*, \sigma_j^2\mathbf{I})$, and $\boldsymbol{\beta}_* \sim \mathcal{N}(\boldsymbol{\mu}, \sigma_*^2\boldsymbol{I})$. Note, $\sigma_j^2$ controls how much group $j$ depends on common parents, and $\sigma_*^2$ controls the strength of the overall prior. Here, the model with smaller samples borrows statistical strength from larger samples because $\boldsymbol{\beta}_j$s are linked through $\boldsymbol{\beta}_*$. You can create a log probability to optimize the parameters using (MLPP Eq 9.110). This is called **Hierarchical Bayes for Multi-task Learning**.
- Suppose we generalize the multi-task learning scenario to allow the response to include information at the group level, $\mathbf{x}_j$, as well as at the item level, $\mathbf{x}_{ij}$. Similarly, we can allow the parameters to vary across groups, $\boldsymbol{\beta}_j$, or to be tied across groups, $\boldsymbol{\alpha}$. For instance suppose $y_{ij}$ be amount of cholesterol for person $j$ at measurement $i$. Let $x_{ij}$ be the age of the person, and $x_j$ be the ethnicity. The primary goal is to determine if there are significant differences in the mean cholesterol among different ethnicities, after accounting for age. This can be modeled as a **generalized linear mixed effects model (GLMM)**.
- **Learning to Rank** (MLPP 9.7) is a problem of finding a function that can rank order a set of items given a query. Suppose we have a query $q$ and a set of documents $d^1, \ldots, d^m$ that might be relevant to $q$. We would like to sort the documents in decreasing order of relevance. Let's say we define the relevance of a document $d$ to query $q$, as $\text{sim}(q, d) \triangleq \mathcal{P}(q|d) = \prod_{i=1}^{n} \mathcal{P}(q_i|d)$, where $q_i$ is the $i$th word in the query and $\mathcal{P}(q_i|d)$ is a multinoulli distribution estimated from document $d$:
  - One method for learning to rank is a pointwise approach (MLPP 9.7.1), where we collect training data where we give the relevance of set of documents for each query. We define a feature vector $\mathbf{x}(q, d)$ for each query document pair. If the training labels are binary (i.e. relevant or not relevant), we have standard binary classification $\mathcal{P}(y = 1|\mathbf{x}(q, d))$. If the training labels are ordered relevancy labels, we can use ordinal regression to give a rating $\mathcal{P}(y = r|\mathbf{x}(q, d))$.
  - Since it is easier to compare to objects rather than give an absolute relevance, we can develop a pairwise approach (MLPP 9.7.2). In this case we have binary classification $\mathcal{P}(y_{jk} = 1|\mathbf{x}(q, d_j), \mathbf{x}(q, d_k))$ if the relevance of document $d_j$ is greater than $d_k$. If we map the relevance to a scoring function $f(\cdot)$, we can transform this to:

$$\mathcal{P}(y_{jk} = 1|\mathbf{x}(q, d_j), \mathbf{x}(q, d_k)) = \text{sigm}\left(f(\mathbf{x}_j) - f(\mathbf{x}_k)\right) \tag{8.9}$$

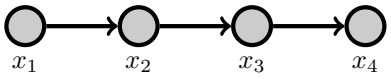
  - There's also a listwise approach which considers the relative relevance of all the documents simultaneously (MLPP 9.7.3).
- You can see different loss functions for ranking at (MLPP 9.7.4) which can be used for optimizing to find an optimal ranking.



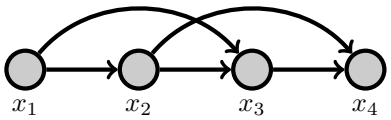

(a) Example Directed Acyclic Graph




(b) Naïve Bayes Classifier




(c) First order Markov chain




(d) Second order Markov chain

Figure 1: Examples of DGM. Grayed nodes indicate observed variables, whereas white indicate hidden/latent ones.

# 9 Directed Graphical Models (Bayes Network / Belief Network)

- The **chain rule** (MLPP 10.1.1) which can be used to represent a joint distribution as follows (ordering of variables could be changed):

$$\mathcal{P}(x_{1:V}) = \mathcal{P}(x_1)\mathcal{P}(x_2|x_1)\mathcal{P}(x_3|x_1,x_2)\mathcal{P}(x_4|x_1,x_2,x_3)\dots\mathcal{P}(x_V|x_{1:V-1}) \tag{9.1}$$

  The conditioning parameter $\theta$ is dropped due to brevity. Suppose each variable has $K$ states, then $\mathcal{P}(x_1)$ can be represented by $\mathcal{O}(K)$ numbers ($K-1$ to be exact). $\mathcal{P}(x_2|x_1)$ can be represented with $\mathcal{O}(K^2)$ numbers by writing $\mathcal{P}(x_2 = j|x_1 = i) = T_{ij}$; we say that $\mathbf{T}$ is a stochastic matrix because each row needs to sum to 1: $\sum_j T_{ij} = 1$. $T$ is called the **conditional probability table (CPT)**. You can condense the CPT to a **condition probability distribution (CPD)** which is useful for evaluating the probability once all $x_{1:T}$ is observed; we can also use it for class-conditional density $\mathcal{P}(\mathbf{x}|y = c)$, thus making generative classifier. But, this is not useful for prediction because it depends on all other variables.

- Recall that variables $X$ and $Y$ are **conditionally independent (CI)** given $Z$ if:

$$X \perp Y|Z \iff \mathcal{P}(X,Y|Z) = \mathcal{P}(X|Z)\mathcal{P}(Y|Z) \tag{9.2}$$

  If we suppose $x_{t+1} \perp x_{1:t-1}|x_t$, or in other words "the future is independent of the past, given the present." This is called the (first order) **markov assumption**. We can use the same assumption to break down the joint distribution in Equation 9.1:

$$\mathcal{P}(x_{1:V}) = \mathcal{P}(x_1)\prod_{t=2}^{V}\mathcal{P}(x_t|x_{t-1}) \tag{9.3}$$

  This is called the (first order) **markov chain**. This can be represented by an initial distribution over states $\mathcal{P}(x_1 = k)$, plus a **state transition matrix** $\mathcal{P}(x_t = j|x_{t-1} = i)$

- A **graphical model (GM)** is a way to represent a joint distribution by making CI assumptions. In particular, the nodes in the graph represent random variables, and the (lack of) edges represent CI assumptions.

- A **directed graphical model (DGM)** is a graphical model whose graph is a DAG (**directed acyclic graph (DAG)** is a directed graph with no directed cycles. On a similar note, a **directed tree** is a DAG. If we allow a node to have multiple parents, its a **polytree**; and with single parents it is a **moral tree**). DGM are also called Bayesian Networks, Belief Networks, or Causal Networks.

- A **Markov blanket** of a variable is its parents, its children, and the children of its parents (excluding itself) (BRML Def 2.5). For instance, in Figure 1a, the markov blanket of variable $x_4$ is $\underbrace{x_1, x_2}_{\text{parents}}, \underbrace{x_5, x_6}_{\text{children}}, \underbrace{x_3}_{\text{parents of children}}$.

We can also say that all the information about $x_i$ is contained in its markov blanket $\mathcal{M}_B(x_i)$. In other words two variables $x_i$ and $x_j$ (where $x_j \notin \mathcal{M}_B(x_i)$) are conditionally independent given the markov blanket of one of them i.e. $x_i \perp x_j \,|\, \mathcal{M}_B(x_i)$ (BRML Remark 3.3). Essentially, the markov blanket $\mathcal{M}_B(x_i)$ is the smallest set of nodes which renders the node $x_i$ conditionally independent of all other nodes in the graph. In Figure 1a, for $x_4$, the **full conditional** is the probability involving all conditionals in the markov blanket: $\mathcal{P}(x_4|x_1, x_2)\mathcal{P}(x_6|x_4)\mathcal{P}(x_5|x_3, x_4)$.

- The key property of DAGs is that the nodes can be ordered such that parents come before children. This is called a topological ordering, and it can be constructed from any DAG. Given such an order, we define the **ordered Markov property** to be the assumption that a node only depends on its immediate parents, not on all predecessors in the ordering, i.e.:

$$x_s \perp x_{\text{pred}(s)\setminus \mathsf{pa}(s)} \mid x_{\mathsf{pa}(s)} \tag{9.4}$$

where $\text{pred}(s)$ are all the predecessors of $s$, whereas $\mathsf{pa}(s)$ are just the parents of $s$. This is a natural generalization of the first-order Markov property to from chains to general DAGs. For instance for the DAG in Figure 1a encodes the following joint distribution:

$$\mathcal{P}(\mathbf{x}_{1:6}) = \mathcal{P}(x_1)\mathcal{P}(x_2|\cancel{x_1})\mathcal{P}(x_3|x_1, \cancel{x_2})\mathcal{P}(x_4|x_1, x_2, \cancel{x_3})\mathcal{P}(x_5|\underline{x_1, \cancel{x_2}}, x_3, x_4)\mathcal{P}(x_6|\underline{x_1, \cancel{x_2}, \cancel{x_3}}, x_4, \cancel{x_5}) \tag{9.5}$$

$$= \mathcal{P}(x_1)\mathcal{P}(x_2)\mathcal{P}(x_3|x_1)\mathcal{P}(x_4|x_1, x_2)\mathcal{P}(x_5|x_3, x_4)\mathcal{P}(x_6|x_4) \tag{9.6}$$

$$\mathcal{P}(\vec{x}_{1:V}|G) = \prod_i \mathcal{P}(x_i|x_{\mathsf{pa}(i)}) \tag{9.7}$$

The part $\mathcal{P}(\cdot|G)$ is there to indicate that the CI are only true for the graph $G$. Note that if each node has $K$ states and has $\mathcal{O}(F)$ parents, then the number of total parameters in all the CPTs will be $\mathcal{O}(VK^F)$, which is much less than the model which makes no CI assumptions, $\mathcal{O}(K^V)$.

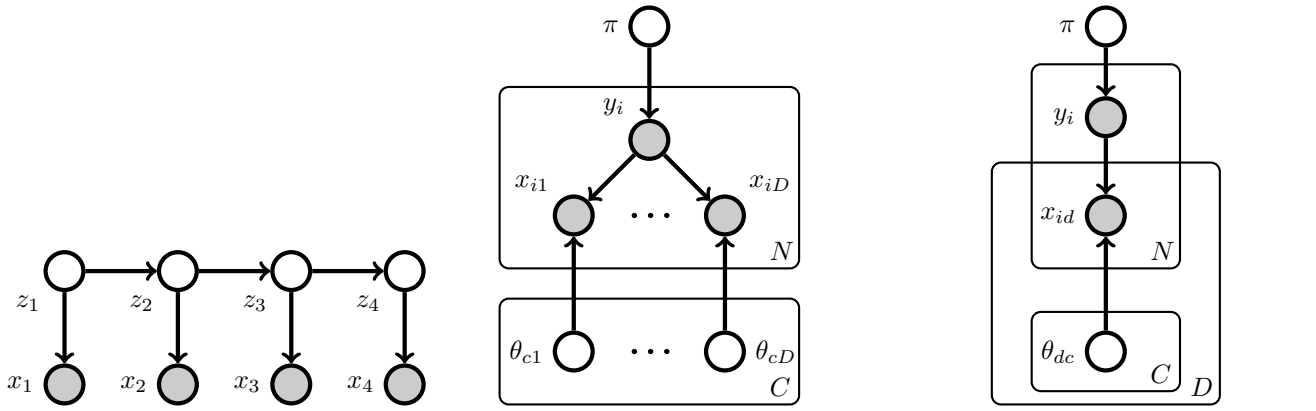
  - (MLPP 10.2.1) Naïve Bayes Classifier assumes the features are conditionally independent given the class label. This is illustrated in Figure 1b. This allows us to write the distribution as follows, where $D$ is the number of features:

$$\mathcal{P}(y, \mathbf{x}) = \mathcal{P}(y) \prod_{j=1}^{D} \mathcal{P}(x_j|y) \tag{9.8}$$

  - (MLPP 10.2.2) In a first order markov chain (Figure 1c) the assumption as that for random variable $x_t$, knowing $x_{t-1}$ is enough to know about $\mathbf{x}_{1:t-2}$. In a second order markov chain (Figure 1d) the dependence for $x_t$ include $x_{t-2}$ apart from $x_{t-1}$:

$$\mathcal{P}(\mathbf{x}_{1:V}) = \mathcal{P}(x_1)\mathcal{P}(x_2|x_1) \prod_{t=3}^{V} \mathcal{P}(x_t|x_{t-1}, x_{t-2}) \tag{9.9}$$

    Unfortunately even second order markov assumption maybe inadequate if there are long-range correlations amongst observations (ofcourse building higher and higher order models would result in having to estimate too many parameters). The alternative is to assume that there is an underlying hidden process that can be modeled by a first-order Markov chain, but that the data is a noisy observation of this process. The result is known as a **Hidden Markov model (HMM)**, and is illustrated in Figure 2a. Here, $\mathbf{z}$ are the hidden variables, and $\mathbf{x}$ are observed variables. The CPD $\mathcal{P}(z_t|z_{t-1})$ is the **transition model**, whereas the CPD $\mathcal{P}(x_t|z_t)$ is the **observation model**. We are usually interested in inferring the hidden variables - like the words spoken from the observed variable - like the sound waveform. We would like to compute the hidden state given the data, i.e. to compute $\mathcal{P}(z_t|\mathbf{x}_{1:t}, \boldsymbol{\theta})$ - this is called **state estimation**.




(a) First order HMM (b) Naïve Bayes with single plates (c) Naïve Bayes with nested plates

Figure 2: Other DGM examples

- Generally, **Inference** in a graphical model simply means computation of marginal probabilities. Mathematically, marginal probabilities are defined in terms of sums over all the possible states of all the other nodes in the system. For example if we want the marginal probability of the last node $\mathcal{P}(x_N)$, we in general need to compute:

$$\mathcal{P}(x_N) = \sum_{x_1} \sum_{x_2} \cdots \sum_{x_{N-1}} \mathcal{P}(x_1, x_2, \cdots, x_N) \tag{9.10}$$

  In later Sections, we will refer to the marginal probabilities that we compute approximately as "beliefs," and denote the belief of a node $i$ as $\mathsf{bel}(x_i)$.

- Estimating unknown quantities from known ones is called **probabilistic inference** (MLPP 10.3). Concretely, inference refers to computing the posterior distribution of unknowns $\mathbf{x}_h$, given the knowns $\mathbf{x}_v$:

$$\mathcal{P}(\mathbf{x}_h|\mathbf{x}_v, \boldsymbol{\theta}) = \frac{\mathcal{P}(\mathbf{x}_h, \mathbf{x}_v|\boldsymbol{\theta})}{\mathcal{P}(\mathbf{x}_v|\boldsymbol{\theta})} = \frac{\mathcal{P}(\mathbf{x}_h, \mathbf{x}_v|\boldsymbol{\theta})}{\sum_{\mathbf{x}'_h} \mathcal{P}(\mathbf{x}'_h, \mathbf{x}_v|\boldsymbol{\theta})} \tag{9.11}$$

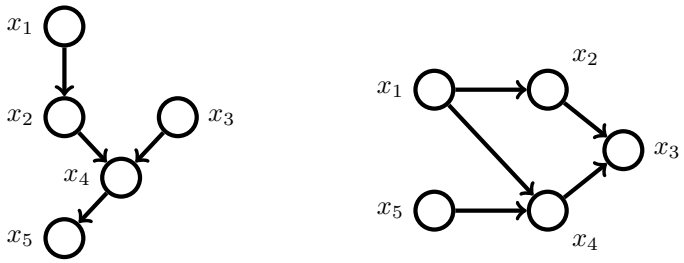
  Essentially we are **conditioning** on the data by **clamping** the visible variables to their observed values, $\mathbf{x}_v$, and then normalizing, to go from $\mathcal{P}(\mathbf{x}_h, \mathbf{x}_v)$ to $\mathcal{P}(\mathbf{x}_h|\mathbf{x}_v)$. Also if some of the hidden variables are not of interest, we can marginalize them out: $\mathcal{P}(\mathbf{x}_q|\mathbf{x}_v, \boldsymbol{\theta}) = \sum_{\mathbf{x}_n} \mathcal{P}(\mathbf{x}_q, \mathbf{x}_n|\mathbf{x}_v, \boldsymbol{\theta})$.

- **Learning** (MLPP 10.4) usually means computing a MAP estimate of the parameters from the data:

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{N} \log \mathcal{P}(\mathbf{x}_{i,v}|\boldsymbol{\theta}) + \log \mathcal{P}(\boldsymbol{\theta}) \tag{9.12}$$

  where $\mathbf{x}_{i,v}$ are the visible variables in case $i$. To a Bayesian, there is no difference between learning and inference because the parameters are unknown variables and should also be inferred. In this view, the main difference between hidden variables and parameters is that the number of hidden variables grows with the amount of training data (since there is usually a set of hidden variables for each observed data case), whereas the number of parameters in usually fixed (at least in a parametric model).

- When inferring parameters from data, we often assume the data is iid. This is illustrated between the $D$ features of $x_{id}$ in Figure 2b. This illustrates the assumption that each case was generated independently but from the same distribution. Notice that the $N$ data points are only independent conditional on the parameter




(a) $x_4$ is a collider on the path $x_1 - x_2 - x_4 - x_3$

(b) An example where $x_1$ and $x_5$ are dependent given $x_3$

Figure 3: DGM concepts for d-separation and d-connection

$\pi$; marginally, the data cases are dependent. Nevertheless, we can see that, in this example, the order in which the data cases arrive makes no difference to our beliefs about $\pi$, since all orderings will have the same sufficient statistics. Hence we say the data is **exchangeable**.

- The **plates** (rounded rectangles) (MLPP 10.4.1) around nodes just represent repetition of variables by the amount specified in the lower right corner. For example, the top plate in Figure 2b specifies that there are $N$ data cases, and there are $C$ class parameters. In the latter case the dots between specify that there is a class parameter for each of the $D$ features. The model in Figure 2c specifies the same model as Figure 2b. One problem with this representation is that it doesn't indicate that $\theta_{dc}$ is used to generate $x_{id}$ iff $y_i = c$, otherwise it is ignored. This is an example of **context specific independence**, since CI relationship $x_{id} \perp \theta_{dc}$ only holds if $y_i \neq c$. The Naïve Bayes classifier joint probability in these figures can be written as:

$$\mathcal{P}(\mathbf{x}_i, y_i = c, \boldsymbol{\theta}_c, \pi) \;=\; \mathcal{P}(\pi) \prod_{i=1}^{N} \left( \mathcal{P}(y_i = c|\pi) \prod_{d=1}^{D} \mathcal{P}(x_{id} \,|\, y_i = c,\, \theta_{dc}) \right) \tag{9.13}$$

- At the heart of any graphical model is a set of conditional independence assumptions. We write $\mathbf{x}_A \perp_G \mathbf{x}_B \,|\, \mathbf{x}_C$ if $A$ is independent of $B$ given $C$ in the graph $G$. Let $I(G)$ be the set of all such CI statements encoded in the graph. We say that $G$ is an **I-map** (independence map) for distribution $p$, or that $p$ is Markov wrt $G$, iff $I(G) \subseteq I(p)$, where $I(p)$ is the set of all CI statements that hold for distribution $p$. In other words, the graph is an I-map if it does not make any assertions of CI that are not true of the distribution. Note that a fully connected graph is an I-map of all distributions, since it makes no CI assertions at all (since no edges are missing). We therefore say $G$ is a **minimal I-map** of $p$ if $G$ is an I-map of $p$, and if there is no $G' \subset G$ which is an I-map of $p$.

- (BRML 3.3.2) Given a path $\mathcal{P}$, a **collider** is a node $c$ on $\mathcal{P}$ with neighbors $a$ and $b$ on $\mathcal{P}$ such that $a \to c \leftarrow b$. If there is a path between $x$ and $y$ which contains a collider, and this collider (or any of its descendants) are not in the conditioning set, then this path does not make $x$ and $y$ dependent. If there is a path between $x$ and $y$ which contains no colliders and no conditioning variables, then this path **d-connects** $x$ and $y$. Note that a collider is defined relative to a path. In simple terms, the path should be such that the arrows adjacent to $d$ on the path should point toward it. Figure 3a shows that $x_4$ is collider on the path $x_1 - x_2 - x_4 - x_3$ but not on the path $x_1 - x_2 - x_4 - x_5$.

- Finding **d-separation** gives us an easy to find conditional independence in a graph. That is what d(ependence)-separation actually means. Two disjoint set of nodes $\mathcal{X}$ and $\mathcal{Y}$ are d-separated, given another disjoint set of observed nodes $\mathcal{E}$ iff each undirected path from every node $x \in \mathcal{X}$ to every node $y \in \mathcal{Y}$ is d-separated by $\mathcal{E}$: if $e \in \mathcal{E}$ then $x \perp y \,|\, e$. Now the question is, what is d-separation? We say an undirected path $\mathcal{P}$ is d-separated by a set of nodes $\mathcal{E}$ (containing the evidence) iff at least one of the following conditions are true:

- $\mathcal{P}$ contains a chain, $s \to m \to t$ or $s \leftarrow m \leftarrow t$, where $m \in \mathcal{E}$.
- $\mathcal{P}$ contains a tent or fork, $s \leftarrow m \to t$, where $m \in \mathcal{E}$.
- $\mathcal{P}$ contains a collider or v-structure, $s \to m \leftarrow t$, where $m \notin \mathcal{E}$ and nor any of $m$'s descendants are in $\mathcal{E}$.

Figure 3b where $x_1$ and $x_5$ are not d-separated given $x_3$ because none of the above conditions are true. The one of most interest is $x_4$ which is a collider on the path between $x_1$ and $x_5$ but, its descendent, $x_3$ is in the conditioning set. Hence, $x_1 \not\perp x_5 \,|\, x_3$.

- Some lessons on conditional independence (MLPP 10.5.1):
  - Observing a middle node of a chain breaks the Markov chain in two i.e. $x \to y \to z \;\Rightarrow\; x \perp z \,|\, y$.
  - Observing a root node separates its children (as in naïve Bayes classifier) i.e. $x \leftarrow y \to z \;\Rightarrow\; x \perp z \,|\, y$.
  - Observing a common child does *not* separate the parents, in other words, makes its parents *dependent* i.e. $x \to y \leftarrow z \;\Rightarrow\; x \not\perp z \,|\, y$. Although, if we marginalize $y$ out of the distribution, $x$ and $z$ become independence i.e. $x$ and $z$ are **marginally independent** over $y$.
  - Observing the parents of a node separates the node from all its non-descendants (excluding its parents) i.e. $t \perp \text{nd}(t) \backslash \mathsf{pa}(t) \,|\, \mathsf{pa}(t)$. This is called the **directed local markov property**.
- We can create an algorithm called **Bayes Ball algorithm** (MLPP 10.5.1) to see if nodes in $x \in \mathcal{X}$ and $z \in \mathcal{Z}$ are conditionally independent given node $y \in \mathcal{Y}$. The nodes $\mathcal{Y}$ are shaded, because they indicate that they have been observed (are in the conditional set). The aim of the game is to reach from any node in $\mathcal{X}$ to any node in $\mathcal{Z}$. If you can find such a path, then $\mathcal{X}$ and $\mathcal{Z}$ are conditionally dependent given $\mathcal{Y}$. The rules can be derived from the list above.
- **Influence (decision) diagrams** are used to represent multi-stage (Bayesian) decision problems by using graphical notation. It has three kinds of symbols: (1) random variables / chance nodes which are denoted by oval as before; (2) **decision nodes** where you have to take a decision/action, represented by rectangles; and (3) **utility nodes / value nodes** which tell the reward/loss for taking certain actions, represented by diamonds. What we need to computed is the expected utility given a decision. So let's say our decision variable is $d$. Let's say our utility depends on this decision $d$ and a latent state $h$. The utility is defined for each decision $d \in \mathcal{D}$ against the hidden state $h \in \mathcal{H}$: $U(d, h)$:

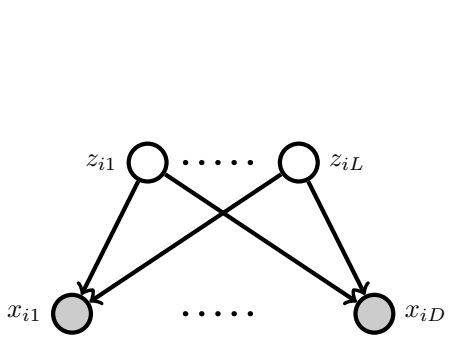
$$\text{EU}(d) = \sum_{h \in \mathcal{H}} \mathcal{P}(h) U(d, h) \tag{9.14}$$

We just need to find the decision $d^*$ which maximize the expected utility $\arg\max_d \text{EU}(d)$.

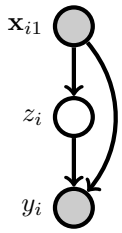
- **Partially Observed Markov Decision Process (POMDP)** (see Section 25) are basically HMMs augmented with actions and reward nodes (some explanation ). Unlike POMDP, in **Markov Decision Process (MDP) (see Section 25)** all states are fully observed. This is much easier to solve because we only have to compute the mapping from observed states to actions - which can be solved by dynamic programming (BRML 7.5).

# 10 Mixture Models and the EM Algorithm

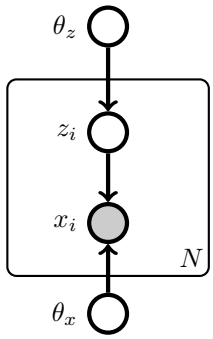
- Models with hidden variables are also known as **Latent Variable Models (LVMs)** (MLPP 11.1). Even though such models might be harder to fit, they have less parameters than models that directly represent correlation in the visible space. Plus the hidden variables can serve as a bottleneck, which computes a compressed representation of data. Figure 4a shows an LVM with $L$ latent variables, and $D$ visible variables, where usually $D \gg L$.




(a) A many-to-many LVM




(b) Mixture of Experts




(c) LVM represented as a DGM

- (MLPP 11.2) If the LVM has a discrete prior representing a discrete latent state i.e. $z_i \in \{1, \dots, K\}$, we will use a discrete prior for this, $\mathcal{P}(z_i) = \text{Cat}(\pi)$. For the likelihood we use $\mathcal{P}(\mathbf{x}_i|z_i = k) = \mathcal{P}_k(\mathbf{x}_i)$, where $\mathcal{P}_k$ is the $k$'th **base distribution** for the observations. The overall model is known as a **mixture model**, since we are mixing together $K$ base distributions as follows:

$$\mathcal{P}(\mathbf{x}_i|\boldsymbol{\theta}) = \sum_{k=1}^{K} \pi_k \mathcal{P}_k(\mathbf{x}_i|\boldsymbol{\theta}) \tag{10.1}$$

  This is a convex combination of the $\mathcal{P}_k$ since we are taking a weighted sum, where the **mixing weights** $\pi_k$ satisfy $0 \leq \pi_k \leq 1$ and $\sum_{k=1}^{K} \pi_k = 1$.

- **Mixture of Gaussians (MoG)** or **Gaussian Mixture Model (GMM)** (MLPP 11.2.1), is where each base distribution in the mixture is a multivariate Gaussian with mean $\boldsymbol{\mu}_k$ and covariance matrix $\boldsymbol{\Sigma}_k$:

$$\mathcal{P}(\mathbf{x}_i|\boldsymbol{\theta}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{10.2}$$

- Suppose our data consists of $D$-dimensional bit vectors. In this case the appropriate class conditional density is a product of Bernoullis, which is called the **Mixture of Multinoullis** (MLPP 11.2.2):

$$\mathcal{P}(\mathbf{x}_i|\mathbf{z}_i = k, \boldsymbol{\theta}) = \prod_{j=1}^{D} \text{Ber}(x_{ij}|\mu_{jk}) = \prod_{j=1}^{D} \mu_{jk}^{x_{ij}} (1 - \mu_{jk})^{1 - x_{ij}} \tag{10.3}$$

  where $\mu_{jk}$ is the probability that bit $j$ turns on in cluster $k$.

- Mixture models are used for density modeling $\mathcal{P}(\mathbf{x}_i)$ which are useful in data compression, outlier detection, and creating generative classifier, where we model each class-conditional density $\mathcal{P}(\mathbf{x}|y = c)$. Another use is to use them for clustering (MLPP 11.2.3), where we compute the posterior probability of point $\mathbf{x}_i$ belonging to mixture $k$. This is also known as the responsibility of cluster $k$ for point $i$:

$$r_{ik} \triangleq \mathcal{P}(z_i = k|\mathbf{x}_i, \boldsymbol{\theta}) = \frac{\mathcal{P}(z_i = k|\boldsymbol{\theta})\mathcal{P}(\mathbf{x}_i|z_i = k, \boldsymbol{\theta})}{\sum_{k'=1}^{K} \mathcal{P}(z_i = k'|\boldsymbol{\theta})\mathcal{P}(\mathbf{x}_i|z_i = k', \boldsymbol{\theta})} \tag{10.4}$$

  This is called **soft clustering** and is identical to generative classifiers except that, here, we never observe $z_i$ (whereas in generative classifier we observe $y_i$, which acts as $z_i$ in training time). We can also do **hard clustering** where we find the MAP solution for above:

$$z_i^* = \arg\max_k \log \mathcal{P}(z_i = k|\mathbf{x}_i, \boldsymbol{\theta}) = \arg\max_k \left(\log \mathcal{P}(z_i = k|\boldsymbol{\theta}) + \log \mathcal{P}(\mathbf{x}_i|z_i = k, \boldsymbol{\theta})\right) \tag{10.5}$$

You can represent each mixture / cluster by its **prototype** or **centroid**. One word of caution is on the selection of $K$ - having too little might not model the data well - having too many would encourage the use of some clusters in artificially splitting some latent factor, which need not be. Adding a little bit of supervision or using informative priors can help a lot in such cases.

- We can also use mixture models for discriminative methods. For instance if you are doing linear regression for house prices in a city against some attribute of the house for instance the location, it might make sense to have different models for different areas. This makes sense because the prices in certain neighborhoods might be better defined by their own models than having a single model to describe the prices in the whole city. In the latter case, a single model might also break when using regularization, because inherently the model would demand a lot of flexibility to explain the data. Hence we can apply a different regression (or classification) function for a different part of the input space. We can model this by allowing the mixing weights and the mixture densities to be input dependent which is represented in Figure 4b:

$$\mathcal{P}(y_i|\mathbf{x}_i, z_i = k, \boldsymbol{\theta}) = \mathcal{N}(y_i|\mathbf{w}_k^\mathsf{T}\mathbf{x}_i, \sigma_k^2) \tag{10.6}$$

$$\mathcal{P}(z_i|\mathbf{x}_i, \boldsymbol{\theta}) = \text{Cat}(z_i|\mathcal{S}(\mathbf{V}^\mathsf{T}\mathbf{x}_i) \tag{10.7}$$

  This is called a **mixture of experts** (MLPP 11.2.4) - i.e. each model is considered an expert in a certain region of the input space. $\mathcal{P}(z_i = k|\mathbf{x}_i, \boldsymbol{\theta})$ is called the **gating function** because it decides what expert to use. The overall prediction can be written as:

$$\mathcal{P}(y_i|\mathbf{x}_i, \boldsymbol{\theta}) = \sum_k \mathcal{P}(y_i|\mathbf{x}_i, z_i = k, \boldsymbol{\theta})\mathcal{P}(z_i = k|\mathbf{x}_i, \boldsymbol{\theta}) \tag{10.8}$$

  Mixture of Experts can also be used for inverse problem where each input might have multiple answers. For instance, consider the location of the end effector of a robot $y$ and the states of its joints $\mathbf{x}$. Each location of the end effector might be produced by multiple joint configurations.

- (MLPP 11.3) When we have complete data and a factored prior, the posterior over the parameters also factorizes, making computations very simple. Unfortunately this is not true if we have missing data/hidden variables. If we look at Figure 4c which is a DGM of an LVM, we can see that $\theta_z$ and $\theta_x$ are dependent, making parameter estimation hard. But, if we observe $z_i$, then by d-separation $\theta_z \perp \theta_x|\mathcal{D}$ and hence the posterior would factorize. When we have not observed $z_i$, it also complicates the computation of MAP and ML estimates. This is because whenever you choose a different $z_i$ we get a different unimodal likelihood. When we marginalize $z_i$ out, we get a multi-modal posterior for $\mathcal{P}(\boldsymbol{\theta}|\mathcal{D})$. These modes correspond to different labeling of the clusters. This leads to parameters which are **unidentifiable** (MLPP 11.3.1) since there is no unique MLE - hence there cannot be a unique MAP estimate - and hence the posterior must be multimodal. Hence, for instance, finding the optimal MLE for a GMM is a NP-hard. The most common work-around is just to approximate a MAP estimate - like in EM algorithm. The reasoning behind it clear from Figure 4c where there are $N$ latent variables, each of which get to see one data point. However there are only two latent parameters each of which gets to see $N$ data points. Hence, the posterior uncertainty about the parameters is much less than the posterior uncertainty over the variables, which justifies the computing of $\mathcal{P}(z_i|\mathbf{x}_i, \hat{\boldsymbol{\theta}})$ while not bothering about $\mathcal{P}(\boldsymbol{\theta}|\mathcal{D})$. It can be proved that the likelihood function in the case when $z_i$ is not observed is non-convex.

- As said before, if we have missing data and/or latent variables, then computing the ML/MAP becomes hard. One way is to apply a gradient-based optimizer to find a local minimum of the NLL: $\ell_-(\boldsymbol{\theta} = -\frac{1}{N}\log\mathcal{P}(\mathcal{D}|\boldsymbol{\theta})$. But, this scheme might be hard if you want to enforce additional constraints. An easier technique is to use **Expectation Maximization** (MLPP 10.4), which is a simple iterative algorithm, often with closed-form updates at each step. It alternates between two steps: (1) **E Step**: infer the missing values given the parameters; (2) **M Step**: optimize the parameters given the "filled in" data. This is essentially exploiting the fact that if the data was fully observed, computing the ML/MAP estimate would be easy.

- (MLPP 11.4.1) EM algorithm defines the complete data log likelihood term as follows, which could be solved if $z_i$ was observed:

$$\ell_c(\boldsymbol{\theta}) \triangleq \sum_{i=1}^{N} \log \mathcal{P}(\mathbf{x}_i, \mathbf{z}_i|\boldsymbol{\theta}) \tag{10.9}$$

This cannot be computed because $\boldsymbol{z}_i$ is unknown. So in this case, the goal of the E step is to compute the **expected sufficient statistics (ESS)** which are terms inside the expected complete data likelihood (on which the MLE depends on):

$$\mathcal{B}(\boldsymbol{\theta}, \boldsymbol{\theta}^{t-1}) = \mathbb{E}\left[\ell_c(\boldsymbol{\theta})|\mathcal{D}, \boldsymbol{\theta}^{t-1}\right] \tag{10.10}$$

where $t$ is the current iteration number. $\mathcal{B}$ is called the **auxiliary function** / the lower bound. The expectation is taken wrt the old parameters, $\boldsymbol{\theta}^{t-1}$, and the observed data $\mathcal{D}$. In the M step we optimize the auxiliary function wrt $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}^t = \arg\max_{\theta} Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{t-1}) \tag{10.11}$$

To perform MAP estimate we maximize after adding the log prior $\log \mathcal{P}(\boldsymbol{\theta})$. The EM algorithm increase the log likelihood of the observed data (plus the log prior, if doing MAP).

- EM for GMM (MLPP 11.4.2): given the number of mixture components $K$ is known. The E step computes the responsibility for each point given a mixture:

$$r_{ik} = \frac{\pi_k \mathcal{P}(\mathbf{x}_i|\boldsymbol{\theta}_k^{t-1})}{\sum_{k'} \pi_{k'} \mathcal{P}(\mathbf{x}_i|\boldsymbol{\theta}_{k'}^{t-1})} \tag{10.12}$$

The M step, we optimize $\mathcal{B}$ wrt $\boldsymbol{\pi}$ and $\boldsymbol{\theta}_k$. We compute, $\boldsymbol{\pi}_k = \frac{1}{N}\sum_i r_{ik}$, which essentially is the weighted number of points assigned to cluster $k$:

$$\boldsymbol{\mu}_k = \frac{\sum_i r_{ik} \boldsymbol{x}_i}{\sum_i r_{ik}} \qquad \boldsymbol{\Sigma}_k = \frac{\sum_i r_{ik} (\boldsymbol{x}_i - \boldsymbol{\mu}_k)(\boldsymbol{x}_i - \boldsymbol{\mu}_k)^\mathsf{T}}{\sum_i r_{ik}} \tag{10.13}$$

After computing the new estimates, we set $\boldsymbol{\theta}^t = (\pi_k, \mu_k, \boldsymbol{\Sigma}_k)$ for $k = 1 : K$ and go to the next E step.

- In GMM EM it often helps convergence by removing the mean and dividing by the standard deviation.

- A popular variant of GMM EM is **k-means** where we suppose that $\boldsymbol{\Sigma}_k = \sigma^2 \boldsymbol{I}_D$ is fixed and $\pi_k = \frac{1}{K}$ is fixed, so the only thing that needs to be updated is $\boldsymbol{\mu}_k \in \mathbb{R}^D$. Here the posterior computed during the E step is:

$$\mathcal{P}(z_i = k|\mathbf{x}_i, \boldsymbol{\theta}) \approx \mathbb{I}(k = z_i^*) \tag{10.14}$$

where $z^* = \arg\max_k \mathcal{P}(z_i = k|\mathbf{x}_i, \boldsymbol{\theta})$. This is sometimes called hard EM because we are doing hard assignments of points to clusters. Since the covariance is spherical, the most probabale cluster can be simply found by:

$$z_i^* = \arg\min_k \|\mathbf{x}_i - \boldsymbol{\mu_k}\|_2^2 \tag{10.15}$$

Given the hard cluster assignments, the M step updates each cluster center by computing the mean of all points assigned to it:

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{i:z_i=k} \mathbf{x}_i \tag{10.16}$$

Since K-means is not a proper EM algorithm, since it is not maximizing likelihood. Instead, it can be interpreted as a greedy algorithm for approximately minimizing a loss function related to data compression (MLPP 11.4.2.6)

- Both k-means and EM needs to be initialized (MLPP 11.4.2.7). It is common to pick $K$ data points at random, and make them as initial cluster centers. The other method that works well is to pick the initial point uniformly at random. Then each subsequent point is picked from the remaining points with probability proportional to its squared distance to the points' closest cluster center.
- MLE might overfit data quite severely. For instance it is possible to get one of the centers a single data point, which will result in likelihood of infinity. This called the collapsing variance problem. An easy solution to this is to perform MAP estimation. When the dimensionality increases, usually MLE falls into degenerate solutions, making MAP estimation the only feasible solution (MLPP 11.4.2.8).
- EM for mixture of experts (MLPP 11.4.3).
- EM can be generalized for DGM with hidden variables (MLPP 11.4.4). In the E step, we just estimate the hidden variables and in the M step we will compute the MLE using these filled-in values. We will suppose that all CPDs are in tabular form. $N_{tck}$ is the number of times node $t$ is in state $k$ while its parents are in state $c$. Where if computing MAP, the mean is:

$$\bar{\theta}_{tck} = \frac{N_{tck} + \alpha_{tck}}{\sum_{k'}(N_{tck'} + \alpha_{tck'})} \tag{10.17}$$

  If it was MLE, the mean would take the same form, except without $\alpha_{tck}$. Hence, each CPT can be written as:

$$\mathcal{P}(x_{it}|\mathbf{x}_{i,\mathsf{pa}(t)}, \boldsymbol{\theta}_t) = \prod_{c=1}^{K_{\mathsf{pa}(t)}} \prod_{k=1}^{K_t} \theta_{tck}^{\mathbb{I}(x_{it}=k,\, \mathbf{x}_{i,\mathsf{pa}(t)}=c)} \tag{10.18}$$

  where $K_{\mathsf{pa}(t)}$ is the number of states the parents of $t$ can be in, and $K_t$ is the number of states node $t$ can be in. Hence the log likelihood takes the form:

$$\log \mathcal{P}(\mathcal{D}|\boldsymbol{\theta}) = \sum_{t=1}^{V} \sum_{c=1}^{K_{\mathsf{pa}(t)}} \sum_{k=1}^{K_t} N_{tck} \log \theta_{tck} \tag{10.19}$$

  Hence, the complete data log likelihood is:

$$\mathbb{E}[\log \mathcal{P}(\mathcal{D}|\boldsymbol{\theta})] = \sum_t \sum_c \sum_k \bar{N}_{tck} \log \theta_{tck} \tag{10.20}$$

$$\bar{N}_{tck} = \sum_{i=1}^{N} \mathbb{E}\left[\mathbb{I}(x_{it} = k, \mathbf{x}_{i,\mathsf{pa}(t)} = c\right] = \sum_i \mathcal{P}(x_{it} = k,\, \mathbf{x}_{i,\mathsf{pa}(t)} = c|\mathcal{D}_i) \tag{10.21}$$

  where $i$ indexes all the $N$ data points. $\mathcal{D}_i$ are the visible variables in case $i$. $\mathcal{P}(x_{it} = k,\, \mathbf{x}_{i,\mathsf{pa}(t)} = c|\mathcal{D}_i)$ is known as the **family marginal** and can be computed using any GM inference method. The $\bar{N}_{tck}$ are the expected sufficient statistics, and this is the output of the E step. Given this ESS, the output o M step is simply (if doing MLE):

$$\hat{\theta}_{tck} = \frac{\bar{N}_{tck}}{\sum_{k'} \bar{N}_{tck'}} \tag{10.22}$$

- The EM algorithm (CVPrince 7.3 and 7.8) is a general purpose tool for fitting $\boldsymbol{\theta}$ in models of the form:

$$\hat{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} \left[\sum_{i=1}^{N} \log \left[\int \mathcal{P}(\mathbf{x}_i, \mathbf{z}_i|\boldsymbol{\theta}) d\mathbf{z}_i\right]\right] = \arg\max_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \tag{10.23}$$

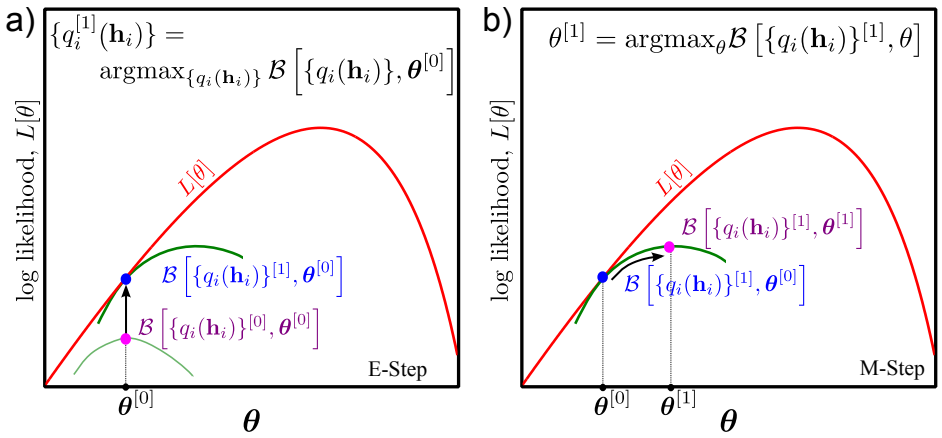



Figure 5: (CVPrince Fig 7.23) showing the steps for EM. The hidden variables $\mathbf{z}_i$ are represented as $\mathbf{h}_i$ here.

The EM algorithm works by defining a lower bound $\mathcal{B}\left[\{q_i(\mathbf{z}_i)\}, \boldsymbol{\theta}\right]$ on the log likelihood above and iteratively increasing this bound. The lower bound is simply a function that is parameterized by and some other quantities and is guaranteed to always return a value that is less than or equal to the log likelihood $L[\boldsymbol{\theta}]$ for any given set of parameters $\boldsymbol{\theta}$ (Figure 5). The lower bound is:

$$\mathcal{B}\left[\{q_i(\mathbf{z}_i)\}, \boldsymbol{\theta}\right] = \sum_{i=1}^{N} \int q_i(\mathbf{z}_i) \log\left[\frac{\mathcal{P}(\mathbf{x}_i, \mathbf{z}_i|\boldsymbol{\theta})}{q_i(\mathbf{z}_i)}\right] d\mathbf{z}_i \leq L[\boldsymbol{\theta}] \tag{10.24}$$

The EM algorithm manipulates both the parameters $\boldsymbol{\theta}$ and the distributions $\{q_i(\mathbf{z}_i)\}_{i=1}^N\}$ to increase the lower bound. It alternates between:

- E step: Updating the probability distribution $\{q_i(\mathbf{z}_i)\}_{i=1}^N\}$ to improve the bound in Equation 10.24.
- M step: Updating the parameters $\boldsymbol{\theta}$ to improve the bound in Equation 10.24.

- You can run EM in an online way. One way to do that is **incremental EM** (MLPP 11.4.8.2). Another way to do it is by **Stepwise EM** (MLPP 11.4.8.3) which is usually faster than incremental and batch EM. In terms of accuracy, stepwise EM is usually as good or sometimes even better than batch EM; incremental EM was often worse than both methods. In Stepwise EM whenever we compute the sufficient statistics $\mathbf{s}_i$ we move $\boldsymbol{\mu}$ toward it. For each iteration the stepsize $\eta_k$ is a function of the step number.

- EM can be coupled with **deterministic annealing** (MLPP 11.4.9) where you smooth the posterior landscape by raising it to a temperature and then gradually cooling it, which slowly moves the posterior to the original shape. In some cases doing exact inference in E step is not possible because of the intractability of computing the exact posterior $\mathcal{P}(\mathbf{z}_i|\mathbf{x}_i, \boldsymbol{\theta}^t)$of the latent variables - here if we do approximate inference in E step it is called **variational EM**, or if we sample from the posteior it is called **Monte Carlo EM**.

- Picking the right number of latent variables which controls the model complexity, for instance in case of mixture models picking $K$, is non trivial (MLPP 11.5). The optimal Bayesian approach would be to pick the model with the largest marginal likelihood $K^* = \arg\max_k \mathcal{P}(\mathcal{D}|K)$. There are two problems with doing that. (1) Evaluating the marginal likelihood for an LVM is quite difficult (possible workarounds are BIC and cross-validation); (2) Searching over a large number of models can be time consuming. One solution is to do stochastic sampling in the space of models, which can quickly determine if a certain value of $K$ is poor, hence does not warrant the exploration of the posterior in the part of model space.

- For non-probabilistic models, one can do model selection (MLPP 10.5.2) by observing the squared reconstruction error - which in the case of k-means is just finding the sum of distances to the closest cluster center

(MSE). Note, that if we were using a probabilistic model, we could have observed the NLL on the test set and found the trough in the u-shaped curve. This is one of the big arguments for using probabilistic models. If you are forced to use a non-probabilistic model and have to choose a $K$, one way is to find a knee in the reconstruction error graph with increasing $K$. The idea is the with $K < K^*$, where $K^*$ is some true number of clusters, the rate of decrease of error would be high since we splitting things that should not be grouped together. When $K > K^*$ the error reduces at a much slower rate because we are artificially breaking natural clusters.

- Fitting models with missing data (MLPP 11.6).

# 11 Latent Linear Models

- (MLPP 12.1) One limitation of mixture model is that only a single latent variable is used to generate observations. In particular, each observation can only come from one of the $K$ prototypes. An alternative is to use a vector of real-valued latent variables $\mathbf{z}_i \in \mathbb{R}^L$. The simplest prior is to use a Gaussian: $\mathcal{P}(\mathbf{z}_i) = \mathcal{N}(\mathbf{z}_i|\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$. If the observations are also continuous, so $\mathbf{x}_i \in \mathbb{R}^D$, we may use a Gaussian for the likelihood. Like in linear regression, we can suppose the mean is a linear function of the latent variables:

$$\mathcal{P}(\mathbf{x}_i|\mathbf{z}_i, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}_i \,|\, \mathbf{W}\mathbf{z}_i + \boldsymbol{\mu}, \boldsymbol{\Psi}) \tag{11.1}$$

  where $\mathbf{W}$ is a $D \times L$ **factor loading matrix**, and $\boldsymbol{\Psi}$ is a $D \times D$ diagonal covariance matrix (since the whole point of the model is to force $\mathbf{z}_i$ to explain the correlation, rather than having it inside the covariance). This model is called **Factor Analysis (FA)**. When $\boldsymbol{\Psi} = \sigma^2\mathbf{I}$, is called **Probabilistic Principal Components Analysis (PPCA)**.

- The induced marginal distribution of FA is a Gaussian:

$$\mathcal{P}(\mathbf{x}_i|\boldsymbol{\theta}) = \int \mathcal{N}(\mathbf{x}_i|\mathbf{W}\mathbf{z}_i + \boldsymbol{\mu}, \boldsymbol{\Psi})\mathcal{N}(\mathbf{z}_i|\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)d\mathbf{z}_i \tag{11.2}$$

$$= \mathcal{N}(\mathbf{x}_i|\mathbf{W}\boldsymbol{\mu}_0 + \boldsymbol{\mu}, \boldsymbol{\Psi} + \mathbf{W}\boldsymbol{\Sigma}_0\mathbf{W}^\mathsf{T}) \tag{11.3}$$

  We can set $\boldsymbol{\mu}_0 = \mathbf{0}$ and absorb it into $\boldsymbol{\mu}$. Similarly, we can set $\boldsymbol{\Sigma}_0 = \mathbf{I}$ and absorb that into a new factor loading matrix $\tilde{\mathbf{W}} = \mathbf{W}\boldsymbol{\Sigma}_0^{-\frac{1}{2}}$. We thus see that FA approximates the covariance matrix of the visible vector using a low-rank decomposition (MLPP 12.1.1).

- (MLPP 12.1.2) The hope FA is that latent factors $\mathbf{z}$ would reveal something interesting about the data. To do this we need to compute the posterior over the latent factors:

$$\mathcal{P}(\mathbf{z}_i|\mathbf{x}_i, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{z}_i|\mathbf{m}_i, \boldsymbol{\Sigma}_i) \tag{11.4}$$

$$\boldsymbol{\Sigma}_i \triangleq \left(\boldsymbol{\Sigma}_0^{-1} + \mathbf{W}^\mathsf{T}\boldsymbol{\Psi}^{-1}\mathbf{W}\right)^{-1} \tag{11.5}$$

$$\mathbf{m}_i \triangleq \boldsymbol{\Sigma}_i\left(\mathbf{W}^\mathsf{T}\boldsymbol{\Psi}^{-1}(\mathbf{x}_i - \boldsymbol{\mu}) + \boldsymbol{\Sigma}_0^{-1}\boldsymbol{\mu}_0\right) \tag{11.6}$$

  Note, that $\boldsymbol{\Sigma}_i$ is independent of $i$ hence can be denoted as $\boldsymbol{\Sigma}$. You can view each point by ploting the **latent factors / latent scores** $\mathbf{m}_i$ in the latent $\mathbb{R}^L$ space.

- The parameters of an FA model are unidentifiable (MLPP 12.1.3). This is because we can not uniquely identify $\mathbf{W}$, and as a corollary the latent factors. Non-identifiability does not affect the predictive performance of the model, however, it does affect the loading matrix, and hence the interpretation of the latent factors. A few solutions to make $\mathbf{W}$ identifiable: (1) force it to be orthonormal, (2) force it to be lower triangular, (3) induce zeros by putting a sparsity promoting priors on the weights.

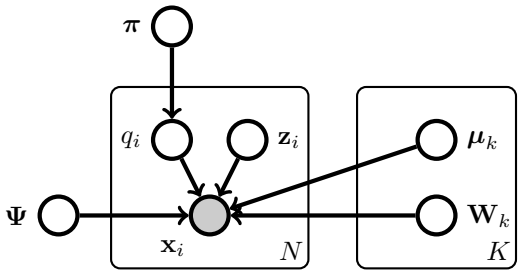



Figure 6: DGM of a Mixture of Factor Analysis (MFA)

- (MLPP 12.1.4) FA model assumes that the data lives on a low dimensional linear manifold. In reality, most data is better modeled by some form of low dimensional curved manifold. One way to approximate it is to use a piecewise linear manifold. Let the $k$'th linear subspace of dimensionality $L_k$ be represented by $\mathbf{W}_k$ for $k = 1 : K$. Suppose we have a latent indicator $q_i \in \{1, \dots, K\}$ specifying which subspace we should use to generate the data. We then sample $\mathbf{z}_i$ from a Gassian prior and pass it through the $\mathbf{W}_k$ matrix (where $k = q_i$) and add noise. More precisely, the model is as follows:

$$\mathcal{P}(\mathbf{x}_i|\mathbf{z}_i, q_i = k, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_k + \mathbf{W}_k\mathbf{z}_i, \boldsymbol{\Psi}) \tag{11.7}$$

$$\mathcal{P}(\mathbf{z}_i|\boldsymbol{\theta}) = \mathcal{N}(\mathbf{z}_i|\mathbf{0}, \mathbf{I}) \tag{11.8}$$

$$\mathcal{P}(\mathbf{q}_i|\boldsymbol{\theta}) = \text{Cat}(q_i|\boldsymbol{\pi}) \tag{11.9}$$

  This is called a **mixture of factor analysers (MFA)**. This is shown in Figure 6.

- (MLPP 12.2) When $\boldsymbol{\Psi} = \sigma^2\mathbf{I}$, and $\mathbf{W}$ to be orthonormal, and when $\sigma^2 \to 0$, this model reduces to **Principal Component Analysis**. When $\sigma^2 > 0$ it is known as **Probabilistic PCA (PPCA)**. Moreover, PPCA does not require $\mathbf{W}$ to be orthogonal.

- (MLPP 12.2.1) Suppose we want to find an orthogonal set of $L$ linear basis vectors $\mathbf{w}_j \in \mathbb{R}^D$ and the corresponding scores $\mathbf{z}_i \in \mathbb{R}^L$, such that we minimize the average reconstruction error:

$$J(\mathbf{W}, \mathbf{Z}) = \frac{1}{N}\sum_{i=1}^{N} \|\mathbf{x}_i - \hat{\mathbf{x}}_i\|^2 \tag{11.10}$$

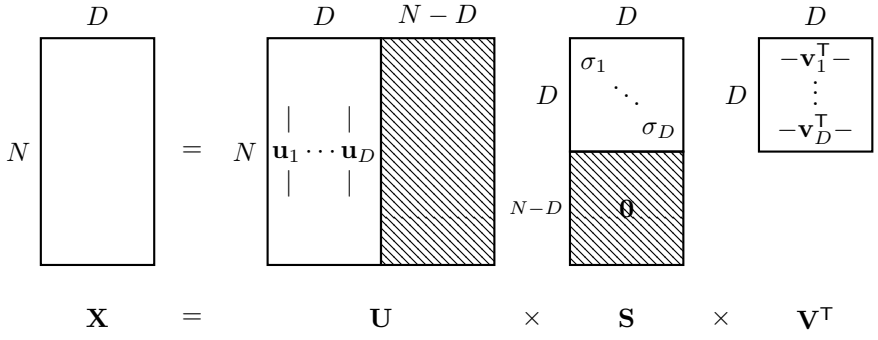
  where $\hat{\mathbf{x}}_i = \mathbf{W}\mathbf{z}$, subject to the constraint that $\mathbf{W}$ is orthonormal. Equivalently we can write the objective as follows:

$$J(\mathbf{W}, \mathbf{Z}) = \|\mathbf{X} - \mathbf{W}\mathbf{Z}^\mathsf{T}\|_F^2 \tag{11.11}$$

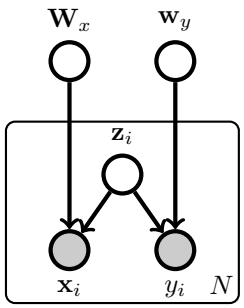
  where Frobenius norm is $\|\mathbf{A}\|_F = \|\mathbf{A}(:)\|_2$. The optimal solution is obtained by setting $\hat{\mathbf{W}} = \mathbf{V}_L$ where $\mathbf{V}_L$ contains the $L$ eigenvectors with largest eigenvalues of the empirical covariance matrix, $\hat{\boldsymbol{\Sigma}} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_i\mathbf{x}_i^\mathsf{T}$ (we assume the $\mathbf{x}_i$ have zero mean). Furthermore, the optimal low-dimensional encoding of the data is given by $\hat{\mathbf{z}}_i = \mathbf{W}^\mathsf{T}\mathbf{x}_i$, which is an orthogonal projection of the data onto the column space spanned by the eigenvectors.

- (MLPP 12.2.1) Since principal directions are the ones which have the maximum variance, you don't want PCA to find directions if the scale of two dimensions is different. It is therefore standard practice to normalize the data before computing PCA.

- You can also use **Singular Value Decomposition (SVD)** (MLPP 12.2.3) for PCA. This basically generalizes the notion of eigenvectors to non-square matrices. The schematic 7a shows the decomposition of matrix $\mathbf{X}$ into $\mathbf{USV}^\mathsf{T}$. $\mathbf{U}$ matrix has orthonormal columns $\mathbf{u_i}$ which are the left singular vectors ($\mathbf{UU}^\mathsf{T} = \mathbf{I}_{N\times N}$). $\mathbf{V}$ matrix has orthonormal rows and columns, where the columns $\mathbf{v}_i$ are the right singular vectors ($\mathbf{V}^\mathsf{T}\mathbf{V} = \mathbf{V}\mathbf{V}^\mathsf{T} = \mathbf{I}_{D\times D}$).




(a) SVD example where $N > D$




(b) Supervised PCA

Figure 7

$\mathbf{S}$ has $\min(N, D)$ singular values $\sigma_i \geq 0$ on the diagonal with the rest of the matrix being 0. We can always drop the lower $(N - D) \times D$ part of $S$ and, hence, the $N \times (N - D)$ left part of $\mathbf{U}$, because they wouldn't contribute to $\mathbf{X}$. This form of SVD takes $\mathcal{O}(ND \min(N, D))$ time to compute. It can be proved that the eigenvectors of $\mathbf{X}^\mathsf{T}\mathbf{X}$ are equal to $\mathbf{V}$, the right singular vectors of $\mathbf{X}$. Also, the eigenvectors of $\mathbf{X}\mathbf{X}^\mathsf{T}$ are equal to $\mathbf{U}$, the left singular vectors of $\mathbf{X}$. The eigenvalues of both $\mathbf{X}^\mathsf{T}\mathbf{X}$ and $\mathbf{X}\mathbf{X}^\mathsf{T}$ are equal to the squared singular values, $\sigma_i^2$.

- SVD can be used for a low rank approximation of $\mathbf{X}$, so that we only use $L < \min(N, D)$ singular values. This curtails all the matrices in the decomposition, hence only picking $L$ left and right singular vectors, giving **truncated SVD**:

$$\underbrace{\mathbf{X}}_{N \times D} \approx \underbrace{\mathbf{U}_L}_{N \times L} \underbrace{\mathbf{S}_L}_{L \times L} \underbrace{\mathbf{V}_L^\mathsf{T}}_{L \times D} \tag{11.12}$$

  From PCA, we know that $\hat{\mathbf{W}} = \mathbf{V}_L$ and $\hat{\mathbf{Z}} = \mathbf{X}\hat{\mathbf{W}} = \mathbf{U}_L\mathbf{S}_L\mathbf{V}_L^\mathsf{T}\mathbf{V}_L = \mathbf{U}_L\mathbf{S}_L$. Moreover, since $\hat{\mathbf{X}} = \hat{\mathbf{Z}}\hat{\mathbf{W}}^\mathsf{T}$, we deduce that $\hat{\mathbf{X}} = \mathbf{U}_L\mathbf{S}_L\mathbf{V}_L^\mathsf{T}$. This shows that PCA is the same as truncated SVD - and shows that PCA is the best low rank approximation to the data.

- More about probabilistic PCA in (MLPP 12.2.4).

- Apart from SVD, we can use EM to fit a PCA model (MLPP 12.2.5). Let $\tilde{\mathbf{Z}}$ be a $L \times N$ matrix storing the posterior means (low-dimensional representations) along its columns. Similarly, $\tilde{\mathbf{X}} = \mathbf{X}^\mathsf{T}$. In the E step we just compute the orthogonal projection of the data:

$$\tilde{\mathbf{Z}} = (\mathbf{W}^\mathsf{T}\mathbf{W})^{-1}\mathbf{W}^\mathsf{T}\tilde{\mathbf{X}} \tag{11.13}$$

  The M step is like linear regression where we replace the observed inputs by the expected values of the latent variables:

$$\mathbf{W} = \tilde{\mathbf{X}}\tilde{\mathbf{Z}}^\mathsf{T}(\tilde{\mathbf{Z}}\tilde{\mathbf{Z}}^\mathsf{T})^{-1} \tag{11.14}$$

  The EM algorithm converges to a solution where $\mathbf{W}$ spans the same linear subspace as that defined by the first $L$ eigenvectors. There is a cool analogy for EM for PCA in the case $D = 2$ and $L = 1$. Consider some point in $\mathbb{R}^2$ attached by springs to a rigid rod, whose orientation is defined by a vector $\mathbf{w}$. Let $\mathbf{z}_i$ be the location where the $i$'th spring attaches to the rod. On the E step, we hold the rod fixed, and let the attachment points slide around so as to minimize the spring energy (which is proportional to the sum of squared residuals). In the M step, we hold the attachment points fixed and let the rod rotate so as to minimize the spring energy. If $N, D \gg L$, EM can be faster than SVD for PCA. Furthermore, advantages usually coupled with EM are transferred to this method too.

- Model selection in FA/PPCA (MLPP 12.3.1) can be performed in a way where we set the model to its maximal size, and then use a technique called **automatic relevancy determination**, combined with EM to automatically prune out irrelevant weights. When the sample size is small, the method automatically prefers simpler models, but as the sample size gets sufficiently large, the method adopts a more complex model.
- Since PCA is not a probabilistic method, you would need to use other techniques for model selection (MLPP 12.3.2). Seeing the reconstruction error is one method which is given by $\hat{\mathbf{x}}_i = \mathbf{W}\mathbf{z}_i + \boldsymbol{\mu}$ where $\mathbf{z}_i = \mathbf{W}^\mathsf{T}(\mathbf{x}_i - \boldsymbol{\mu})$, and $\mathbf{W}$ and $\boldsymbol{\mu}$ are estimated from $\mathcal{D}_{\text{train}}$. But, you would notice that the error keeps going doing with increasing principal components, because PCA is a compression technique (the more latent dimensions you give, the better it will be able to approximate the test data). Another way is to see the retained eigenvalues in decreasing order. Since these techniques don't give a clear signal we need to use an alternate technique. One way to do this is to find a regime change / kink in the error graph. For instance if $\lambda_k$ is some form of error such that $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_{L_{\max}}$ - for instance in PCA this can be eigenvalues. We want to find a threshold $L$, such $\lambda_k \sim \mathcal{N}(\mu_1, \sigma^2)$ if $k \leq L$, and $\lambda_k \sim \mathcal{N}(\mu_2, \sigma^2)$ if $k > L$. Hence we partition the data $L = 1 : L_{\max}$ and compute the MLE. This gives us the **profile log likelihood**:

$$\ell L = \sum_{k=1}^{L} \log \mathcal{N}(\lambda_k | \mu_1(L), \sigma^2(L)) + \sum_{k=L+1}^{K} \log \mathcal{N}(\lambda_k | \mu_2(L), \sigma^2(L)) \tag{11.15}$$

  Finding $L^* = \arg\max \ell L$, gives a good way of choosing the right model.
- You can also do PCA for categorical data with a Gaussian prior, and categorical model conditioned on $\vec{z}_i$ enveloped in a softmax function (MLPP 12.4).
- When you want to combine two sets of related data into a low dimensional embedding, it is an example of **data fusion**. One task is to predict one element of the pair, say $y_i$, from the other one, $\mathbf{x}_i$. The **supervised PCA** (MLPP 12.5.1) model is:

$$\begin{aligned} \mathcal{P}(\mathbf{z}_i) &= \mathcal{N}(\mathbf{0}, \mathbf{I}_L) && (11.16)\\ \mathcal{P}(y_i|\mathbf{z}_i) &= \mathcal{N}(\mathbf{w}_u^\mathsf{T}\mathbf{z}_i + \mu_y, \sigma_y^2) && (11.17)\\ \mathcal{P}(\mathbf{x}_i|\mathbf{z}_i) &= \mathcal{N}(\mathbf{W}_x\mathbf{z}_i + \boldsymbol{\mu}_x, \sigma_x^2 \boldsymbol{I}_D) && (11.18) \end{aligned}$$

  which is shown in Figure 7b. This model is like PCA, except that the target variable $y_i$ is taken into account when learning the low dimensional embedding. Because the model is jointly Gaussian, $\mathcal{P}(y_i|\mathbf{x}_i)$ is also Gaussian. Other techniques in this area are **partial least squares** (MLPP 12.5.2) and **canonical correlation analysis** (MLPP 12.5.3).
- If the goal is to separate many people speaking simultaneously forming a linear combination on a single recording, it is a kind of **blind signal separation** or **blind source separation** (MLPP 12.6), where "blind" means we know nothing about the source of the signals. We can formalize the problem, with $\mathbf{x}_t \in \mathbb{R}^D$ is the observed signal at the sensors at "time" $t$, and $\mathbf{z}_t \in \mathbb{R}^L$ be the vector of source signals:

$$\mathbf{x}_t = \mathbf{W}\mathbf{z}_t + \boldsymbol{\epsilon}_t \tag{11.19}$$

  where $\mathbf{W}$, the mixing matrix, is an $D \times L$ matrix, and $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Psi})$. The goal is to infer the source signals, $\mathcal{P}(\mathbf{z}_t|\mathbf{x}_t, \boldsymbol{\theta})$. Uptil the model looks very similar to FA/PCA. In PCA, though, we suppose each source is independent i.e. we suppose the prior $\mathcal{P}(\mathbf{z}_t)$ is Gaussian. In this model we will relax this assumption, and let the source distributions be any non-Gaussian distribution:

$$\mathcal{P}(\mathbf{z}_t) = \prod_{j=1}^{L} \mathcal{P}_j(z_{tj}) \tag{11.20}$$

Without loss of generality, we can constrain the variance of the source distributions to be 1, because any other variance can be modeled by scaling the rows of $\mathbf{W}$ appropriately. This model is called **Independent Component Analysis (ICA)** (MLPP 12.6). One limitation of ICA is that $\mathbf{W}$ needs to be square, i.e. same number of signal sources as sensors.

- MLE for ICA (MLPP 12.6.1).

# 12 Sparse Linear Models

- Sometimes prediction is possible only when a set of variables are seen in unison (MLPP 13.1). This selection can be done using a model-based approach. If the model is a generalized linear model of the form $\mathcal{P}(y|\mathbf{x}) = \mathcal{P}(y|f(\mathbf{w}^\mathsf{T}\mathbf{x}))$, we can encourage the weight vector $\mathbf{w}$ to be sparse (i.e. mostly zeros). Advantages of such models include: (1) finding relevant features when $N \gg D$, because you don't want to overfit; (2) it includes models where the problem is finding the most useful subset of training examples, which can help reduce overfitting and computational cost (this is known as **sparse kernel machine**; (3) finding a small number of basis functions for sparse representation of a signal.

- Variable selection can also be posed in a Bayesian way (MLPP 13.2). Let $\gamma_j = 1$ when variable $j$ is selected, and $\gamma_j = 0$ when its not. Our goal would be to compute the posterior over models:

$$\mathcal{P}(\boldsymbol{\gamma}|\mathcal{D}) = \frac{e^{-f(\boldsymbol{\gamma})}}{\sum_{\boldsymbol{\gamma}'} e^{-f(\boldsymbol{\gamma}')}} \tag{12.1}$$

  where $f(\gamma)$ is the cost function for selecting a variable: $f(\boldsymbol{\gamma}) \triangleq -\left[\log \mathcal{P}(\mathcal{D}|\boldsymbol{\gamma}) + \log \mathcal{P}(\boldsymbol{\gamma})\right]$
  One way to see if a particular variable is useful is to compute the posterior marginal **inclusion probabilities** $\mathcal{P}(\gamma_j = 1|\mathcal{D})$. From this we can just compute the **median model** by deciding a threshold $\tau$:

$$\hat{\boldsymbol{\gamma}} = \{j \,:\, \mathcal{P}(\gamma_j = 1|\mathcal{D}) > \tau\} \tag{12.2}$$

  Note that when the number of variables are $D$, there are $2^D$ way of choosing your variables (possible models). It will be impossible to compute the full posterior if we are looking a large number of dimensions, or even handling summaries such as the marginal inclusion probabilities.

- The posterior is $\mathcal{P}(\boldsymbol{\gamma}|\mathcal{D}) \propto \mathcal{P}(\boldsymbol{\gamma})\mathcal{P}(\mathcal{D}|\boldsymbol{\gamma})$.
  It is common to use the following prior on the bit vector:

$$\mathcal{P}(\boldsymbol{\gamma}) \;=\; \prod_{j=1}^{D} \mathrm{Ber}(\gamma_j|\pi_0) \;=\; \pi_0^{\|\boldsymbol{\gamma}\|_0}(1-\pi_0)^{D-\|\boldsymbol{\gamma}\|_0} \tag{12.3}$$

$$\log \mathcal{P}(\mathcal{D}|\boldsymbol{\gamma}) \;=\; -\lambda\|\boldsymbol{\gamma}\|_0 + \text{const} \tag{12.4}$$

  where $\pi_0$ is the probability a feature is relevant and $\|\gamma\|_0 = \sum_{j=1}^{D} \gamma_j$ is the $\ell_0$ pseudo-norm which in this case just counts how many variables were selected. The second equation just gives the log prior, where $\lambda \triangleq \frac{1-\pi_0}{\pi_0}$ controls the sparsity of the model. The likelihood can be written as:

$$\mathcal{P}(\mathcal{D}|\boldsymbol{\gamma}) = \mathcal{P}(\mathbf{y}|\mathbf{W}, \boldsymbol{\gamma}) = \int\int \mathcal{P}(\mathbf{y}|\mathbf{X}, \mathbf{w}, \boldsymbol{\gamma})\mathcal{P}(\mathbf{w}|\boldsymbol{\gamma}, \sigma^2)\mathcal{P}(\sigma^2) d\mathbf{w} d\sigma^2 \tag{12.5}$$

  where a reasonable prior is $\mathcal{N}(0, \sigma^2\sigma_w^2)$ where $\sigma_w^2$ controls how big we expect the coefficients to be. The prior $\mathcal{P}(w_j|\sigma^2, \gamma_j)$ is **spike and slab** model (MLPP 13.2.1) because when $\sigma_w^2 \to \infty$ the distribution $\mathcal{P}(w_j|\sigma_j = 1)$ approaches a uniform distribution which can be though of as a slab of constant height. Since the marginal likelihood cannot be computed in closed form, we approximate it using BIC, which becomes:

$$\log \mathcal{P}(\boldsymbol{\gamma}|\mathcal{D}) \approx \log \mathcal{P}(\mathbf{y}|\mathbf{W}, \hat{\mathbf{w}}_{\boldsymbol{\gamma}}, \hat{\sigma}^2) - \frac{\|\boldsymbol{\gamma}\|_0}{2} \log N - \lambda\|\boldsymbol{\gamma}\|_0 + \text{const} \tag{12.6}$$

where $\hat{\mathbf{w}}_{\boldsymbol{\gamma}}$ is the ML or MAP estimate based on $\mathbf{X}_{\boldsymbol{\gamma}}$, the design matrix where we select only the columns of $\mathbf{X}$ where $\gamma_j = 1$. The model has the form $\gamma_j \rightarrow w_j \rightarrow \mathbf{y}$.

- In the **Bernoulli-Gaussian Model** (MLPP 13.2.2) to $\ell_0$ regularization, the model is like $\gamma_j \rightarrow \mathbf{y} \leftarrow w_j$, where unlike the spike-and-slab model we don't integrate out the irrelevant coefficients. Here the cost function will become:

$$f(\mathbf{w}) = \|\mathbf{y} - \mathbf{W}\mathbf{w}\|_2^2 + \lambda\|\mathbf{w}\|_0 \tag{12.7}$$

  This is called $\ell_0$ **regularization**. We have converted the discrete optimization problem (over $\boldsymbol{\gamma} \in \{0,1\}^D$) into a continuous one over $\mathbf{w} \in \mathbb{R}^D$; however, the $\ell_0$ pseudo-norm makes the objective very non-smooth.

- Since the number of model is $2^D$ we definitely cannot fully explore the full posterior. Instead we have to resort to heurestics. All of the methods we will discuss will search through the space of models and evaluating the cost $f(\boldsymbol{\gamma})$ at each point/model. This requires fitting the model (i.e., computing $\arg\max \mathcal{P}(\mathcal{D}|\mathbf{w})$), or evaluating its marginal likelihood (i.e., computing $\int \mathcal{P}(\mathcal{D}|\mathbf{w})\mathcal{P}(\mathbf{w})d\mathbf{w}$) at each step. This is sometimes called the wrapper method, since we "wrap" our search for the best model (or set of good models) around a generic model-fitting procedure. For this to happen we need an efficient way to evaluate the score for a new model $\gamma'$ given a previous model $\gamma$. One way to allow this is to change a single bit at a time. If that is the case there are methods out there to make computing $f(\gamma')$ quicker (MLPP 13.2.3):
  - Use greedy hill climbing, where at each step we define the neighborhood of the current model to be all models that can be reached by a single bit of $\boldsymbol{\gamma}$. This is called **single best replacement**. We can start with an empty set $\boldsymbol{\gamma} = \mathbf{0}$.
  - If we set $\lambda = 0$ in Equation 12.7, so there is no penalty on complexity. Hence we can start with an empty set, and add one feature which best reduces the cost function. This is equivalent to **orthogonal least squares** which in turn is equivalent to greedy **forwards selection**.
  - There are methods based on observing the current residual vector at each step (**orthogonal matching pursuits** or **matching pursuits**).
  - You can also start with the full set i.e. $\boldsymbol{\gamma} = \mathbf{1}$, and then deleting the worst feature. This is called **backward selection**. This is good because variables are removed in the presence of other variables which might be dependent on it. However, this is sort of infeasible for large problems, since the saturated model will be too expensive to fit.

- To approximate the posterior one option is to use MCMC (MLPP 13.2.3.2). The standard approach is to use Metropolis Hastings, where the proposal distribution just flips single bits. This enables the efficient computation of $\mathcal{P}(\boldsymbol{\gamma}'|\mathcal{D})$ given $\mathcal{P}(\boldsymbol{\gamma}|\mathcal{D})$.

- (MLPP 13.3) Part of the computational difficulty in finding the posterior mode of $\mathcal{P}(\boldsymbol{\gamma}, \boldsymbol{\mathcal{D}})$ is that $\gamma_j$ variables are discrete. One solution is to replace them by continuous variables. This can be done by using continuous priors that encourage $w_j = 0$ by putting a lot of probability density near the origin, such as a zero-mean Laplace distribution. If we try to get the MAP estimation, then the penalized log likelihood has the form:

$$f(\mathbf{w}) = -\log \mathcal{P}(\mathcal{D}|\mathbf{w}) - log\mathcal{P}(\mathbf{w}|\lambda) = \text{NLL}(\mathbf{w}) + \lambda\|w\|_1 \tag{12.8}$$

  where $\|\mathbf{w}\|_1 = \sum_{j=1}^D \|w_j\|$ is the $\ell_1$ norm of $\mathbf{w}$, For suitably large $\lambda$ the estimate $\hat{\mathbf{w}}$ will be sparse. Indeed, this can be thought of as a convex approximation to the non-convex $\ell_0$ objective:

$$\arg\min_{\mathbf{w}} \text{NLL}(\mathbf{w}) + \lambda\|\mathbf{w}\|_0 \tag{12.9}$$

  In the case of linear regression, $\ell_1$ objective becomes:

$$f(\mathbf{w}) = \sum_{i=1}^N -\frac{1}{2\sigma^2}(y_i - (w_0 + \mathbf{w}^\mathsf{T}\mathbf{x}_i))^2 + \lambda\|w\|_1 \tag{12.10}$$

$$= \text{RSS}(\mathbf{w}) + \lambda'\|\mathbf{w}\|_1 \tag{12.11}$$

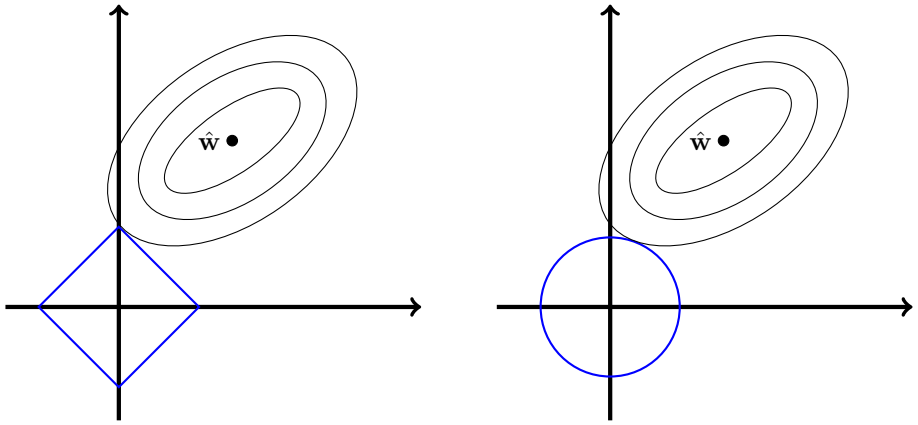



Figure 8: Illustration of $\ell_1$, lasso (left) and $\ell_2$, ridge regression (right) regularization of a least squares problem. The blue square or circle indicates the bound $B$ on the regularizer and the contours indicate the RSS objective function. Each contour gives points where the squared error is the same. The $\ell_1$ corresponds to Laplacian prior whereas $\ell_2$ corresponds to Gaussian prior.

where $\lambda' = 2\lambda\sigma^2$. This method is known as **basis pursuit denoising (BPDN)**. In general, the technique of putting a zero-mean Laplace prior on the parameters and performing MAP estimation is called $\ell_1$ **regularization**.

- (MLPP 13.3.1) BPDN objective is the following non-smooth objective function:

$$\min_{\mathbf{w}} \text{RSS}(\mathbf{w}) + \lambda \|\mathbf{w}\|_1 \tag{12.12}$$

which can be rewritten as constrained but smooth objective

$$\min_{\mathbf{w}} \text{RSS}(\mathbf{w}) \quad \text{s.t.} \quad \|\mathbf{w}\|_1 \leq B \tag{12.13}$$

Here $B$ is an upper bound on the $\ell_1$ norm of the weights: a small (tight) bound $B$ corresponds to a large penalty $\lambda$, and vice versa. Equation 12.13 is also an example of a **quadratic program (QP)**, since we have a quadratic objective subject to linear inequality constraints. This equation is also known as the **lasso** which stands for "least absolute shrinkage and selection operator".

- Similarly we can write ridge regression:

$$\min_{\mathbf{w}} \text{RSS}(\mathbf{w}) + \lambda \|\mathbf{w}\|_2^2 \tag{12.14}$$

which can be rewritten as constrained but smooth objective

$$\min_{\mathbf{w}} \text{RSS}(\mathbf{w}) \quad \text{s.t.} \quad \|\mathbf{w}\|_2^2 \leq B \tag{12.15}$$

See Figure 8 where we plot the RSS objective function and the $\ell_1$ and $\ell_2$ constraint surfaces. We know that the optimal solution occurs at the point where the lowest level set of the objective function intersects the constraint surface. It is clear from the figure that you are more likely to hit a corner in $\ell_1$ which corresponds to sparse solutions, which lie on the coordinate axes.

- The lasso objective is given in Equation 12.12. Unfortunately, the $\|\mathbf{w}\|_1$ term is not differentiable whenever $w_j = 0$ (MLPP 13.3.2). This is an example a non-smooth optimization problem. For this we need to extend the notion of a derivative to handle these non-smooth functions.

- Lasso is a biased estimator, since it selects a subset of a variables, and shrinks all the coefficients by penalizing the absolute values.
- The **regularization path** (MLPP 13.3.4) is the plot of values $\hat{w}_j(\lambda)$ against $\lambda$. As we increase $\lambda$ the solution vector $\hat{\mathbf{w}}(\lambda)$ will tend to get sparser, although not necessarily monotonically. It can be shown that each non-zero coefficient has a piece-wise linear path in the regularization path.
- A downside to using $\ell_1$ regularization to select variables is that it can give quite different results if the data is perturbed slightly (MLPP 13.3.5). The Bayesian approach, which estimates posterior marginal inclusion probabilities $\mathcal{P}(\gamma_j = 1|\mathcal{D})$ is much more robust. A frequentist solution is to use bootstrap re-sampling, and rerun the estimator on different version of the data. By computing how often each variable is selected across different trials we can approximate the posterior inclusion probabilities. This method is known as **stability selection**. We can threshold the stability selection (Bootstrap) inclusion probabilities at some level, say 90% and thus derive a sparse estimator. This is class **bootstrap lasso (blasso)**. It can be proven and is empirically shown that blasso recovers the true model in a wider range of conditions than vanilla lasso.
- $\ell_1$ regularization algorithms:
  - Sometimes its hard to optimize all the variables simultaneously , but it is easy to optimize them one by one. This is the method of **coordinate descent** (MLPP 13.4.1) where we solve for the $j$'th coefficient with all the others held fixed:

    $$w_j^* = \arg\min_z f(\mathbf{w}z\mathbf{e}_j) - f(\mathbf{w}) \tag{12.16}$$

    where $\mathbf{e}_j$ is the $j$'th unit vector. We can cycle through the variable deterministically or via sampling at random.
  - **LARS** (MLPP 13.4.2) works as follows. It starts with a large value of $\lambda$, such that only the variable that is most correlated with the response vector $\mathbf{y}$ is chosen. Then $\lambda$ is decreased until a second variable is found which has the same correlation (in terms of magnitude) with the current residual as the first variable. You can solve for this new value of $\lambda$ analytically. We also allow the removal of variables.
  - **EM for Lasso** (MLPP 13.4.4) which might seem odd because there are no hidden variables. The key point here is that we represent the Laplace distribution as a Gaussian scale mixture (GSM). In the presence of all other algorithms why use EM. EM, unlike other methods, gives a way to compute the full posterior $\mathcal{P}(\mathbf{w}|\mathcal{D})$, rather than just the MAP estimate. This technique is known as **Bayesian Lasso**.
- There are some extensions to $\ell_1$ regularization which are worth noting (MLPP 13.5.1). In the standard case, we assume 1 to 1 correspondence between the parameters and the variables, so that if $\hat{w}_j = 0$, we interpret this to mean that variable $j$ is excluded. But in **multinomial logistic regression** each feature is associated with $C$ different weights, one per class. In **linear regression with categorical inputs** each input is one-hot encoded into a vector of length $C$. In **multi-task learning** we are learning multiple related prediction problems. For instance, we might have $C$ separate regression or binary classification problems. Thus each feature is associated with $C$ separate weights. Here, we may want to use a feature for all or for none of the tasks, and thus select weights at a group level. If we use an $\ell_1$ regularizer of the form $\|\mathbf{w}\| = \sum_j \sum_c |w_{jc}|$, we may end up with some elements of $\mathbf{w}_{j,:}$ being zero and some not. To prevent this kind of situation, we partition the parameter vector into $G$ groups. Now, our objective to minimize is:

  $$J(\mathbf{w}) = \text{NLL}(\mathbf{w}) + \sum_{g=1}^{G} \lambda_g \|\mathbf{w}_g\|_2 \qquad \|\mathbf{w}_g\|_2 = \sqrt{\sum_{j \in g} w_j^2} \tag{12.17}$$

  where $\|\mathbf{w}_g\|_2$ is the 2-norm of the group weight vector. If the NLL is least squares, this method is called **group lasso**. We often would use larger penalty for larger groups which can be enforced by setting $\lambda_g = \lambda\sqrt{d_g}$. Also note if we had used the square of the 2-norm $\|\mathbf{w}_g\|_2^2$ we would have a model equivalent to ridge regression.

By using the square root, we are penalizing the radius of a ball containing the group's weight vector: the only way for the radius to be small is if all elements are small. Thus the square root induces group sparsity. Another way to induce this sparsity would be to use the **infinity-norm**: $\|\mathbf{w}_g\|_\infty = \max_{j \in g} |w_j|$.

- Sometimes you not only want the coefficients to be sparse, but also similar to each in its neighborhood. Binary segmentation is one example. We can use a **fused lasso** in this case (MLPP 13.5.2). We can model it with the following objective:

$$J(\mathbf{w}, \lambda_1, \lambda_2) = \sum_{i=1}^{N} (y_i - w_i)^2 + \underbrace{\lambda_1 \sum_{i=1}^{N} |w_i| + \lambda_2 \sum_{i=1}^{N-1} |w_{i+1} - w_i|}_{\textbf{fused lasso penalty}} \tag{12.18}$$

  You can generalize this idea beyond chain structures to graph structures $G(V, E)$, where the penalty becomes of this form:

$$J(\mathbf{w}, \lambda_1, \lambda_2) = \sum_{s \in V} (y_s - w_s)^2 + \lambda_1 \sum_{s \in V} |w_s| + \lambda_2 \sum_{(s,t) \in E} |w_s - w_t| \tag{12.19}$$

  This is called the **graph-guided fused lasso**.

- Although lasso works, it has couple of problems: (1) If a group of variables are highly correlated, LARS usually selects one variable from them and leaves others in the group out; you can only use group lasso when you know the grouping structure; (2) in $D > N$ case, lasso can select at most $N$ variables before it saturates; (3) if $N > D$ and the variables are correlated, it is empirically shown that prediction accuracy ridge is better than lasso. For these cases **elastic net** (MLPP 13.5.3) works which is a hybrid between lasso and ridge regression. In this model, the objective function is:

$$J(\mathbf{w}, \lambda_1, \lambda_2) = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + \lambda_1 \|\mathbf{w}\|_2^2 + \lambda_2 \|\mathbf{w}\|_1 \tag{12.20}$$

  This function is strictly convex (assuming $\lambda_2 > 0$) so there is a unique global minima even if $\mathbf{X}$ is not full rank. It can be shown that any strictly convex penalty on $\mathbf{w}$ will exhibit the grouping effect (where neighbors have similar weights $w$). For instance, if two features are exactly equal $\mathbf{X}_{:j} = \mathbf{X}_{:k}$, one can show that their estimates are also equal $\hat{w}_j \approx \hat{w}_k$.

- Even though using Laplace priors gives a convex problem, it does not put enough probability mass near 0, so it does not sufficiently suppress noise. Plus it does not put enough probability mass on large values, so it causes shrinkage of relevant coefficients. The solution is to **non-convex** regularizer/priors (MLPP 13.6). Even though we cannot reach the global optimum anymore - it empirically performs better than $\ell_1$ regularization both in terms of predictive accuracy and in detecting relevant variables.

- A natural non-convex generalization of $\ell_1$ regularization is called **bridge regression** ($b > 0$):

$$\hat{\mathbf{w}} = \text{NLL}(\mathbf{w}) + \lambda \sum_j |w_j|^b \tag{12.21}$$

  Unfortunately the objective is not convex of $b < 1$ and it is not sparsity promoting when $b > 1$, so $\ell_1$ is the tightest convex approximation to $\ell_0$ norm.

- Lasso has the problem of giving biased estimates. This because a large value of $\lambda$ not only squashes irrelevant parameters, but also over-penalizes the relevant ones. The way to solve this is to associate a different penalty parameter with each parameter. Of course, it is completely infeasible to tune $D$ parameters by cross-validation, but this poses no problem to the Bayesian. This model is called the **hierarchical adaptive lasso (HAL)** (MLPP 13.6.2). Both the predictive accuracy and selection of variables (setting zeros at right places) is better than lasso. EM for HAL (MLPP 13.6.2.1).

- **Sparse Bayesian Learning (SBL)** or **Automatic Relevance Determination (ARD)** (MLPP 13.7) where we integrate out $\mathbf{w}$ (while simultaneously encouraging sparsity) and maximize the marginal likelihood wrt $\boldsymbol{\tau}$ - which can be combined with basis function expansion in a linear model which gives rise to **relevance vector machine (RVM)**.
- **Sparse Coding** (MLPP 13.8) is about using sparse priors for unsupervised learning. We discussed ICA before which is only different from PCA in using a non-Gaussian prior. If we make this prior to be sparsity promoting, like a Laplace distribution, we will be approximating each observed vector $\mathbf{x}_i$ as a sparse combination of of the basis vectors (columns of $\mathbf{W}$); note that the sparsity pattern (controlled by $\mathbf{z}_i$) changes from data case to data case. If we relax the constraint that $\mathbf{W}$ is orthogonal, we get a method called **sparse coding**. In this context we call the factor loading matrix $\mathbf{W}$ a **dictionary**; each column of which is referred to as an **atom**. In view of the sparse representation, it is common for $L > D$ in the case we call the representation **overcomplete** ($D$ is the dimensionality of the features and $L$ is the number of features). In sparse coding, the dictionary can be fixed or learned. If it is fixed, it is common to use a wavelet or DCT basis, since many natural signals can be well approximated by a small number of such basis functions. However, it is also possible to learn the dictionary by maximizing the likelihood:

$$\log \mathcal{P}(\mathcal{D}|\mathbf{W}) = \sum_{i=1}^{N} \log \int_{\mathbf{z}_i} \mathcal{N}(\mathbf{x}_i|\mathbf{W}\mathbf{z}_i,\, \sigma^2\mathbf{I})\mathcal{P}(\mathbf{z}_i)d\mathbf{z} \tag{12.22}$$

  Again, sparse coding puts sparsity promoting priors on the latent factors $\mathbf{z}_i$. The equation above is hard to maximize - we usually solve the following approximation (MLPP 13.8.1):

$$\log \mathcal{P}(\mathcal{D}|\mathbf{W}) \approx \sum_{i=1}^{N} \max_{\mathbf{z}_i} \left[\log \int_{\mathbf{z}_i} \mathcal{N}(\mathbf{x}_i|\mathbf{W}\mathbf{z}_i,\, \sigma^2\mathbf{I})\mathcal{P}(\mathbf{z}_i)\right] \tag{12.23}$$

  If $\mathcal{P}(\mathbf{z}_i)$ is Laplace, then we can rewrite the NLL as:

$$\text{NLL}(\mathbf{W}, \mathbf{Z}) = \sum_{i=1}^{N} \frac{1}{2}\|\mathbf{x}_i - \mathbf{W}\mathbf{z}_i\|_2^2 + \lambda\|\mathbf{z}_i\|_1 \tag{12.24}$$

  To prevent $\mathbf{W}$ from becoming arbitrarily large, it is common to constrain the $\ell_2$ norm of its columns to be less than or equal to 1: $\mathcal{C} \triangleq \{\mathbf{W} \in \mathbb{R}^{D\times L} \text{ s.t. } \mathbf{w}_j^\mathsf{T}\mathbf{w}_j \leq 1\}$. Then we solve $\min_{\mathbf{W}\in\mathcal{C},\, \mathbf{Z}\in R^{N\times L}} \text{NLL}(\mathbf{W}, \mathbf{Z})$. For optimizing it we alternate between optimizing for $\mathbf{W}$ (called the analysis phase), and optimizing for $\mathbf{Z}$ (called the synthesis phase) - hence the whole procedure is called the **analysis-synthesis loop** (MLPP 13.8.1). For fixed $\mathbf{z}_i$, the optimization over $\mathbf{W}$ is simple least squares problem. For fixed $\mathbf{W}$, the optimization over $\mathbf{Z}$ is identical to the lasso problem.
- A similar model is **non-negative matrix factorization (NMF)** (MLPP 13.8.1):

$$\min_{\mathbf{W}\in\mathcal{C},\, \mathbf{Z}\in R^{N\times L}} \frac{1}{2}\sum_{i=1}^{N} \|\mathbf{x}_i - \mathbf{W}\mathbf{z}_i\|_2^2 \qquad \text{s.t.} \qquad \mathbf{W} \geq 0,\, \mathbf{z}_i \geq 0 \tag{12.25}$$

  Note here we are not tuning any hyper-parameters. The intuition behind this constraint is that the learned dictionary is maybe more interpretable if it is a positive sum of positive parts, rather than sparse sum of atoms that may be positive or negative. Combining NMF with sparse prior on latent factors is called **non-negative sparse coding**.
- We can also impose sparsity constraint on both the factors $\mathbf{z}_i$, and the dictionary $\mathbf{W}$. We call this **sparse matrix factorization** (MLPP 13.8.1). To ensure strict convexity, we can use an elastic net type penalty on the weights:

$$\min_{\mathbf{W},\, \mathbf{Z}} \frac{1}{2}\sum_{i=1}^{N} \|\mathbf{x}_i - \mathbf{W}\mathbf{z}_i\|_2^2 \,+\, \lambda\|\mathbf{z}_i\|_1 \qquad \text{s.t.} \qquad \|\mathbf{w}_j\|_2^2 + \gamma\|\mathbf{w}_j\|_1 \leq 1 \tag{12.26}$$

- Let's say you are given, $\mathbf{y} \in \mathbb{R}^M$ which is the low dimensional projection of $\mathbf{x} \in \mathbb{R}^D$, i.e. $M \ll D$. Let's say we know that $\mathbf{y} = \mathbf{R}\mathbf{x} + \boldsymbol{\epsilon}$, where $\mathbf{R} \in \mathbb{R}^{M \times D}$ is called the **sensing matrix**, and $\boldsymbol{\epsilon}$ is noise term, usually Gaussian. We assume that $\mathbf{R}$ is known, and it corresponds to different linear projections of $\mathbf{x}$. Our goal is to infer $\mathcal{P}(\mathbf{x}|\mathbf{y}, \mathbf{R})$. How can we hope to recover all of $\mathbf{x}$ if we do not measure all of $\mathbf{x}$? The answer is: we can use Bayesian inference with an appropriate prior, that exploits the fact that natural signals can be expressed as a weighted combination of a small number of suitably chosen basis functions. That is, we assume $\mathbf{x} = \mathbf{W}\mathbf{z}$, where $\mathbf{z}$ has a sparse prior, and $\mathbf{W}$ is a suitable dictionary. This is called **compressed sensing** or **compressive sensing** (MLPP 13.8.3).

# 13 Kernels

- When the data cannot be represented by a fixed-size feature vector, what to do? One approach is to assume that you have a way of measuring the similarity between the objects, that doesn't require them to be preprocessed into feature vectors. Let $\kappa(\mathbf{x}, \mathbf{x}') \geq 0$ be some measure of similarity between objects $\mathbf{x}$, $\mathbf{x}' \in \mathcal{X}$, where $\mathcal{X}$ could be any abstract space; we call $\kappa$ a **kernel function** which is real valued function (MLPP 14.1). Usually it is symmetric, $\kappa(\mathbf{x}, \mathbf{x}') = \kappa(\mathbf{x}', \mathbf{x})$, and non-negative. It can be interpreted as a similarity function but it is not a requirement. Some kernels:
  - **squared exponential kernel (SE kernel)** or **Gaussian kernel** (MLPP 14.2.1):

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{x}')^\intercal \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \mathbf{x}')\right) \tag{13.1}$$

    If $\boldsymbol{\Sigma}$ is diagonal we can write:

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{1}{2}\sum_{j=1}^{D}\frac{1}{\sigma_j^2}(x_j - x_j')^2\right) \tag{13.2}$$

    We can interpret $\sigma_j$ as defining the **characteristic length scale** of dimension $j$. If $\sigma_j = \infty$ the corresponding dimension is ignored; hence it is an **ARD kernel** 12. If $\boldsymbol{\Sigma}$ is spherical, we get the isotropic kernel:

$$\kappa(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2}\right) \tag{13.3}$$

    Here $\sigma^2$ is known as the **bandwidth**. Kernel in Equation 13.3 is an example of a **radial basis function (RBF) kernel**, which are functions of $\|\mathbf{x} - \mathbf{x}'\|$.
  - (MLPP 14.2.2) If we have a bag of words representation of two documents $\mathbf{x}_i$ and $\mathbf{x}_i'$ - where $x_{ij}$ tells how many times words $j$ occurs in document $i$. We can use the **cosine similarity** in this case (which is just measuring the cosine angle between two vectors):

$$\kappa(\mathbf{x}_i, \mathbf{x}_i') = \frac{\mathbf{x}_i^\intercal \mathbf{x}_i'}{\|\mathbf{x}_i\|_2 \|\mathbf{x}_i'\|_2} \tag{13.4}$$

    To avoid artificial boosting from words like "the", "and", and when discriminative words are used in bursts, we can replace the word count with a new feature vector called **term frequency inverse document frequency (TF-IDF)**. Term frequency reduces the impact of words that occur many times within one document $\text{tf}(x_{ij}) \triangleq \log(1 + x_{ij})$. Inverse document frequency measures after how many document a word occurs on average, $\text{idf}(j) \triangleq \log \frac{N}{1 + \sum_{i=1}^{N} \mathcal{I}(x_{ij} > 0)}$. Here $N$ is number of total documents. The TF-IDF measure is as follows:

$$\text{tf-idf}(\mathbf{x}_i) \triangleq [\text{tf}(x_{ij}) \times \text{idf}(j)]_{j=1}^{V} \tag{13.5}$$

    $\text{tf-idf}(\mathbf{x}_i)$ can be replace $\mathbf{x}$ in Equation 13.4 to give better results.

– If the **Gram matrix**:

$$\mathbf{K} = \begin{bmatrix} \kappa(\mathbf{x}_1, \mathbf{x}_1) & \cdots & \kappa(\mathbf{x}_1, \mathbf{x}_N) \\ \vdots & \ddots & \vdots \\ \kappa(\mathbf{x}_N, \mathbf{x}_1) & \cdots & \kappa(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix} \tag{13.6}$$

is positive definite for any set of inputs $\{\mathbf{x}_i\}_{i=1}^N$, we call such a kernel $\kappa$ a **mercer kernel** or **positive definite kernel** (MLPP 14.2.3). Both the Gaussian kernel or the cosine kernel are mercer kernels. The use of this comes from eigenvector decomposition of $\mathbf{K} = \mathbf{U}^\mathsf{T}\mathbf{\Lambda}\mathbf{U}$, where $\mathbf{\Lambda}$ is a diagonal matrix of the eigenvalues $\lambda_i > 0$. Considering an element of $\mathbf{K}$ can be computed as $k_{ij} = (\mathbf{\Lambda}^{\frac{1}{2}}\mathbf{U}_{:,i})^\mathsf{T}(\mathbf{\Lambda}^{\frac{1}{2}}\mathbf{U}_{:,j})$, we can write $k_{ij} = \boldsymbol{\phi}(\boldsymbol{x}_i)^\mathsf{T}\boldsymbol{\phi}(\boldsymbol{x}_j)$. Of course, $\boldsymbol{\phi}(\boldsymbol{x}_i) = \mathbf{\Lambda}^{\frac{1}{2}}\mathbf{U}_{:,i}$. Hence, we see that the kernel can be computed as an inner product of some feature vectors that are implicitly defined by the eigenvectors of $\mathbf{K}$. In general if the kernel is Mercer, then there exists a function $\boldsymbol{\phi}$ mapping $vvx \in \mathcal{X} \in \mathbb{R}^D$ such that:

$$\kappa(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}') \tag{13.7}$$

where $\boldsymbol{\phi}$ depends on the eigen functions of $\kappa$ (so $D$ could be potentially an infinite dimensional space). **Polynomial kernel** and **Sigmoid kernel** explained in (MLPP 14.2.3).

– Deriving the feature vector implied by a kernel is in general quite difficult, and only possible if the kernel is Mercer. However, deriving a kernel from a feature vector is easy: we just use

$$\kappa(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}') = \langle \boldsymbol{\phi}(\mathbf{x}),\ \boldsymbol{\phi}(\mathbf{x}') \rangle \tag{13.8}$$

If $\boldsymbol{\phi}(\boldsymbol{x}) = \boldsymbol{x}$, we get the **linear kernel** (MLPP 14.2.4) defined by: $\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{x}^\mathsf{T}\mathbf{x}'$. This is useful when the data is already high dimensional, and if the features individually are informative. In such cases the decision boundary is likely to be representable as a linear combination of the original features, so it is not necessary to work in some other feature space.

– **Matern kernel** (MLPP 14.2.5) are commonly used for Gaussian Process Regression:

$$\kappa(r) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left( \frac{r\sqrt{2\nu}}{\ell} \right)^\nu K_\nu \left( \frac{r\sqrt{2\nu}}{\ell} \right) \tag{13.9}$$

where $r = \|\mathbf{x} - \mathbf{x}'\|$, $\nu > 0$, $\ell > 0$ and $K_\nu$ is modified Bessel function. As $\nu \to \infty$, this approaches the SE kernel. If $\nu = \frac{1}{2}$ the kernel simplifies to:

$$\kappa(r) = \exp(-r/\ell) \tag{13.10}$$

If $D = 1$, and we use this kernel to define a Gaussian process, we get the **Ornstein-Uhlenbeck process**, which describes the velocity of a particle undergoing Brownian motion.

– **String kernels** (MLPP 14.2.6) are the kind which describe the similarity of two strings. If you have two strings $\mathbf{x}$ and $\mathbf{x}'$. Each string is composed of sub-strings $s$, for instance, $\mathbf{x} = usv$. Now let us denote $\phi_s(\mathbf{x})$ be the number of times sub-string $s$ occurs in string $\mathbf{x}$. The kernel is defined as:

$$\kappa(\mathbf{x}, \mathbf{x}') = \sum_{s \in \mathcal{A}^*} w_s \phi(\mathbf{x}) \phi(\mathbf{x}') \tag{13.11}$$

where $w_s \geq 0$ and $\mathcal{A}^*$ is the set of all strings of any length. This is a Mercer kernel and can be computed in $\mathcal{O}(|\mathbf{x}| + |\mathbf{x}'|)$ time, for certain weights $\{w_s\}$. If we set $w_s = 0$ for $|s| > 1$ we get a **bag-of-characters kernel**. If $s$ needs to be surrounded by spaces then we get a **bag-of-words kernel**. If $w_s = 0$ when $|s| \neq k$, we get the **k-spectrum kernel**.

- In computer vision we can generate a bag-of-words representation of an image using some feature descriptor like SIFT - generate SIFT descriptors around some features detected; quantize the feature space (but they maybe variable-sized bags); map each feature set to a multi-resolution histogram; compare using weighted histogram intersection. This is called the **Pyramid Match Kernel** (MLPP 14.2.7). This is a Mercer kernel.
- Suppose we have a probabilistic generative model of feature vectors $\mathcal{P}(\mathbf{x}|\boldsymbol{\theta})$ (MLPP 14.2.8). We can make kernels from this which are good for discriminative tasks. One approach is to use the **Probability Product Kernel**:

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \int \mathcal{P}(\mathbf{x}|\mathbf{x}_i)^\rho \mathcal{P}(\mathbf{x}|\mathbf{x}_j)^\rho d\mathbf{x} \tag{13.12}$$

  where $\rho > 0$ and $\mathcal{P}(\mathbf{x}|\mathbf{x}_i)$ is approximated by $\mathcal{P}(\mathbf{x}|\hat{\boldsymbol{\theta}}(\mathbf{x}_i))$, where $\hat{\boldsymbol{\theta}}(\mathbf{x}_i))$ is a parameter estimate using a single data vector. This makes sense because the fitted model is only being used to see how similar two objects are. So if we fit a model to $\mathbf{x}_i$ and the model thinks $\mathbf{x}_j$ is likely, that means $\mathbf{x}_i$ and $\mathbf{x}_j$ are similar. for instance if $\mathcal{P}(\mathbf{x}|\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\mu}, \sigma^2\mathbf{I})$, where $\sigma^2$ is fixed, $\rho = 1$, and we use $\hat{\boldsymbol{\mu}}(\mathbf{x}_i) = \mathbf{x}_i$ and $\hat{\boldsymbol{\mu}}(\mathbf{x}_j) = \mathbf{x}_j$, then we find:

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{(4\pi\sigma^2)^{D/2}} \exp\left(-\frac{1}{4\sigma^2}\|\mathbf{x}_i - \mathbf{x}_j\|^2\right) \tag{13.13}$$

  where is up to constant factor the RBF kernel. This technique works even if the sequences are of real-valued vectors, unlike the string kernel.
- A more efficient way to use a generative model for a kernel is to use **Fisher kernel**:

$$\kappa(\mathbf{x}, \mathbf{x}') = \mathbf{s}(\mathbf{x})^\mathsf{T} \mathbf{I}^{-1} \mathbf{s}(\mathbf{x}') \tag{13.14}$$

  where $\mathbf{s}$ is the gradient of the log likelihood, or score function, evaluated at the MLE $\hat{\boldsymbol{\theta}}$; and $\mathbf{I}$ is the Fisher information matrix:

$$\mathbf{s}(\mathbf{x}) \triangleq \nabla_{\boldsymbol{\theta}} \log \mathcal{P}(\mathbf{x}|\boldsymbol{\theta}) \,|_{\hat{\boldsymbol{\theta}}}, \qquad \mathbf{I} = -\nabla^2 \log \mathcal{P}(\mathbf{x}|\boldsymbol{\theta}) \,|_{\hat{\boldsymbol{\theta}}} \tag{13.15}$$

  Here $\hat{\boldsymbol{\theta}}$ is a function of all the data, so similarity of $\mathbf{x}$ and $\mathbf{x}'$ is computed in the context of all data as well. Note, that we only have to fit one model. The intuition behind this kernel is that the direction in the parameter space in which $\mathbf{x}$ would like the parameters to move (from $\hat{\boldsymbol{\theta}}$) so as maximize its own likelihood, is similar to the $\mathbf{x}'$'s direction, with respect to the geometry encoded by the curvature of the likelihood function.

- Given all these kernels how can they be used for classification or regression. One way is to use a **kernel machine** (MLPP 14.3.1) which is a GLM where the input is of the form:

$$\boldsymbol{\phi}(\mathbf{x}) = [\kappa(\mathbf{x}, \boldsymbol{\mu}_1), \ldots, \kappa(\mathbf{x}, \boldsymbol{\mu}_K)] \tag{13.16}$$

  where $\boldsymbol{\mu}_k \in \mathcal{X}$ are a set of $K$ centroids. This is called a **kernelized feature vector** - here we don't need to necessarily use Mercer kernels. If $\kappa$ is an RBF kernel, then this is called an RBF network. One way to use this in logistic regression would be to define $\mathcal{P}(y|\mathbf{x}, \boldsymbol{\theta}) = \text{Ber}(\mathbf{w}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}))$. This provides a simple way to define a non-linear decision boundary. For linear regression, you can define $\mathcal{P}(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{w}^\mathsf{T}\boldsymbol{\phi}, \sigma^2)$
- The key problem with **kernel machines** is choosing the centroids $\boldsymbol{\mu}_k$. We cannot uniformly place centroids in the whole $D$ dimensional because of the curse of dimensionality. Another way is to assign a prototype after clustering the space - but the problem is that regions of space that have high density are not necessarily the ones where the prototypes are most useful in representing the output (clustering is unsupervised and might

not be useful for prediction) - plus we need to pick then number of cluster centers. One approach is to make each data point as a prototype:

$$\boldsymbol{\phi}(\mathbf{x}) = [\kappa(\mathbf{x}, \mathbf{x}_1), \ldots, \kappa(\mathbf{x}, \mathbf{x}_N)] \tag{13.17}$$

Now we have $D = N$, hence we have as many parameters as data points. However we can use any of the sparsity promoting priors for $\mathbf{w}$ to efficiently selecting a subset of the training exemplars. We call this **sparse vector machines** (MLPP 14.3.2). We can use $\ell_1$ regularization, and this would be called **$\ell_1$-regularized vector machine (L1VM)**. If we don't want sparsity, we can use $\ell_2$ regularization, and this would be called **$\ell_2$-regularized vector machine (L2VM)**. We can get greater sparsity by using ARD/SBL, resulting in a method called **Relevance Vector Machine (RVM)**. Rather than using a sparsity promoting prior, we can essentially modify the likelihood term (which is un-natural from Bayesian stand-point), and this method is called **Support Vector Machine (SVM)**. RVM are the sparsest of them all, and the fastest to train because you can fit the parameters using empirical Bayes - which only requires fitting a single model.

- Rather than defining our feature vector in terms of kernels, $\boldsymbol{\phi}(\mathbf{x}) = [\kappa(\mathbf{x}, \mathbf{x}_1), \ldots, \kappa(\mathbf{x}, \mathbf{x}_N)]$, we can instead work with the original feature vectors $\mathbf{x}$, but modify the algorithm so that it replaces all inner products of the form $\langle \mathbf{x}, \mathbf{x}' \rangle$ with a call to the kernel function, $\kappa(\mathbf{x}, \mathbf{x}')$. This is called the **kernel trick** (MLPP 14.4). The general idea is that, if we have an algorithm formulated in such a way that the input vector $\mathbf{x}$ enters only in the form of scalar products, then we can replace that scalar product with some other choice of kernel (PRML 6 intro). For instance if we had an algorithm with the term $\langle \mathbf{x}, \mathbf{x}' \rangle^2$, where $\mathbf{x} \in \mathbb{R}^2$, we can apply the kernel trick as follows:

$$\begin{aligned}
\langle \mathbf{x}, \mathbf{x}' \rangle^2 &= (\mathbf{x}_1 \mathbf{x}'_1 + \mathbf{x}_2 \mathbf{x}'_2)^2 \\
&= \mathbf{x}_1^2 \mathbf{x}_1'^2 + \mathbf{x}_2^2 \mathbf{x}_2'^2 + 2\mathbf{x}_1 \mathbf{x}'_1 \mathbf{x}_2 \mathbf{x}'_2 \\
&= \left\langle (\mathbf{x}_1^2,\, \mathbf{x}_2^2,\, \sqrt{2}\mathbf{x}_1\mathbf{x}_2),\, (\mathbf{x}_1'^2,\, \mathbf{x}_2'^2,\, \sqrt{2}\mathbf{x}'_1\mathbf{x}'_2) \right\rangle \qquad (13.18) \\
\kappa(\mathbf{x}, \mathbf{x}') &= \langle \boldsymbol{\phi}(\mathbf{x}),\, \boldsymbol{\phi}(\mathbf{x}') \rangle \qquad (13.19)
\end{aligned}$$

  For this trick to work the kernel needs to be a Mercer kernel.

- In **Kernelized nearest neighbor classification** (MLPP 14.4.1), if you are working with a 1NN classifier, we just need to compute the Euclidean distance of a test vector to all training points, find the closest one, and find its label:

$$\|\mathbf{x}_i - \mathbf{x}_{i'}\|_2^2 = \langle \mathbf{x}_i, \mathbf{x}_i \rangle + \langle \mathbf{x}_{i'}, \mathbf{x}_{i'} \rangle - 2\langle \mathbf{x}_i, \mathbf{x}_{i'} \rangle \tag{13.20}$$

- At times K-means is not appropriate to use Euclidean distance to measure dissimilarity for structured objects. We move to **K-medoids** algorithm where the difference is that each center is a data vector chosen from the data. This helps us to deal with integers rather than objects. For finding centroids we just look for the data point which minimizes the sum of distances to all other points in that cluster:

$$m_k = \underset{i\,:\,z_i = k}{\arg\min} \sum_{i'\,:\,z_{i'} = k,\, i \neq i'} d(i, i') \tag{13.21}$$

  Here $z_i$ is the cluster membership for point $i$. Also $d(i, i') \triangleq \|\mathbf{x}_i - \mathbf{x}_{i'}\|_2^2$. This method can be modified to a classifier, by computing the nearest medoid for each class. This is known as **nearest medoid classification**. The algorithm can be kernelized by using Equation 13.20 to replace the distance $d(i, i')$ (MLPP 14.4.2).

- Given $\mathbf{X}$, the $N \times D$ design matrix, in ridge regression we want to minimize:

$$J(\mathbf{w}) = (\mathbf{y} - \mathbf{X}\mathbf{w})^\mathsf{T}(\mathbf{y} - \mathbf{X}\mathbf{w}) \;+\; \lambda \|\mathbf{w}\|^2 \tag{13.22}$$

We start by using the Gram matrix $\mathbf{K} = \mathbf{X}\mathbf{X}^\mathsf{T}$. If we set $\boldsymbol{\alpha} \triangleq (\mathbf{K} + \lambda \mathbf{I}_N)^{-1}\mathbf{y}$, we can have:

$$\mathbf{w} = \mathbf{X}^\mathsf{T}\boldsymbol{\alpha} = \sum_{i=1}^{N} \alpha_i \mathbf{x}_i \tag{13.23}$$

Hence, this tells us that the solution vector is just a linear sum of $N$ training vectors. We can use this to create **Kernelized ridge regression** (MLPP 14.4.3). When we plug this in at test time to compute the predictive mean, we get:

$$\hat{f}(\mathbf{x}) = \mathbf{w}^\mathsf{T}\mathbf{x} = \sum_{i=1}^{N} \alpha_i \mathbf{x}_i^\mathsf{T}\mathbf{x} = \sum_{i=1}^{N} \alpha_i \kappa(\mathbf{x}, \mathbf{x}_i) \tag{13.24}$$

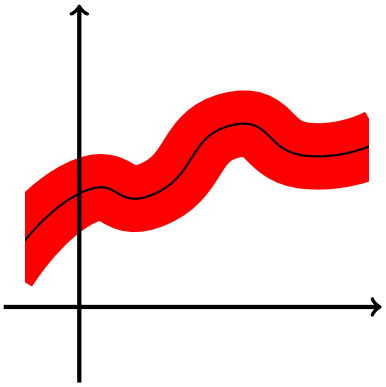
- PCA requires finding the eigenvectors of the sample covariance matrix $\mathbf{S} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i^\mathsf{T} = \frac{1}{N}\mathbf{X}^\mathsf{T}\mathbf{X}$. However, we can also compute PCA by finding the eigenvectors of the inner product matrix $\mathbf{X}\mathbf{X}^\mathsf{T}$. This allows us to produce a nonlinear embedding, using the kernel trick, a method known as **Kernel PCA (kPCA)** (MLPP 14.4.4). Recall that when use $\mathbf{K} = \mathbf{X}\mathbf{X}^\mathsf{T}$, the Mercer's theorem implies some underlying feature space, so we are implicitly replacing $\mathbf{x}_i$ with $\boldsymbol{\phi}(\mathbf{x}_i)$. Given that, now, $\boldsymbol{\Phi}$ is the design matrix, $\mathbf{S}_\phi = \frac{1}{N}\sum_i \boldsymbol{\phi}_i \boldsymbol{\phi}_i^\mathsf{T}$ is the corresponding covariance matrix in the feature space. Whereas linear PCA is limited to using $L \leq D$ components, in kPCA, we can use up to $N$ components, since the rank of $\boldsymbol{\Phi}$ is $N \times D^*$, where $D^*$ is the (potentially infinite) dimensionality of the embedded feature vectors.
- As discussing before one way derive a sparse kernel machine is to use a GLM with kernel basis functions, plus a sparsity-promoting prior. An alternative is to change the objective function from negative log likelihood to some other loss function. In the ridge regression case, we know that the solution to this has the form $\hat{\mathbf{w}} = (\mathbf{X}^\mathsf{T}\mathbf{X} + \lambda \mathbf{I})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}$, and plug-in predictions take the form $\hat{w}_0 + \hat{\mathbf{w}}^\mathsf{T}\mathbf{x}$ (given that $\hat{y}_i = \mathbf{w}^\mathsf{T}\mathbf{x}_i + w_0$). Like in kPCA, we can rewrite these equations in a way that only involves inner products of the form $\mathbf{x}^\mathsf{T}\mathbf{x}'$, which we can replace by calls to a kernel function, $\kappa(\mathbf{x}, \mathbf{x}')$. This is kernelized, but not sparse. If we replace the quadratic / log-loss with some other loss function, we can ensure that the solution is sparse, so that the predictions only depend on a subset of the training data, known as **support vectors** (MLPP 14.5). This combination of the kernel trick plus a modified loss function is known as the **Support Vector Machine (SVM)**.
  - SVMs are un-natural from a probabilistic stand point: (1) They encode sparsity in the loss function rather than in the prior; (2) they encode kernels by using an algorithmic trick rather than being an explicit part of the model; (3) SVMs do not give probabilistic outputs.
  - **SVMs for Regression** (MLPP 14.5.1): One problem with kernelized ridge regression is that the solution vector $\mathbf{w}$ depends on all the training inputs. SVM can make a more sparse estimate. (Vapnik 1997) proposed a variant of huber loss function which is called **epsilon insensitive loss function**:

    $$L_\epsilon(y, \hat{y}) \triangleq \begin{cases} 0 & \text{if } |y - \hat{y}| < \epsilon \\ |y - \hat{y}| - \epsilon & \text{otherwise} \end{cases} \tag{13.25}$$

    This means that any prediction inside the $\epsilon$-tube around the true value will not be penalized. In Figure 9a points in blue are penalized and points in yellow are not. The corresponding objective function is usually written in the following form:

    $$J(\mathbf{w}) = C\sum_{i=1}^{N} L_\epsilon(y_i, \hat{y}_i) + \frac{1}{2}\|\mathbf{w}\|^2 \tag{13.26}$$

    where $\hat{y}_i = f(\mathbf{x}_i) = \mathbf{w}^\mathsf{T}\mathbf{x}_i + w_0$ and $C = 1/\lambda$ is a regularization constant. The objective is convex but not differentiable, because of the absolute value function in the loss term. Nevertheless it can be



(a) $\epsilon$ loss

Figure 9

solved by standard quadratic program. The optimal solution has the form $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i$ where $\alpha_i \geq 0$. Furthermore, it turns out the $\boldsymbol{\alpha}$ vector is sparse, because we don't care about errors which are within the $\epsilon$ margin. The $\mathbf{x}_i$ for which $\alpha_i > 0$ are called **support vectors**; these are the points for which the errors lie on or outside the tube. Once the model is trained we can make predictions by:

$$\begin{aligned}\hat{y}(\mathbf{x} &= \hat{w}_0 + \hat{\mathbf{w}}^\mathsf{T}\mathbf{x} \\ &= \hat{w}_0 + \sum_i \alpha_i \mathbf{x}_i^\mathsf{T}\mathbf{x} \\ &= \hat{w}_0 + \sum_i \alpha_i \kappa(\mathbf{x}_i, \mathbf{x})\end{aligned} \tag{13.27}$$

– **SVMs for Classification** (MLPP 14.5.2): We use a **hinge loss** where, like before, $\eta = f(\mathbf{x}) = \mathbf{w}^\mathsf{T}\mathbf{x} + w_0$ and the labels are $y \in \{-1, +1\}$:

$$L_{\text{hinge}}(y, \eta) = \max(0, 1 - y\eta) = (1 - y\eta)_+ \tag{13.28}$$

Here $\eta = f(\mathbf{x})$ is our confidence in choosing label $y = 1$; however, it need not have any probabilistic meaning. The overall objective looks like:

$$\min_{\mathbf{w}, w_0} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N}(1 - y_i f(\mathbf{x}_i))_+ \tag{13.29}$$

This can be solved via a quadratic program. One can show the solution is of the form $\hat{\mathbf{w}} = \sum_i \alpha_i \mathbf{x}_i$ where $\alpha_i = \lambda_i y_i$ and where $\boldsymbol{\alpha}$ is sparse because of the hinge loss. The $\mathbf{x}_i$ for which $\alpha_i > 0$ are called support vectors; these are the points which have been incorrectly classified or are classified correctly but are on the inside of the margin. At test time, prediction is done using:

$$\hat{y}(\mathbf{x}) = \text{sgn}\left(\hat{w}_0 + \hat{\mathbf{w}}^\mathsf{T}\mathbf{x}\right) = \text{sgn}\left(\hat{w}_0 + \sum_{i=1}^{N} \alpha_i \kappa(\mathbf{x}_i, \mathbf{x})\right) \tag{13.30}$$

This takes $\mathcal{O}(sD)$ time to compute, where $s \leq N$ is the number of support vectors. This depends on the sparsity level and hence on the regularizer $C$.

– Consider a point $\mathbf{x}$ in the induced space: $\mathbf{x} = \mathbf{x}_\perp + r\frac{\mathbf{w}}{\|\mathbf{w}\|}$ where $r$ is the distance of $\mathbf{x}$ from the decision boundary whose normal vector is $\mathbf{w}$ and $\mathbf{x}_\perp$ is the orthogonal projection of $\mathbf{x}$ onto this boundary. Hence:

$$f(\mathbf{x}) \;=\; \mathbf{w}^\mathsf{T}\mathbf{x} + w_0 \;=\; (\mathbf{w}^\mathsf{T}\mathbf{x}_\perp + w_0) + r\frac{\mathbf{w}^\mathsf{T}\mathbf{w}}{\|\mathbf{w}\|} \;=\; (\mathbf{w}^\mathsf{T}\mathbf{x}_\perp + w_0) + r\|\mathbf{w}\| \tag{13.31}$$

Since $f(\mathbf{x}_\perp) = 0$, so $f(\mathbf{x}) = r\|\mathbf{w}\|$ and $r = \frac{f(\mathbf{x})}{\|\mathbf{w}\|}$. We would like $r$ to be as large as possible. There might be many lines that separate the training data (especially in higher dimensions), but intuitively the best one to pick would be the one that maximizes the margin. In addition, to ensure each point is on the right side of the boundary we say $f(\mathbf{x}_i y_i > 0$. So our objective is $\max_{\mathbf{w},w_0} \min_{i=1}^N \frac{y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i + w_0)}{\|\mathbf{w}\|}$. Since there is a scale ambiguity, we can say $y_i f(\mathbf{x}_i) = 1$ for the point that is closest to the decision boundary. Hence $y_i f(\mathbf{x}_i) \geq 1$ for all $i$'s. Thus the objective becomes:

$$\min_{\mathbf{w},w_0} \frac{1}{2}\|\mathbf{w}\|^2 \qquad \text{s.t.} \qquad y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i + w_0) \geq 1, \quad i = 1 \,:\, N \tag{13.32}$$

This constraint is makes SVM a **large margin classifier** (MLPP 14.5.2.2). Now, if the data is not linearly separable (even after the kernel trick), there will be no feasible solution with the constraint $y_i f(\mathbf{x}_i) \geq 1$ for all $i$'s. We introduce a slack variable $\xi_i \geq 0$, which would be (1) $\xi_i = 0$ if the point is on or across the correct margin boundary; (2) $0 < \xi_i \leq 1$ the point lies inside the margin even though its on the correct side of the boundary; (3) $\xi_i > 1$ if the point is on the wrong side of the boundary. We put these soft margin constraints in the objective, transforming it to:

$$\min_{\mathbf{w},w_0} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^N \xi_i \qquad \text{s.t.} \qquad \xi_i \geq 0, \quad y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i + w_0) \geq 1 - \xi_i, \quad i = 1 \,:\, N \tag{13.33}$$

Optimizing this is equivalent to minimizing the objective in Equation 13.29. The parameter $C$ is a regularization parameter that controls the errors we are willing to tolerate in the training set. We can set it to $C = 1/(\nu N)$ where $0 < \nu \leq 1$ controls the fraction of mis-classified points that we allow during the training phase. This is class a **$\nu$-SVM classifier**, where $\nu$ can be set from cross-validation.

– An SVM classifier gives hard-labeling $\hat{y}(\mathbf{x}) = \text{sign}(f(\mathbf{x}))$. However, we often want ta measure of confidence in our prediction (MLPP 14.5.2.3). One heuristic approach is to interpret $f(\mathbf{x})$ as the log-odds ratio, $\log \frac{\mathcal{P}(y=1|\mathbf{x})}{\mathcal{P}(y=0|\mathbf{x})}$. We can then convert the output of an SVM to a probability using: $\mathcal{P}(y = 1|\mathbf{x}, \boldsymbol{\theta}) = \text{sigm}(af(\mathbf{x}) + b)$, where $a$, $b$ can be estimated by maximum likelihood on a separate validation set. However, the resulting probabilities are not particularly well calibrated, since there is nothing in the SVM training process that justifies interpreting $f(\mathbf{x})$ as a log-odds ratio.

– **SVM for multi-class classification** (MLPP 14.5.2.4): This can be hard because the output of the classifier is not on a calibrated scale, and would be difficult to compare to each other. When we train $C$ binary classifiers for **one-vs-all**, where the output of each classifier is $f_c(\mathbf{x})$ (being +ve for class $c$), but this results in regions which are ambiguously labeled. A common alternative i to pick $\hat{y}(\mathbf{x}) = \arg\max_c f_c(\mathbf{x})$, but this technique doesn't work either because the values might not be comparable across classifiers. Also there is a problem of **class imbalance** because when training one-vs-all classifiers, usually you have much more samples in the negative case. Another approach is **one-vs-one** (also called **all pairs**), in which we train $C(C-1)/2$ classifiers to discriminate all pairs $f_{c,c'}$. We then classify a point into the class which has the highest number of votes. However this approach can also result in ambiguities. Most of these problems are a direct result of having no probabilistic interpretation of the SVM output - an alternative is to use a kernel inside multi-class classifier like RVM or L1VM.

– Choosing the right $C$ for SVM is done via cross-validation (MLPP 14.5.3). If you are using RBF kernel, the kernel parameter $\gamma = \frac{1}{2\sigma^2}$ is tightly coupled with $C$ in the sense that if you want to use a narrow kernel, we need heavy regularization, hence a small $C$.

  - In summary SVMs use three ingredients: the kernel trick, sparsity, and the large margin principle (MLPP 14.5.4). The kernel trick is necessary to avoid underfitting (since you can better fit the model sometimes in a higher dimensional space). Whereas sparsity and large margin principle avoid overfitting (and both arise from the use of the hinge loss function).
  - Probabilistic interpretation of the hinge loss can be achieved by using a pseudo-likelihood which can be represented as a Gaussian scale mixture (MLPP 14.5.5).

- Comparisons of different discriminative kernel methods (L1VM, L2VM, RVM, SVM, Guassian Processes (GPs)) (MLPP 14.6): (1) One question is how does each method compare on the objective $J(\mathbf{w}) = -\log \mathcal{P}(\mathcal{D}|\mathbf{w}) - \log \mathcal{P}(\mathbf{w})$. For L2VM, L1VM, and SVM this objective is convex - for RVM it is not. GPs are Bayesian methods that do not perform any parameter estimation. (2) For optimizing the kernel parameters (such as the bandwidth of the RBF kernel, and the level of regularization) Gaussian prior method like L2VM, RVM and GPs use gradient based optimizers to maximize the marginal likelihood; for SVM and L1VM we use the slower cross-validation method; (3) L1VM, RVM, and SVM are all spase kernel methods, and GPs and L2VM use all training examples i.e. they are not sparse. The advantage of sparsity is shorter test time and sometimes improved accuracy. (4) All methods give a probabilistic output except SVM. (5) All methods except SVMs are naturally multiclass, by using a multinoulli output instead of Bernoulli. (6) SVMs and GPs are the only ones to require a Mercer kernel. Conclusion: If speed matters use an RVM, but if well-calibrated probabilistic output matters (for instance in active learning or control problems) use a GP. SVMs are sensible for structured output case, where likelihood based methods can be slow.

- A different kind of kernel known as a smoothing kernel can be used for unsupervised non-parametric density estimation $\mathcal{P}(\mathbf{x})$, as well as creating generative classifiers for classification and regression by making models of the form $\mathcal{P}(y, \mathbf{x})$ (MLPP 14.7). A **smoothing kernel** (MLPP 14.7.1) satisfies these properties:

$$\int \kappa(x)dx = 1, \quad \int x\kappa(x)dx = 0, \quad \int x^2\kappa(x)dx > 0 \tag{13.34}$$

  A simple example is a **Gaussian kernel**: $\kappa(x) \triangleq \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$. We control the width of the kernel by a bandwidth parameter $h$: $\kappa_h(x) \triangleq \frac{1}{h}\kappa\left(\frac{x}{h}\right)$. This can be generalized to vector values as $\kappa_h(\mathbf{x}) = \kappa_h(\|h\|)$. In the case of the Gaussian kernel this becomes:

$$\kappa_h(\mathbf{x}) = \frac{1}{h^D(2\pi)^{D/2}} \prod_{j=1}^{D} \exp\left(-\frac{x_j^2}{2h^2}\right) \tag{13.35}$$

  Gaussian kernels have unbounded support. As an alternative with compact support is **Epanechnikov kernel**: $\kappa(x) \triangleq \frac{3}{4}(1-x^2)\mathbb{I}(|x| \leq 1)$. Compact support is important for efficiency reasons, since one can use fast NN methods to evaluate the density. Unlike Epanechnikov kernel, **tri-cube kernel** has two continuous derivative at support boundary: $\kappa(x) \triangleq \frac{70}{81}(1-|x|^3)^3\mathbb{I}(|x| \leq 1)$. We also have the **boxcar kernel**: $\kappa(x) \triangleq \mathbb{I}(|x| \leq 1)$.

- Unlike parametric density estimator (like GMM), in **Kernel density estimation (KDE)** (MLPP 14.7.2) you don't need to specify $K$. All you need to do is allocate one cluster per data point, so $\boldsymbol{\mu}_i = \mathbf{x}_i$. In this case the model becomes:

$$\mathcal{P}(\mathbf{x}|\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N} \kappa_h(\mathbf{x} - \mathbf{x}_i) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{N}(\mathbf{x}|\mathbf{x}_i, \sigma^2\mathbf{I}) \tag{13.36}$$

  This is called **Parzen window density estimator** or **kernel density estimator (KDE)**, and is simple non-parametric density model. The advantage is that there is no model fitting required (except the bandwidth which can be set by cross-validation). The disadvantage is that model takes a lot of memory to store and lot of time to evaluate. When using a boxcar kernel, it is like counting how many data points land within an interval of size $h$ around $x_i$. The usual way to pick $h$ is to minimize an estimate (such as cross-validation) of the frequentist risk.

- KDE for KNN (MLPP 14.7.3): Instead of fixing a bandwidth $h$, we allow the bandwidth or volume to be different for each data point. Specifically we will grow a volume around $\mathbf{x}$ until we encounter $K$ data points, regardless of their class label. Given $V(\mathbf{x})$ is volume required, $N_c(\mathbf{x})$ be examples of class $c$ in that volumes, and $N_c$ be the total number of examples of class $c$, the likelihood becomes:

$$\mathcal{P}(\mathbf{x}|y=c,\mathcal{D}) = \frac{N_c(\mathbf{x})}{N_c V(\mathbf{x})} \tag{13.37}$$

The class posterior can be computed to:

$$\mathcal{P}(y=c|\mathbf{x},\mathcal{D}) = \frac{N_c(\mathbf{x})}{K} \tag{13.38}$$

- **Kernel Regression** (MLPP 14.7.4), where the goal is to compute the conditional expectation:

$$f(\mathbf{x}) = \mathbb{E}[y|\mathbf{x}] = \int y\mathcal{P}(y|\mathbf{x})dy = \frac{\int y\mathcal{P}(\mathbf{x},y)dy}{\int \mathcal{P}(\mathbf{x},y)dy} \tag{13.39}$$

We can use KDE to approximate the joint density $\mathcal{P}(\mathbf{x},y)$ as follows:

$$\mathcal{P}(x,y) \approx \frac{1}{N}\sum_{i=1}^{N} \kappa_h(\mathbf{x}-\mathbf{x}_i)\kappa_h(\mathbf{y}-\mathbf{y}_i) \tag{13.40}$$

We can use this to rewrite the conditional expectation, and the method is called **kernel regression / kernel smoothing / Nadaraya-Watson model** (MLPP 14.7.4):

$$f(\mathbf{x}) = \sum_{i=1}^{N} w_i(\mathbf{x})y_i \qquad w_i(\mathbf{x}) \triangleq \frac{\kappa_h(\mathbf{x}-\mathbf{x}_i)}{\sum_{i'=1}^{N}\kappa_h(\mathbf{x}-\mathbf{x}_{i'})} \tag{13.41}$$

- **Locally weighted regression** (MLPP 14.7.5).

# 14 Gaussian Processes

- In supervised learning, we observe some inputs $\mathbf{x}_i$ and some outputs $y_i$. We assume that $y_i = f(\mathbf{x}_i)$ for some unknown function $f$, possibly corrupted by noise. Up until now we have focused on parametric representations for the function $f$ so that instead of inferring $\mathcal{P}(f|\mathbf{X},\mathbf{y})$, we infer $\mathcal{P}(\boldsymbol{\theta}|\mathbf{X},\mathbf{y})$. **Gaussian processes** (MLPP 15.1) is a way to perform Bayesian inference over functions themselves. Gaussian process defines a prior over functions, which can be converted to a posterior over functions after observing some data. Although it might seem difficult to define a distribution over a function, it turns out it is just a matter of defining a distribution over the function's values at a finite, but arbitrary set of points, say $\mathbf{x}_1,\ldots,\mathbf{x}_N$. A GP assumes that $\mathcal{P}(f(\mathbf{x}_1),\ldots,f(\mathbf{x}_N))$ is jointly Gaussian, with some mean $\boldsymbol{\mu}(\mathbf{x})$ and covariance $\boldsymbol{\Sigma}(\mathbf{x})$ given by $\Sigma_{ij} = \kappa(\mathbf{x}_i,\mathbf{x}_j)$, where $\kappa$ is a positive definite kernel function. The key idea is that if the kernel thinks that $\mathbf{x}_i$ and $\mathbf{x}_j$ are similar, then we expect the output of the function at those points to be similar too. A Gaussian process is a generalization of the Gaussian probability distribution. Whereas a probability distribution describes random variables which are scalars or vectors (for multivariate distributions), a stochastic process governs the properties of functions (Rasmussen and Williams 2006 [1]). If you ask only for the properties of the function at a finite number of points, then inference in the Gaussian process will give you the same answer if you ignore the infinitely many other points, as if you would have taken them all into account! The key advantage of GP over L1VM, RVM, and SVM is well calibrated probabilistic output.

[1] `http://www.gaussianprocess.org/gpml/chapters/`

- A gaussian process is a collection of random variables, any finite number of which have a joint Gaussian distribution. A GP is completely specified by its mean function $m(\mathbf{x})$ and covariance function $\kappa(\mathbf{x}, \mathbf{x}')$, where we consider a real process $f(\mathbf{x})$:

$$m(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})] \tag{14.1}$$
$$\kappa(\mathbf{x}, \mathbf{x}') = \mathbb{E}\left[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x}') - m(\mathbf{x}'))^{\mathsf{T}}\right] \tag{14.2}$$

  and we will write the Gaussian Process as:

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}),\, \kappa(\mathbf{x}, \mathbf{x}')) \tag{14.3}$$

  Usually for notational simplicity we will take the mean function to be 0. In our case the random variables represent the value of the function $f(\mathbf{x})$ at location $\mathbf{x}$. We require that $\kappa$ should be positive definite kernel. For any finite set of points, this process defines a joint Gaussian:

$$\mathcal{P}(\mathbf{f}|\mathbf{X}) = \mathcal{N}(\mathbf{f}|\boldsymbol{\mu}, \mathbf{K}) \tag{14.4}$$

  where $K_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$ and $\boldsymbol{\mu} = (m(\mathbf{x}_1), \ldots, m(\mathbf{x}_N))$.

- Suppose we observe training data set $\mathcal{D} = \{(x_i, f_i),\, i = 1 : N\}$ where $f_i = f(x_i)$ is the noise free observation of the function evaluated at $\mathbf{x}_i$ Given a test set $\mathbf{X}_*$ of size $N_* \times D$, we want to predict the function outputs $\mathbf{f}_*$. If we ask GP to predict $f(\mathbf{x})$ for a value it has already seen, we want the GP to return the answer $f(\mathbf{x})$ with no uncertainty. In other words, it should act as an interpolator of the training data. This will only happen if we assume the observations are noiseless (MLPP 15.2.1). By definition of the GP, the joint distribution has the following form:

$$\begin{pmatrix} \mathbf{f} \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \boldsymbol{\mu} \\ \boldsymbol{\mu}_* \end{pmatrix}, \begin{pmatrix} \mathbf{K} & \mathbf{K}_* \\ \mathbf{K}_*^{\mathsf{T}} & \mathbf{K}_{**} \end{pmatrix}\right) \tag{14.5}$$

  where $\mathbf{K} = \kappa(\mathbf{X}, \mathbf{X})$ is $N \times N$, $\mathbf{K}_* = \kappa(\mathbf{X}, \mathbf{X}_*)$ is $N \times N_*$, and $\mathbf{K}_{**} = \kappa(\mathbf{X}_*, \mathbf{X}_*)$ is $N_* \times N_*$. By rules of conditioning of a Gaussian, the posterior has the following form:

$$\mathcal{P}(\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{f}) = \mathcal{N}(\mathbf{f}_*|\boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*) \tag{14.6}$$
$$\boldsymbol{\mu}_* = \boldsymbol{\mu}(\mathbf{X}_*) + \mathbf{K}_*^{\mathsf{T}}\mathbf{K}^{-1}(\mathbf{f} - \boldsymbol{\mu}(\mathbf{X})) \tag{14.7}$$
$$\boldsymbol{\Sigma}_* = \mathbf{K}_{**} - \mathbf{K}_*^{\mathsf{T}}\mathbf{K}^{-1}\mathbf{K}_* \tag{14.8}$$

  The model would perfectly interpolate the training data, and the predictive uncertainty increases as we move further away from the observed data.

- Predictions using noisy observations (MLPP 15.2.2). Effect of kernel parameters (MLPP 15.2.3).

- Rather than enumerating kernel parameters on a grid and then evaluating them one by one, we can use empirical Bayes approach, which will allow us to use continuous optimization approaches in order to maximize the marginal likelihood (MLPP 15.2.4):

$$\mathcal{P}(\mathbf{y}|\mathbf{X}) = \int \mathcal{P}(\mathbf{y}|\mathbf{f}, \mathbf{X})\mathcal{P}(\mathbf{f}|\mathbf{X})d\mathbf{f} \tag{14.9}$$

  You can also compute the posterior of the hyper-parameters rather than just a point estimate (MLPP 15.2.4.2). Another approach is **multiple kernel learning** (MLPP 15.2.4.3), where rather than learning the kernel parameters, we consider multiple base kernels where the final kernel is a weighted sum of the base kernels: $\kappa(\mathbf{x}, \mathbf{x}') = \sum_j w_j \kappa_j(\mathbf{x}, \mathbf{x}')$. Then the learning task is to optimize the weights $\mathbf{w}$.

- If **GPs in the GLM setting** (MLPP 15.3), we can employ GPs for classification. As with Bayesian logistic regression, the main difficulty is that the Gaussian prior is not conjugate to the bernoulli/multinoulli likelihood. There are several approximations one can adopt, but the simplest and fastest is the Gaussian approximation (MLPP 15.3). For binary classification, at convergence the Gaussian approximation of the posterior takes the form:

$$\mathcal{P}(\mathbf{f}|\mathbf{X},\mathbf{y}) \approx \mathcal{N}\left(\hat{\mathbf{f}},\, (\mathbf{K}^{-1}+\mathbf{W})^{-1}\right) \tag{14.10}$$

  where $\mathbf{W} \triangleq -\nabla^2 \log \mathcal{P}(\mathbf{y}|\mathbf{f})$. The posterior predictive (MLPP 15.3.1.2) takes the form:

$$\mathcal{P}(f_*|\mathbf{x}_*,\mathbf{X},\mathbf{y}) = \mathcal{N}(\mathbb{E}[f_*], \mathrm{var}[f_*]) \tag{14.11}$$

  To convert this into a predictive distribution for binary responses, we use:

$$\pi_* = \mathcal{P}(y_*=1|\mathbf{x}_*,\mathbf{X},\mathbf{y}) \approx \int \mathrm{sigm}(f_*)\mathcal{P}(f_*|\mathbf{x}_*,\mathbf{X},\mathbf{y})df_* \tag{14.12}$$

- GPs for multi-class classification (MLPP 15.3.2). GPs for Poisson regression (MLPP 15.3.3) where the Matern kernel is used.
- Bayesian linear regression is equivalent to a GP with covariance function $\kappa(\mathbf{x},\mathbf{x}') = \mathbf{x}^\mathsf{T}\boldsymbol{\Sigma}\mathbf{x}'$ (MLPP 15.4.1).
- **SVM comparison to GPs** (MLPP 15.4.3): Note that SVM has the following objective function for binary classification:

$$J(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{N}(1-y_i f_i)_+ \tag{14.13}$$

  where $f_i \triangleq f(\mathbf{x}_i)$. Since the optimal solution has the form $\mathbf{w} = \sum_i \alpha_i \mathbf{x}_i$, so $\|\mathbf{w}\|^2 = \sum_{i,j}\alpha_i\alpha_j\mathbf{x}_i^\mathsf{T}\mathbf{x}_j$. Kernelizing we get $\|\mathbf{w}\|^2 = \boldsymbol{\alpha}\mathbf{K}\boldsymbol{\alpha}$. If we absorb the $\hat{w}_0$ term into one of the kernels, we have $\mathbf{f} = \mathbf{K}\boldsymbol{\alpha}$, so $\|\mathbf{w}\|^2 = \mathbf{f}^\mathsf{T}\mathbf{K}^{-1}\mathbf{f}$. Hence the SVM objective can be written as:

$$J(\mathbf{f}) = \frac{1}{2}\mathbf{f}^\mathsf{T}\mathbf{f} + C\sum_{i=1}^{N}(1-y_i f_i)_+ \tag{14.14}$$

  Compare this to the MAP estimation for GP classifier:

$$J(\mathbf{f}) = \frac{1}{2}\mathbf{f}^\mathsf{T}\mathbf{f} - \sum_{i=1}^{N}\log \mathcal{P}(y_i|f_i) \tag{14.15}$$

  Although there is no conversion from SVM to the GP classifier because the hinge loss has no conversion to the logistic loss, because hinge loss is strictly 0 for errors larger 1 - which gives SVM sparse solutions.

- As discussed before kernel PCA is applying kernel trick to regular PCA. **Gaussian process latent variable model (GP-LVM)** is different way to combine kernels with probabilistic PCA (MLPP 15.5).
- The principal drawback of GPs is that it takes $\mathcal{O}(N^3)$ time to use. This is because of the need to invert the $N \times N$ kernel matrix $\mathbf{K}$. This is a key limitation of GPs on being used on larger datasets. Some approximation methods have been devised - see (MLPP 15.6).

# 15 Adaptive Basis Function Models

- In the previous two sections we talked about kernel methods to create non-linear models for regression and classification. The prediction takes the form $f(\mathbf{x}) = \mathbf{w}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x})$, where we define:

$$\boldsymbol{\phi}(\mathbf{x}) = [\kappa(\mathbf{x}, \boldsymbol{\mu}_1), \ldots, \kappa(\mathbf{x}, \boldsymbol{\mu}_N)] \tag{15.1}$$

  and where $\boldsymbol{\mu}_k$ are either all the training data or some subset. Models of this form essentially perform a form of template matching, whereby they compare the input $\mathbf{x}$ to the stored prototypes $\boldsymbol{\mu}_k$. Although this can work well, it relies on having a good kernel function to measure the similarity between data vectors. Sometimes coming a good kernel function is quite difficult. In this section we work with an alternative approach which dispenses kernels altogether. Rather we try to learn useful features $\boldsymbol{\phi}(\mathbf{x})$ directly from input data. That is we will create **adaptive basis functions** (MLPP 16.1) which have the form:

$$f(\mathbf{x}) = w_0 + \sum_{m=1}^{M} w_m \phi_m(\mathbf{x}) \tag{15.2}$$

  where $\phi_m(\mathbf{x})$ is the $m$'th basis function which is learned from data. The basis functions are usually parametric, so we can write $\phi_m(\mathbf{x}) = \phi(\mathbf{x}; \mathbf{v}_m)$, where $\mathbf{v}_m$ are the parameters of the basis functions. We will use $\boldsymbol{\theta} = (w_0, \mathbf{w}_{1:M}, \{\mathbf{v}_m\}_{m=1}^M)$ to denote all the parameters in the model. Note that the model is no longer linear in parameters anymore, hence we can only compute locally optimal MLE or MAP estimate of $\boldsymbol{\theta}$.

- **Classification and Regression Trees (CART)** or **decision trees** (MLPP 16.2.1) are defined by recursively partitioning the input space, and defining a local model in each resulting regions of input space. This can be conveniently represented as a tree, with one leaf per region. CART essentially divides the feature space through axis parallel/aligned splits (since splits are based on a single feature such as $x_i > \tau_m$). We can associate the mean response with each of these regions, resulting in the piecewise constant surface:

$$f(\mathbf{x}) = \mathbb{E}[y|\mathbf{x}] = \sum_{m=1}^{M} w_m \mathbb{I}(\mathbf{x} \in \mathcal{R}_m) = \sum_{m}^{M} w_m \phi(\mathbf{x}; \mathbf{v}_m) \tag{15.3}$$

  where $\mathcal{R}_m$ is the $m$'th region, $w_m$ is the mean response in this region, and $\mathbf{v}_m$ encodes the choice of variable to split on, and the threshold value, on the path from the root to the $m$'th leaf. This makes CART an adaptive basis function, where the basis function define the regions, and the weights specify the response value of each region.

- We can generalize CART to classification where rather than storing the mean response, we store the distribution class labels in each leaf.

- Growing the optimal tree is NP-complete, so it is common to adopt greedy procedures (MLPP 16.2.2). When learning the tree, the question at each node is what feature $j^*$ and what threshold $t^*$ to use to split the space. This can be specified as:

$$(j^*, t^*) = \underset{j \in \{1,\ldots,D\}}{\arg\min} \min_{t \in \mathcal{T}_j} \text{cost}(\{\mathbf{x}_i, \mathbf{y}_i : x_{ij} \leq t\}) + \text{cost}(\{\mathbf{x}_i, \mathbf{y}_i : x_{ij} > t\}) \tag{15.4}$$

  where $x_{ij}$ is the $j$'th feature on the $i$'th data point. We suppose that $x_{ij}$ is real valued or ordinal, so it makes sense thresholding against a numerical value $t$. The set $\mathcal{T}_j$ can be constructed by sorting the current set of inputs $\{x_{ij}\}_{\forall i}$. In case of categorical inputs, the most common approach is to consider splits of the form $x_{ij} = c_k$ and $x_{ij} \neq c_k$ for each possible class label $c_k$.

- Although we can have non-binary trees (multi-way splits), this would result in data fragmentation, meaning too little data might fall into each subtree, resulting in overfitting.

- There are many possible stopping criteria: (1) Is the reduction in cost too small; (2) has the tree exceeded the maximum desired depth; (3) is the distribution of the response in either $\mathcal{D}_L$ or $\mathcal{D}_R$ sufficiently homogeneous (in case all the labels are the same, in which case the distribution is pure); (4) The number of samples is too less in either $\mathcal{D}_L$ or $\mathcal{D}_R$.
- For **regression cost** (MLPP 16.2.2.1), one can simply use the squared error from the mean:

$$\text{cost}(\mathcal{D}) = \sum_{i \in \mathcal{D}} (y_i - \bar{y})^2, \qquad \bar{y} = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} y_i \tag{15.5}$$

- For **classification cost** (MLPP 16.2.2) there are several options. Given all the data points $\mathcal{D}$ at the right or left split, we first compute the fraction of data points in each class (multinoulli model) $\hat{\pi}_c = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \mathbb{I}(y_i = c)$:
  - **Misclassification rate**: We define the most probable class label as $\hat{y}_c = \arg\max_c \hat{\pi}_c$. The corresponding error rate is then: $\frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \mathbb{I}(y_i \neq \hat{y}) = 1 - \hat{\pi}_{\hat{y}}$.
  - **Entropy**: Minimizing the entropy is equivalent to maximizing the information gain between test $X_j < t$ and the class label $Y$, defined by:

$$\text{infoGain}(X_j < t, Y) \triangleq \mathbb{H}(Y) - \mathbb{H}(Y | X_j < t) \tag{15.6}$$

    where $\mathbb{H}(\hat{\pi}) = -\sum_{c=1}^{C} \hat{\pi}_c \log \hat{\pi}_c$
  - **Gini Index**: $1 - \sum_c \hat{\pi}_c^2$ is the expected error rate. To see this, note that $\hat{\pi}_c$ is the probability of a random entry in the leaf belonging to class $c$.

  Entropy and Gini measure are very similar and are more sensitive to changes in class probability than is the misclassification rate.
- To prevent overfitting, we might want to stop growing a tree as soon as the decrease in error is not sufficient enough. This is sort of a myopic view - a good example is xor where you might not split the tree at all. A better solution is to grow the full tree and then perform **pruning** (MLPP 16.2.3). This can be done by pruning the branches with the least increase in error. To determine how far back to prune, we can evaluate the cross-validation error on each such subtree, and then pick the tree whose CV error is within 1 standard error of the minimum.
- Advantage of CART: (1) easy to interpret; (2) handles mixed discrete/continuous input; (3) insensitive to monotone transformation; (4) no need for normalization; (5) auto variable selection; (6) relatively robust to outliers; (7) scales well with large data; (8) can deal with missing inputs. Disadvangtages: (1) prediction might not be as good because of greedy nature of tree construction; (2) small perturbations in data can greatly change the structure of the tree. Any errors on the top will propagate down from the tree. In frequentist terms the trees are high variance estimators which can be fixed by ...
- One way to reduce the variance of an estimate is to average together many estimates. For example, we can train $M$ different trees on different subsets of the data, chosen randomly with replacement, and then compute the ensemble:

$$f(\mathbf{x}) = \frac{1}{M} \sum_{m=1}^{M} f_m(\mathbf{x}) \tag{15.7}$$

  where $f_m$ is the $m$'th tree. This technique is called **bagging**, which stands for **bootstrap aggregating** (MLPP 16.2.5). Unfortunately simply re-running the same algorithm on different random subsets of data doesn't reduce the correlation of the predictors. **Random Forests** not only builds trees on random subsets of data, but also chooses variables at each node split from random variable subset, in order to decrease the variance of the predictor. These have better predictive accuracy. Rather than bagging, which is a frequentist concept, we can perform Bayesian inference over the space of ensembles of trees, which is called **Bayesian adaptive regression trees (BART)**.

- In contrast to CART, hierarchical mixture of experts can partition the input space using any set of nested linear decision boundaries (MLPP 16.2.6). Also, HME predictions are an average of experts (which doesn't lead overfitting). Moreover, fitting an HME involves solving a smooth continuous optimization problem (usually using EM), which is less likely to be prone to local optima than the standard greedy discrete optimization methods used to fit decision trees.
- Another way to create non-linear models with multiple inputs is to use a **generalized additive model (GAM)** (MLPP 16.3), which is a model of the form:

$$f(\mathbf{x}) = \alpha_0 + \sum_{j=1}^{D} f_j(x_j) \tag{15.8}$$

$$= \boldsymbol{\beta}^\mathsf{T} \boldsymbol{\phi}(\mathbf{x}), \qquad \text{where } \boldsymbol{\phi}(\mathbf{x}) = [1, \boldsymbol{\phi}_1(x_1), \ldots, \boldsymbol{\phi}_D(x_D)] \tag{15.9}$$

We can extend GAMs by allowing for interaction effects. In general we can make the following decomposition, even though we need to limit the number of higher-order interactions otherwise there will be too many parameters to fit:

$$f(\mathbf{x}) = \beta_0 + \sum_{j=1}^{D} f_j(x_j) + \sum_{j,k} f_{jk}(x_j, x_k) + \sum_{j,k,l} f_{jkl}(x_j, x_k, x_l) + \ldots \tag{15.10}$$

It is common to use a greedy search to decide which variables to add. The **multivariate adaptive regression splines (MARS)** (MLPP 16.3.3) algorithm is one example of this. It fits models of this form, where it uses a tensor product basis of regression splines to represent the multidimensional regression functions. The whole procedure is closely related to CART.

- **Boosting** (MLPP 16.4) is a greedy algorithm for fitting adaptive basis-function models of the form 15.2, where the $\phi_m$ are generated by an algorithm called a **weak learner**, The algorithm works by applying the weak learner sequentially to weighted versions of the data, where more weight is given to samples that were mis-classified by earlier rounds. A **boosted decision tree** is a model where the weak learner is a decision tree - these are supposed to be quite a good off-the-shelf classifier. Theory says that one could boost the performance (on the training set) of any weak learner arbitrarily high, provided the weak learner could always perform slightly better than chance. The test error also keeps reducing for a long time even after the training error has gone to zero. The test error would eventually go up - but boosting is very resistant to overfitting. The following table shows the common loss functions for boosting:

| Name | Loss | Derivative | $f^*$ | Algorithm |
|---|---|---|---|---|
| Squared error | $\frac{1}{2}(y_i - f(\mathbf{x}_i))^2$ | $y_i - f(\mathbf{x}_i)$ | $\mathbb{E}[y\|\mathbf{x}_i]$ | L2Boosting |
| Absolute error | $[y_i - f(\mathbf{x}_i)]$ | $\text{sgn}(y_i - f(\mathbf{x}_i))$ | $\text{median}(y\|\mathbf{x}_i)$ | Gradient boosting |
| Exponential error | $\exp(-\tilde{y}_i f(\mathbf{x}_i))$ | $-\tilde{y}_i \exp(-\tilde{y}_i f(\mathbf{x}_i))$ | $\frac{1}{2}\log\frac{\pi_i}{1-\pi_i}$ | AdaBoost |
| Logloss | $\log(1 + e^{-\tilde{y}_i f(\mathbf{x}_i)})$ | $\tilde{y}_i - \pi_i$ | $\frac{1}{2}\log\frac{\pi_i}{1-\pi_i}$ | LogitBoost |

- (MLPP 16.4.1) The goal of boosting is to solve the following optimization problem:

$$\min_f \sum_{i=1}^{N} L(y_i,\, f(\mathbf{x}_i)) \tag{15.11}$$

and $L(y, \hat{y})$ is some loss function, and $f$ is assumed to be an ABM model as in equation 15.2. For binary classification, the obvious loss is 0-1 loss, but this is not differentiable. Instead it is common to use logloss, which is a convex upper bound on 0-1 loss. In this case, one cn show that the optimal estimate is given by:

$$f^*(\mathbf{x}) = \frac{1}{2}\log\frac{\mathcal{P}(\hat{y} = 1|\mathbf{x})}{\mathcal{P}(\hat{y} = -1|\mathbf{x})} \tag{15.12}$$

where $\hat{y} = \{-1, +1\}$. An alternative convex upper bound is exponential loss, defined by:

$$L(\hat{y}, f) = \exp(-\hat{y}, f) \tag{15.13}$$

Since finding the optimal $f$ is hard, we shall tackle it sequentially. We initialize by defining:

$$f_0(\mathbf{x}) = \arg\min_{\boldsymbol{\gamma}} \sum_{i=1}^{N} L(y_i, f(\mathbf{x}_i; \boldsymbol{\gamma})) \tag{15.14}$$

For example, if we use squared error, we can set $f_0(\mathbf{x}) = \bar{y}$, and if we use log-loss or exponential loss, we can set $f_0(\mathbf{x}) = \frac{1}{2} \log \frac{\hat{\pi}}{1-\hat{\pi}}$, where $\hat{\pi} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(y_i = 1)$. Then an iteration $m$, we compute:

$$(\beta_m, \boldsymbol{\gamma}_m) = \arg\min_{\beta, \boldsymbol{\gamma}} \sum_{i=1}^{N} L(y_i,\ f_{m-1}(\mathbf{x}_i) + \beta\phi(\mathbf{x}_i; \boldsymbol{\gamma})) \tag{15.15}$$

and then we set: $f_m(\mathbf{x}) := f_{m-1}(\mathbf{x}) + \beta_m \phi(\mathbf{x}; \boldsymbol{\gamma}_m)$. The key point is that we do not go back and adjust earlier parameters. This is why the method is called **forward stagewise additive modeling**. We continue this for a fixed number of iterations $M$. In fact $M$ is the main tuning parameter of the method. Often we pick it by monitoring the performance on a separate validation set, and then stopping once performance starts to decrease; this is called **early stopping**. In practice, better (test set) performance can be obtained by performing "partial updated" of the form: $f_m(\mathbf{x}) := f_{m-1}(\mathbf{x}) + \nu\beta_m \phi(\mathbf{x}; \boldsymbol{\gamma}_m)$. Here $0 < \nu \leq 1$ is a step-size parameter. This is called **shrinkage**.

- **Adaboost** (MLPP 16.4.3) applies a $w_{i,m}$ is a weight applied to a data-case $i$

**Algorithm 3:** AdaBoost for binary classification with exponential loss

$w_i = 1/N$
**for** $i = 1 : N$ **do**
  Fit a classifier $\phi_m(\mathbf{x})$ to the training set using weights $\mathbf{w}$
  Compute $\epsilon_m = \frac{\sum_{i=1}^{N} w_{i,m} \mathbb{I}(\tilde{y}_i \neq \phi_m(\mathbf{x}_i))}{\sum_{i=1}^{N} w_{i,m}}$
  Compute $\alpha_m = \log[(1 - \epsilon_m)/\epsilon_m]$
  Set $w_i \leftarrow w_i \exp[\alpha_m \mathbb{I}(\tilde{y}_i \neq \phi_m(\mathbf{x}_i))]$
**return** $f(\mathbf{x}) = \text{sgn}\left[\sum_{m=1}^{M} \alpha_m \phi_m(\mathbf{x})\right]$

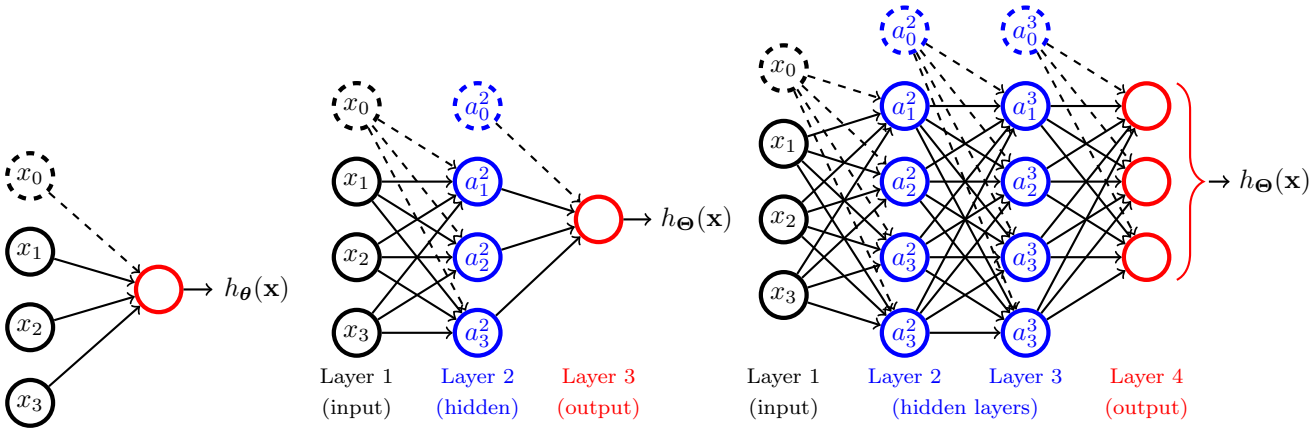
- Since Adaboost uses exponential loss, it puts too much weight on misclassified examples. This method makes it sensitive to outliers. In addition $e^{-\tilde{y}f}$ is not the logarithm of any pmf, for binary variables $\tilde{y} \in \{-1, +1\}$. A natural alternative is to use logloss instead. This only punishes mistakes linearly. Furthermore, it means that we will be able to extract probabilities from the final learned function, using

$$\mathcal{P}(y = 1|\mathbf{x}) = \frac{1}{1 + e^{-2f(\mathbf{x})}} \tag{15.16}$$

This can be used to minimize the expected log-loss and we get the **logitBoost** (MLPP 16.4.4).

- Rather than deriving new versions of boosting for every different loss function, it is possible to derive a generic version; known as **gradient boosting** (MLPP 16.4.5).

- Of all the possible variables, $j = 1 : D$, if we pick the one $j(m)$ that best predicts the residual error - and make this as our weak learner - we have an algorithm called **sparse boosting**(MLPP 16.4.6). This is identical to the matching pursuit algorithm. We will get a sparse solution if $M$ is small.




(a) Neuron model (b) Neural Network (c) Neural Network for 3-ary classification

Figure 10: In 10a we use a logistic unit, where the input $\mathbf{x}$ acting as dendrites, the red node is the processing unit, and the output wire leading to $h_{\boldsymbol{\theta}}(\mathbf{x})$ is the Axon. The dotted line shows the constant bias unit with $x_0 = 1$ input. In 10b each layers output is passed as an input to the next layer. Layers sandwiched between input and output are called the hidden layers. All layers (except the output layer) also have a bias unit. 10c extends the neural network model in 10b to a multi-class classification problem. In this example we are trying to classify three possible classes.

- It is common to use CART models as weak learners. It is usually advisable to use a shallow tree, so that the variance is low, even though the bias will be high - which is compensated with boosting. The parameters to tune are the height of the tree; the number of rounds of boosting, $M$; and $\nu$, the shrinkage factor. If we restrict trees to $J$ leaves ($J = 2$ is a **stump** i.e. only one variable decision; $J = 3$ is three variable decision), then empirically its shown that $J \approx 6$ gives good results. If we combine gradient boosting with shallow regression trees, it is known as **multivariate adaptive regression trees (MART)**.
- Boosting works well because of two reasons (MLPP 16.4.8). It can be viewed as $\ell_1$ regularization because once we have all possible weak learners of the form $\boldsymbol{\phi}(\mathbf{x}) = [\phi_1(\mathbf{x}), \dots, \phi_K(\mathbf{x})]$, we can use $\ell_1$ regularization to select a subset of these. Alternatively we can use boosting where at each step a weak learner creates a new $\phi_k$ on the fly. Another reason for the power of boosting is that it maximizes the margin on the training data.
- The explanation of boosting up until now has been very frequentist. For a more Bayesian view (MLPP 16.4.9), you can think of a mixture of experts model where each expert $\mathcal{P}(y|\mathbf{x}, \boldsymbol{\gamma}_m)$ is like a weak learner. In this scheme, when training with EM, in the E step, where the posterior responsibilities reflect how well the existing experts explain a data point; if this is a poor fit, these data points will have more influence on the next expert that is fitted.
- As the number of features increase, creating a function with polynomial features becomes really expensive. For example, if $\boldsymbol{x} \in \Re^{100}$, the number of parameters required for a function containing all second order features (i.e. $\{x_1^2, x_1x_2, x_1x_3, \dots, x_1x_{100}, x_2^2, x_2x_3, \dots\}$) would be $\approx 5000$. **Neural Networks** (MLPP 16.5) (**multi-layer perceptrons (MLP)**) provide a natural way to deal with this problem.
- Each neuron in the brain is a small processing unit. It has a number of input wires called the **Dendrites**, which passes messages to the nucleus, does some processing, and passes the output through its output wires known as the **Axons**. In an *artificial* Neuron model, we will use a simple logistic unit. These networks receive input, let's say $\mathbf{x} = [x_0, x_1, x_2, x_3]^\mathsf{T}$, and work with parameters $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \theta_3]^\mathsf{T}$, where, $x_0 = 1$ always - hence known as the **bias unit**. The so called **activation function** or **transfer function** $g(\mathbf{x}; \boldsymbol{\theta})$ usually a sigmoid (logistic) or a tanh function. Figure 10a shows an example. The parameters $\boldsymbol{\theta}$ are also called **weights** in neural network literature. It is important that the activation function to be non-linear otherwise the whole model boils down to a large linear model.

- We can augment the neuron model to create a neural network. The representation is similar, as given in Figure 10b. Let's introduce some new notation: $a_i^l$ is the activation of unit $i$ in layer $l$; the neural network is parameterized by $\mathbf{\Theta}^l$, which is a matrix of weights/parameters controlling the function mapping from layer $l$ to layer $l+1$. We will denote row $r$ and column $c$'s value in the parameter matrix as $\mathbf{\Theta}^l_{[r][c-1]}$. The computation involved in a neural network is iterative. For the model given in Figure 10b, the computation is as follows:

$$a_r^2 = g\left(\mathbf{\Theta}^1_{r0}x_0 + \mathbf{\Theta}^1_{r1}x_1 + \mathbf{\Theta}^1_{r2}x_2 + \mathbf{\Theta}^1_{r3}x_3\right) \qquad = g\left(\mathbf{z}_r^2\right) \tag{15.17}$$

$$h_{\mathbf{\Theta}}(\mathbf{x}) = a_1^3 = g\left(\mathbf{\Theta}^2_{10}a_0^2 + \mathbf{\Theta}^2_{11}a_1^2 + \mathbf{\Theta}^2_{12}a_2^2 + \mathbf{\Theta}^2_{13}a_3^2\right) \qquad = g\left(\boldsymbol{z}_1^3\right) \quad , \tag{15.18}$$

where $g(\cdot)$ is the non-linear activation function. Notice that $\mathbf{\Theta}^1 \in \Re^{3\times 4}$ i.e. it is a 3 rows (because of there are 3 un-biased nodes in the hidden layer), 4 columns parameter matrix, where row $r$ gives parameters for the activation function $a_r^2$. If the **network architecture** has $s_l$ units in layer $l$ (excluding the bias unit), and $s_{l+1}$ units in the next layer, then $\mathbf{\Theta}^l \in \Re^{s_{l+1}\,\times\,(s_l+1)}$. For our notation it is important to remember that $\mathbf{\Theta}^l$ are the parameters used by layer $l+1$. We will also denote the input at layer $l$ as $\mathbf{a}^{l-1}$. Hence, $\mathbf{a}^1 = \mathbf{x}$. Now, we can simply write the computation at layer $l$ as:

$$\mathbf{z}^l = \mathbf{\Theta}^{l-1}\mathbf{a}^{l-1} \tag{15.19}$$

$$\mathbf{a}^l = g\left(\mathbf{z}^l\right) \quad , \tag{15.20}$$

where $\mathbf{a}^l = [a_0^l,\, a_1^l,\, \ldots\,,\, a_{s_l}^l]^\mathsf{T}$. The above algorithm is known as **Forward Propagation**, since we are always pushing the outputs from one layer as inputs to the next layer. The idea of learning parameters at layer $l$ from the outputs of layer $l-1$, allows neural networks to learn highly non-linear functions. Adding layers is like adding to the order of features accomodated by our output function. Probabilistically, if we were doing binary classification, we pass the output through a sigmoid, as in a GLM:

$$\mathcal{P}(y|\mathbf{x}, \mathbf{\Theta}) = \mathrm{Ber}(y \,|\, \mathrm{sigm}(\mathbf{\Theta}^{L-1}\mathbf{a}^{L-1})) \tag{15.21}$$

For regression it would be:

$$\mathcal{P}(y|\mathbf{x}, \mathbf{\Theta}) = \mathcal{N}(y \,|\, \mathbf{\Theta}^{L-1}\mathbf{a}^{L-1},\, \sigma^2) \tag{15.22}$$

- The original neural network architecture can be extended to act as a multi-class classification algorithm. In this setting if we are classifying $K$ classes, $h_{\mathbf{\Theta}}(\boldsymbol{x}) \in \Re^K$. For the example given in Figure 10c, $K = 3$. Our aim here would be to have the following binary vector outputs for each of the 3 classes case,

$$h_{\mathbf{\Theta}}(\boldsymbol{x}) \approx \begin{cases} \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^\mathsf{T} & \text{when class 1} \\ \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^\mathsf{T} & \text{when class 2} \\ \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^\mathsf{T} & \text{when class 3} \end{cases} . \tag{15.23}$$

If we add **mutual inhibition** arcs between the output units, ensuring that only one of them turns on, we can enforce a sum-to-one constraint. Essentially we train each output unit to classify a specific class. Hence, Figure 10c has 3 output units (at layer 4). Following our figure we can represent out training set of $N$ examples as $\left(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}\right)$, $\left(\mathbf{x}^{(2)}, \mathbf{y}^{(2)}\right)$, $\ldots$ $\left(\mathbf{x}^{(N)}, \mathbf{y}^{(N)}\right)$, where $\mathbf{y}^{(i)}$ takes one of three vector outputs given in Equation 15.23, i.e:

$$y^{(i)} \in \left\{ \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\} \tag{15.24}$$

Now that we know how to represent neural networks in both multi-class and binary classification problems, the question is how to train them. In other words how do we find our parameters set $\mathbf{\Theta}^l$ for all layers. An interesting situation to think about here is the case of two layer neural network. The parameters of a two layer neural network can be easily found by gradient descent methods we have encountered previously.

- **Cost function for Neural Network Learning**: Some notation: $L$ is the number of layers in the neural network; $s_l$ is the number of units in layer $l$, when not counting the bias unit. For instance, $L = 4$ and $s_2 = 4$ in Figure 10c. Since $K$ is the number of classes being classified, for binary-classification $K = 1$ and there is only 1 output unit giving $y \in \{0, 1\}$. Hence, $h_{\mathbf{\Theta}}(\mathbf{x}) \in \Re$ and the number of units in the output layer is $s_L = 1$. For multi-class classification when $K \geq 3$ (note $K = 2$ should be treated as a binary classification problem - not as a multi-class problem), there are $K$ output units and $y \in \Re^K$ as demonstrated in Equation 15.24. Hence, $h_{\mathbf{\Theta}}(\boldsymbol{x}) \in \Re^K$. Note that $s_L = K$, i.e. the number of output units is equal to the number of dimensions in the output variable $y$.

  The cost function for a neural network would be a generalization of the logistic regression cost function. Rather than having "one logistic regression output per unit, we will have $K$ of them." The cost function:

$$J(\mathbf{\Theta}) = -\frac{1}{N}\left[\sum_{i=1}^{N}\underbrace{\sum_{k=1}^{K}\mathbf{y}_k^{(i)}\log\left(h_{\mathbf{\Theta}}(\mathbf{x}^{(i)})_k\right) + (1-\mathbf{y}_k^{(i)})\log\left(1-h_{\mathbf{\Theta}}(\mathbf{x}^{(i)})_k\right)}_{\text{sum over } K \text{ output units}}\right] + \frac{\lambda}{2N}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\mathbf{\Theta}_{ji}^{l}\right)^2, \tag{15.25}$$

  where $h_{\mathbf{\Theta}}(\mathbf{x}^{(i)}) \in \Re^K$, and $h_{\mathbf{\Theta}}(\mathbf{x}^{(i)})_k$ refers to the $k$-th element in the vector. Similarly, $\mathbf{y}^{(i)} \in \Re^K$, and $\mathbf{y}_k^{(i)}$ refers to the $k$-th element. Remember that $K = 1$ for the binary classification case. As discussed previously, $\mathbf{\Theta}^l \in \Re^{s_{l+1} \times s_l + 1}$, i.e. the number of parameters needed for layer $l+1$ is equivalent to the number of inputs for layer $l$, $s_l$ (plus the bias unit) times the number of units (sans the bias unit) in layer $l+1$, $s_{l+1}$. Note that for the regularization constraint at layer $l+1$, $\sum_{i=1}^{s_l}(\cdot)$ sums over the number of inputs, and $\sum_{j=1}^{s_{l+1}}(\cdot)$ sums over the number of units at that layer.

  Although the first term in Equation 15.25 only sums over the number of output units, $h_{\mathbf{\Theta}}(\cdot)$ is a function over all parameters in the neural network. This will allow us to optimize all the parameters in the neural network.

- **Backpropagation Algorithm**: Given the $J(\mathbf{\Theta})$ in Equation 15.25, we would like to find $\min_{\mathbf{\Theta}} J(\mathbf{\Theta})$. For this, for a gradient dependent optimization method, we would need to compute $J(\mathbf{\Theta})$ and $\frac{\partial}{\partial \mathbf{\Theta}_{ji}^{l}} J(\mathbf{\Theta})$ for any provided $\mathbf{\Theta}$. Suppose that we only have one training example, i.e. $N = 1$. This will reduce Equation 15.25 to

$$J(\mathbf{\Theta}) = -\left[\underbrace{\sum_{k=1}^{K}\mathbf{y}_k^{(i)}\log\left(h_{\mathbf{\Theta}}(\mathbf{x}^{(i)})_k\right) + (1-\mathbf{y}_k^{(i)})\log\left(1-h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})_k\right)}_{\text{sum over } K \text{ output units}}\right] + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\mathbf{\Theta}_{ji}^{(l)}\right)^2 \quad . \tag{15.26}$$

  First, following the forward propagation model (given in Equation 15.19 and 15.20), the final activation for Figure 10c would be:

$$h_{\mathbf{\Theta}}(\mathbf{x}) = \mathbf{a}^4 = g(\mathbf{z}^4) = g\left(\mathbf{\Theta}^3 g\left(\mathbf{\Theta}^2 g\left(\mathbf{\Theta}^1 \mathbf{a}^1\right)\right)\right) \tag{15.27}$$

  Note that if we use Equation 15.27 to compute the $\frac{\partial}{\partial \mathbf{\Theta}_{ji}^{l}} J(\mathbf{\Theta})$, it would be increasingly impossible, as it calls for derivating repeated invocations of the non-linear $g(\cdot)$. We take an alternative approach where we will keep track of $\delta_j^l$, the error in each unit $j$ in layer $l$. The error here means how much we would like to change $\mathbf{a}_j^l$ to get to the true value. Looking at figure 10c, for each output unit ($L = 4$):

$$\delta_j^4 \;=\; \mathbf{a}_j^4 - \mathbf{y}_j \;=\; h_{\mathbf{\Theta}}(\mathbf{x})_j - \mathbf{y}_j \tag{15.28}$$

$$\delta^4 \;=\; \mathbf{a}^4 - \mathbf{y} \tag{15.29}$$

For all the previous layers we have the error terms are computed as follows (why????):

$$\delta^3 = \left(\boldsymbol{\Theta}^3\right)^{\mathsf{T}} \delta^4 .* g'\left(\boldsymbol{z}^3\right) \tag{15.30}$$

$$\delta^2 = \left(\boldsymbol{\Theta}^2\right)^{\mathsf{T}} \delta^3 .* g'\left(\boldsymbol{z}^2\right) . \tag{15.31}$$

where $g'\left(\mathbf{z}^3\right) = \mathbf{a}^3 .* (1 - \mathbf{a}^3)$ and $g'\left(\mathbf{z}^2\right) = \mathbf{a}^2 .* (1 - \mathbf{a}^2)$. Note that there is no $\delta^1$ since there is no "error" in the input term. This algorithm is called **back-propagation** since all the errors are propagated back to previous layers. If we ignore the regularization terms, it can be proved that the partial derivative is equal to

$$\frac{\partial}{\partial \boldsymbol{\Theta}^l_{ji}} J(\boldsymbol{\Theta}) = \mathbf{a}^l_j \delta^{l+1}_i \qquad \text{if } \lambda = 0 \tag{15.32}$$

Since the total error term for $N$ training samples would just be the sum of the error term computed in Equation 15.32 for each example, our final algorithm would be as given in Algorithm 4.

**Algorithm 4:** Neural networks learning by **Back Propagation**

**Data**: Training set $\left\{\left(\mathbf{x}^{(1)}, y^{(1)}\right), \left(\mathbf{x}^{(2)}, y^{(2)}\right), \ldots \left(\mathbf{x}^{(N)}, y^{(N)}\right)\right\}$
$\Delta^l_{ji} := 0 \qquad \forall\, l, i, j$ // will be used to compute $\frac{\partial}{\partial \boldsymbol{\Theta}^l_{ji}} J(\boldsymbol{\Theta})$
/* iterate over all training examples */
**for** $i := 1 : N$ **do**
  $\mathbf{a}^1 := \mathbf{x}^{(i)}$ // initialize the activation of the input layer
  Forward propagation (Eq. 15.19 and 15.20) to compute $\mathbf{a}^l$ for $l \in \{2, 3, \ldots, L\}$
  $\delta^L = \mathbf{a}^L - y^{(i)}$ // compute error on the output layer
  /* back propagate error */
  **for** $l := (L-1)$ *to* $2$ **do**
    $g'\left(\mathbf{z}^l\right) := \mathbf{a}^l .* (1 - \mathbf{a}^l)$
    $\delta^l := \left(\boldsymbol{\Theta}^l\right)^{\mathsf{T}} \delta^{l+1} .* g'\left(\mathbf{z}^l\right)$
  $\Delta^l_{ji} := \Delta^l_{ji} + \boldsymbol{a}^l_i \delta^{l+1}_j$
**for** $\forall\, j$ **do**
  **if** $j = 0$ **then**
    5 $D^l_{ji} = \frac{1}{N} \Delta^l_{ji}$
  **else**
    $D^l_{ji} = \frac{1}{N} \Delta^l_{ji} + \frac{\lambda}{N} \boldsymbol{\Theta}^l_{ji}$

The last for-loop adds the regularization term. It can be mathematically proven that $\frac{\partial}{\partial \boldsymbol{\Theta}^l_{ji}} J(\boldsymbol{\Theta}) = D^l_{ji}$.

- There is one important point about running the optimization over Algorithm 4. Suppose if we initialize all $\boldsymbol{\Theta}^l$ to a constant $\mathbf{0}$, this essentially means setting all the weights to 0. This is equivalent to setting $g(\mathbf{z}^l_j) = \frac{1}{2}$. Hence, all units in all layers will exactly be the same. A direct corollary of this $\mathbf{0}$ initialization is that even all error terms $\delta^l_k$ will be the same. This would make all partial derivatives the same too. Hence, after each update, parameters corresponding to inputs going into each of the $s_{j+1}$ hidden units would be identitical. This is not only true for choosing an initial parameter $\mathbf{0}$ but for any other constant value. Hence it is important to randomly initialize the parameters.

- The purpose of the hidden units is to learn non-linear combinations of the original inputs; this is called **feature extraction** or **feature construction** (MLPP 16.5.1). These hidden features are then passed as input to the final GLM. This approach is particularly useful for problems where the original input features are not very individually informative. A form of MLP which is particularly well suited to 1d signals like speech or text, or 2d signals like images, is the **convolutional neural network**. This is an MLP in which the

hidden units have local **receptive fields** (as in the primary visual cortex), and in which the weights are tied or shared across the image, in order to reduce the number of parameters. Intuitively, the effect of such spatial parameter tying is that any useful features that are "discovered" in some portion of the image can be re-used everywhere else without having to be independently learned. The resulting network then exhibits translation invariance, meaning it can classify patterns no matter where they occur inside the input image. For instance if you have an input image of $29 \times 29$, the first layer could be 6 **feature maps**, each of which is of $13 \times 13$ size. Each one of these $13 \times 13 \times 6 = 1014$ hidden nodes/neurons is computed by convolving $5 \times 5$ weight matrix, adding a bias, and then passing the result through some non-linear function. Each of the 6 feature map only has $5 \times 5 + 1 = 26$ adjustable weights (which need to be learned). This is much less than the full model which did not share these parameters, $1014 \times 26 = 26,364$, as compared to $6 \times 26 = 156$ weights.

- If we allow feedback connections, the model is known as a **recurrent neural network (RNN)** (MLPP 16.5.2); this defines a nonlinear dynamical system.
- **Ensemble Learning** (MLPP 16.6) refers to learning a weighted combination of base models of the form:

$$f(y|\mathbf{x}, \boldsymbol{\pi}) = \sum_{m \in \mathcal{M}} w_m f_m(y|\mathbf{x}) \tag{15.33}$$

where $w_m$ are tunable weights. Ensemble learning is sometimes called a committee method, since each base model $f_m$ gets a weighted vote. This is clearly related to learning adaptive-basis function models. One can also argue that neural net is an ensemble method where $f_m$ represents the $m$'th hidden unit, and $w_m$ are output layer weights. The weights are learned by a method called **stacking** which is simply minimizing loss over all examples trained with LOOCV estimate:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} w_m \hat{f}_m^{-i}(\mathbf{x})\right) \tag{15.34}$$

where $\hat{f}_m^{-i}(\mathbf{x})$ is the predictor obtained by training on data excluding $(\mathbf{x}_i, y_i)$.

- All the models discussed in this section have good predictive accuracy, but they are black boxes in the sense that they are hard to diagnose what features are important. One useful way to measure the effect of a set $s$ of variables on the output is to compute a **partial dependence plot** (MLPP 16.8). This is a plot of $f(\mathbf{x}_s)$ vs $\mathbf{x}_s$, where $f(\mathbf{x}_s)$ is defined as the response to $\mathbf{x}_s$ with the other predictors averaged out:

$$f(\mathbf{x}_s) = \frac{1}{N} \sum_{i=1}^{N} f(\mathbf{x}_s, \mathbf{x}_{i,-s}) \tag{15.35}$$

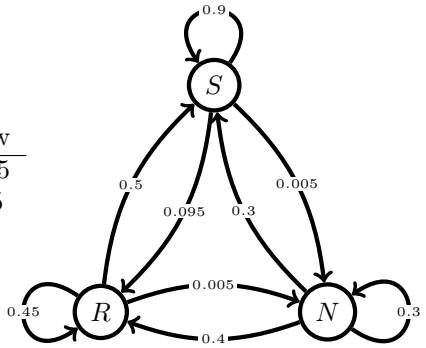
Another useful summary is to find the **relative important of predictor variables**. The basic idea can be viewed in ensemble of trees, where we count how often variable $j$ is used as a node in any of the trees. In particular, let $v_j = \sum_{m=1}^{M} \mathbb{I}(j \in T_m)$ be the proportion of all splitting rules that use $x_j$, where $T_m$ is the $m$'th tree.

# 16 Markov and Hidden Markov Models

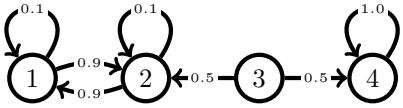
- Here we discuss probabilistic models for sequences of observations, $X_1, \ldots, X_T$, of arbitrary length $T$.
- A Markov chain, as discussed before, assumes that it is enough to know $X_t$ captures all the relevant information for predicting the future:

$$\mathcal{P}(X_{1:T}) = \mathcal{P}(X_1) \prod_{t=2}^{T} \mathcal{P}(X_t|X_{t-1}) \tag{16.1}$$

| | Sunny | Rain | Snow |
|---|---|---|---|
| Sunny (S) | 0.9 | 0.095 | 0.005 |
| Rain (R) | 0.5 | 0.45 | 0.05 |
| Snow (N) | 0.3 | 0.4 | 0.3 |




(a) State transition table and diagram for Atlanta's weather model




(b) A reducible 4-state chain

Figure 11

If we assume the transition function $\mathcal{P}(X_t|X_{t-1})$ is independent of time, then the chain is called **homogeneous**, **stationary**, or **time-invariant**. This is an example of **parameter tying**, since the same parameter is shared by multiple variables. This allows us to model an arbitrary number of variables using a fixed number of parameters; such models are called **stochastic processes**. If we assume the observe variables are discrete, so $X_t \in \{1, \dots, K\}$, this is called a discrete-state or finite-state Markov chain.

- When $X_t$ is discrete, so $X_t \in \{1, \dots, K\}$., the conditional distribution $\mathcal{P}(X_t|X_{t-1})$ can be written as a $K \times K$ matrix, known as the **transition matrix A**, where $A_{ij} = \mathcal{P}(X_t = j|X_{t-1} = i)$ is the probability of going from state $i$ from state $j$ (MLPP 17.2.1). Each row of the matrix sums to one $\sum_j A_{ij} = 1$, hence making it a stochastic matrix. A stationary, finite-state Markov chain is equivalent to a **stochastic automaton**. It is common to visualize such automata by drawing a directed graph (see Figure 11a), where nodes represent states and arrows represent legal transitions, i.e., non-zero elements of **A**. This is known as a **state transition diagram**. The weights associated with the arcs are the probabilities. The $A_{ij}$ element of the transition matrix specifies the probability of getting from $i$ to $j$ in one step. The $n$-step transition matrix $A(n)$ is defined as $A_{ij}(n) \triangleq Pr(X_{t+n} = j|X_t = i)$, which is the probability of getting from $i$ to $j$ in exactly $n$ steps. Obviously $\mathbf{A}(1) = \mathbf{A}$. The **Chapman-Kolmogorov** equations state that:

$$A_{ij}(m+n) = \sum_{k=1}^{K} A_{ij}(m)A_{jk}(n), \qquad \mathbf{A}(m+n) = \mathbf{A}(m)\mathbf{A}(n), \qquad \text{hence} \qquad \mathbf{A}(n) = \mathbf{A}^n \tag{16.2}$$

- We define the state space to be all the words in English (or some other language). The marginal probabilities $\mathcal{P}(X_t = k)$ are called **unigram** statistics. If we use a first-order Markov model, then $\mathcal{P}(X_t = k|X_{t1} = j)$ is called a **bigram model**. If we use a second-order Markov model, then $\mathcal{P}(X_t = k|X_{t-1} = j, X_{t-2} = i)$ is called a **trigram model**. And so on. In general these are called **n-gram models**.

- Given data $\mathcal{D} = (\mathbf{x}_1, \dots, \mathbf{x}_N)$, where $\mathbf{x}_i = (x_{i,1}, \dots, x_{i,T_i})$ is a sequence of $T_i$ states, the log likelihood is given by:

$$\log \mathcal{P}(\mathcal{D}|\boldsymbol{\theta}) = \sum_{i=1}^{N} \log \mathcal{P}(\mathbf{x}_i|\boldsymbol{\theta}) = \sum_{j=1}^{K} N_j^1 \log \pi_j + \sum_{j=1}^{K}\sum_{k=1}^{K} N_{jk} \log A_{jk} \tag{16.3}$$

where $\pi_j$ is the probability that you start a particular state. Plus we define the following counts:

$$N_j^1 \triangleq \sum_{i=1}^{N} \mathbb{I}(x_{i,1=k}), \qquad N_{jk} \triangleq \sum_{i=1}^{N}\sum_{t=1}^{T_i-1} \mathbb{I}(x_{i,t} = j,\, x_{i,t+1} = k) \tag{16.4}$$

where $N_j^1$ shows how many times state $j$ occurs as the first state. $N_{jk}$ just counts how many times state $j$ is followed by state $k$ in the data $\mathcal{D}$. Hence, we can write the MLE as normalized counts:

$$\hat{\pi}_j = \frac{N_j^1}{N}, \qquad \hat{A}_{jk} = \frac{N_{jk}}{\sum_{k=1}^{K} N_{jk}} \tag{16.5}$$

These are essentially MLE estimates for creating the transition table from data (MLPP 17.2.2.1). The problem of zero-counts becomes acute because in an n-gram model there are $\mathcal{O}(K^n)$ parameters to fit. There might be not enough data to find all the parameters. One simple solution is add-one smoothing, which adds one to all empirical counts before normalizing - which supposes all $n$-grams are equally likely unless indicated by data, which is not very realistic.

- A common heuristic used to fix the sparse data problem is **deleted interpolation** (MLPP 17.2.2.2). This defines the transition matrix as a convex combination of the bigram frequencies $f_{jk} = N_{jk}/N_j$ and the unigram frequencies $f_k = N_k/N$:

  $$A_{jk} = (1 - \lambda) f_{jk} + \lambda f_k \tag{16.6}$$

  The term $\lambda$ is usually set by cross validation. There is also a closely related technique called **backoff smoothing**; the idea is that if $f_{jk}$ is too small, we "back off" to a more reliable estimate, namely $f_k$. The deleted interpolation heuristic is an approximation to the prediction made by a simple hierarchical Bayesian model. Unlike deleted interpolation, the Bayesian model uses a context-dependent weight $\lambda_k$ to combine $m_k$, the prior mean, with the empirical frequency $f_{jk}$. The prior mean can be set by seeing the number of different contexts it can occur $m_k \propto |\{j : N_{jk} > 0\}|$, rather than just the number of times it occurs.

- To handle out of vocabulary states, we can use an a special symbol **unk**, and assign a certain amount of probability to it (MLPP 17.2.2.3).

- We have looked at Markov models as joint probability distributions, but we can also view them as stochastic dynamical systems, where we hop between states. In this interpretation we can think of the long term distribution over states, which is known as the **stationary distribution** of the chain.

  Let $A_{ij} = \mathcal{P}(X_t = j \,|\, X_{t-1} = i)$ be the one-step transition matrix and let $\pi_t(j) = \mathcal{P}(X_t = j)$ be the probability of being in state $j$ at time $t$. It is conventional in this context to assume that $\boldsymbol{\pi}$ is a row vector. If we have an initial distribution over states $\boldsymbol{\pi}_0$, then at time 1 we have:

  $$\pi_1(j) = \sum_i \pi_0(i) A_{ij}, \qquad \boldsymbol{\pi}_1 = \boldsymbol{\pi}_0 \mathbf{A} \tag{16.7}$$

  If we can find a stage $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{A}$ then we say we have reached the **stationary distribution** (aka **invariant** or **equilibrium** distribution) (MLPP 17.2.3.1). The thing special about this state is that once we reach it we can never leave this distribution. If we consider a markov chain, we can find the stationary distribution for variable $i$, $\pi_i$ by following this **global balance equation**:

  $$\pi_i \sum_{j\,:\,i \neq j} A_{ij} = \sum_{j\,:\,i \neq j} \pi_j A_{ji}, \quad \text{s.t.} \quad \sum_{i=1}^{K} \pi_i = 1 \tag{16.8}$$

  i.e. the probability of being in state $i$ times the net flow out of state $i$ must equal to the probability of being in each other state $j$ times the net flow from the state into $i$.

- To find a the stationary distribution, we can solve the eigenvector problem $\mathbf{A}^\mathsf{T}\mathbf{v} = \mathbf{v}$ and then set $\boldsymbol{\pi} = \mathbf{v}^\mathsf{T}$, where $\mathbf{v}$ is an eigenvector with eigenvalue 1 (MLPP 17.2.3.2). We can be sure such an eigenvector exists, since $\mathbf{A}$ is a row-stochastic matrix, so $\mathbf{A1} = \mathbf{1}$. We need to normalize $\mathbf{v}$ at the end to ensure it sums to one. Note, however the eigenvectors are only guaranteed to be real-valued if the matrix is positive $A_{ij} > 0$. There are ways to handle this general case where some transition probabilities can be 0 or 1.

- If we look at Figure 11b, we can see that if we start with state 4 we will always remain there - hence its an **absorbing state**. Thus $\boldsymbol{\pi} = (0, 0, 0, 1)$ is a possible stationary distribution; another one is $\boldsymbol{\pi} = (0.5, 0.5, 0, 0)$. If we start at state 3, we will end up at either of the two stationary distributions. This is called a **reducible chain** (MLPP 17.2.3.3). Hence a necessary condition to have a unique stationary distribution is that its a singly connected component i.e. you can reach any state from any other state - this is called **irreducible**.

- Let us say that a chain has a limiting distribution if $\pi_j = \lim_{n\to\infty} A_{ij}^n$ exists and is independent of $i$, for all $j$. If this holds, then the long-run distribution over states will be independent of the starting state:

$$\mathcal{P}(X_t = j) = \sum_i \mathcal{P}(X_0 = i) A_{ij}(t) \to \pi_j \text{ as } t \to \infty \tag{16.9}$$

  We define the **period** of state $i$, to be $d(i) = \gcd\{t : A_{ii}(t) > 0\}$. We say a state $i$ is **aperiodic** if $d(i) = 1$. A sufficient condition to ensure that the state is aperiodic if you can go from state $i$ to state $j$ and back, or there is a self-loop. We say a chain is aperiodic if all its states are aperiodic.

- To generalize the condition for having a stationary distributions to Markov chains whose state space is not finite, we need to generalize these two definitions. Now for a stationary distribution, apart from irreducibility and aperiodicity, you need each state to be **recurrent**. A **recurrent** state means that you will return to this state with probability 1. In Figure 11b, state 3 is non-recurrent / **transient** state. It is clear that any finite-state irreducible chain is recurrent. If our states were integers, and our chain was random walk, and if we started from state 0, then we can always return to state 0, but the distribution over all other states keeps spreading over a larger set of integers as time goes by. Hence, the distribution never converges to a stationary distribution. We can define a state to be **non-null recurrent** if the expected time to return to this state is finite. A state is **ergodic** if it is aperiodic, recurrent, and non-null, and if all states are ergodic, then the chain is ergodic.

  **Theorem 16.1.** Ergodic Markov Chain (MLPP Theorem 17.2.2)*: Every irreducible (signly connected), ergodic Markov chain has a limiting distribution which is equal to $\boldsymbol{\pi}$, its unique stationary distribution.*

- **Hidden Markov Model (HMM)** (MLPP 17.3) consistes of a discrete-time discrete-state Markov chain, with hidden states $z_t \in \{1, \dots, K\}$, plus an observation model $\mathcal{P}(\mathbf{x}_t|z_t)$. The corresponding joint distribution has the form:

$$\mathcal{P}(\mathbf{z}_{1:T}, \mathbf{x}_{1:T}) = \mathcal{P}(\mathbf{z}_{1:T})\mathcal{P}(\mathbf{x}_{1:T}|\mathbf{z}_{1:T}) = \left[\mathcal{P}(z_1)\prod_{t=2}^{T}\mathcal{P}(z_t|z_{t-1})\right]\left[\prod_{t=1}^{T}\mathcal{P}(\mathbf{x}_t|z_t)\right] \tag{16.10}$$

  The observations in an HMM can be discrete or continuous. If they are discrete, it is common for the observation model to be an observation matrix:

$$\mathcal{P}(\mathbf{x}_t = l \,|\, z_t = k, \boldsymbol{\theta}) = B(k, l) \tag{16.11}$$

  If the observations are continuous, it is common for the observation model to be a conditional Gaussian:

$$\mathcal{P}(\mathbf{x}_t|z_t = k, \boldsymbol{\theta}) = \mathcal{N}(\mathbf{x}_t|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \tag{16.12}$$

- **Inference in HMMs** (MLPP 17.4) is basically inferring the hidden state from the observations, assuming the parameters are known. The same algos also apply to other chain-structured graphical models such chain CRFs. An example that is used is the occasionally dishonest casino, where you have a fair and loaded dice, and between some throws you switch dices, where the in the loaded dice you have 1/2 a chance of getting a 6. Here, the observations $x_t$ are the face of the dice rolled, and $z_t$ has two states $z_t \in \{\text{L}, \text{F}\}$. To infer the hidden state there are different methods:

- **Filtering** means computing the **belief state** $\mathcal{P}(z_t|\mathbf{x}_{1:t})$ online, or recursively, as the data steams in. This reduces nouse more than simply estimating the hidden state using just the current estimate, $\mathcal{P}(z_t|\mathbf{x}_t)$. In the context of HMM this can be done by the **forwards algorithm**.
- **Smoothing** means computing $\mathcal{P}(z_t|\mathbf{x}_{1:T})$ offline, given all the evidence. In the context of HMM this can be done by the **forwards-backwards algorithm**.
- **Fixed lag smoothing** means $\mathcal{P}(z_{t-\ell}|\mathbf{x}_{1:t})$, where $\ell > 0$ is called the lag. Gives better performance than filtering, but incurs slight delay. By changing the size of lag one can tradeoff accuracy vs delay.
- **Prediction** we might want to predict the future given the past, i.e., to compute $\mathcal{P}(z_{t+h}|\mathbf{x}_{1:t})$, where $h > 0$ is the prediction horizon. If suppose $h = 2$; then we have:

$$\mathcal{P}(z_{t+2}|\mathbf{x}_{1:t}) = \sum_{z_{t+1}} \sum_{z_t} \mathcal{P}(z_{t+2}|z_{t+1})\mathcal{P}(z_{t+1}|z_t)\mathcal{P}(z_t|\mathbf{x}_{1:t}) \tag{16.13}$$

- **MAP estimation** means computing $\arg\max_{\mathbf{z}_{1:T}} \mathcal{P}(\mathbf{z}_{1:T}|\mathbf{x}_{1:T})$, which is a most probable state sequence (for HMM, this can be done by **Viterbi decoding**). Note that all previous 3 inferences give probability estimates over states, whereas this gives a sequence of states.

- The **forwards algorithm** (MLPP 17.4.2) where we compute the filtered marginals $\mathcal{P}(z_t|\mathbf{x}_{1:t})$ in an HMM. In the first step we compute the **one-step-ahead predictive density**:

$$\mathcal{P}(z_t = j|\mathbf{x}_{1:t-1}) = \sum_i \mathcal{P}(z_t = j|z_{t-1} = i)\mathcal{P}(z_{t-1} = i|\mathbf{x}_{1:t-1}) \tag{16.14}$$

In the update step, we absorb the observed data from time $t$ using Bayes rule:

$$\alpha_t(j) \triangleq \mathcal{P}(z_t = j|\mathbf{x}_{1:t}) = \mathcal{P}(z_t = j|\mathbf{x}_t, \mathbf{x}_{1:t-1}) \tag{16.15}$$

$$= \frac{1}{Z_t}\mathcal{P}(\mathbf{x}_t|z_t = j, \cancel{\mathbf{x}_{1:t-1}})\,\mathcal{P}(z_t = j|\mathbf{x}_{1:t-1}) \tag{16.16}$$

The cancellation is possible because HMMs suppose conditional independence of observations. Where the normalization constant is

$$Z_t \triangleq \mathcal{P}(\mathbf{x}_t|\mathbf{x}_{1:t-1}) = \sum_j \mathcal{P}(\mathbf{x}_t|z_t = j)\mathcal{P}(z_t = j|\mathbf{x}_{1:t-1}) \tag{16.17}$$

This process is known as the **predict-update cycle**. The distribution $\mathcal{P}(z_t|\mathbf{x}_{1:t})$ is called the (filtered) **belief state** at time $t$, and is a vector of $K$ numbers, often denoted by $\boldsymbol{\alpha}_t$.

**Algorithm 5:** Forwards algorithm for HMM

**Input**: Transition matrix $A(i,j) \triangleq \mathcal{P}(z_t = j|z_{t-1} = i)$
**Input**: Local evidence vectors (**emission probability**) $\boldsymbol{\psi}_t(j) \triangleq \mathcal{P}(\mathbf{x}_t|z_t = j)$
**Input**: Initial state distribution $\pi(j) \triangleq \mathcal{P}(z_1 = j)$

$[\alpha_1, Z_1] :=$ **normalize**$(\boldsymbol{\psi}_1 \odot \boldsymbol{\pi})$ // computing initial normalization and belief state
// Recall, $\mathcal{P}(z_1 = j)\mathcal{P}(\mathbf{x}_1|z_1 = j) = \mathcal{P}(\mathbf{x}_1)\mathcal{P}(z_1 = j|\mathbf{x}_1)$

**for** $\tau := 2 : t$ **do**
  $\mathbf{d}_\tau = \mathbf{A}^\top \boldsymbol{\alpha}_{\tau-1}$ // $\mathcal{P}(z_\tau = j|\mathbf{x}_{1:\tau-1})$ which sums over all incoming $z_{\tau-1}$ states
  $[\alpha_\tau, Z_\tau] :=$ **normzalize**$(\boldsymbol{\psi}_\tau \odot \mathbf{d}_\tau)$ // update step

Returns $\boldsymbol{\alpha}_{1:t}$ and $\log \mathcal{P}(\mathbf{x}_{1:t}) = \sum_\tau \log Z_\tau$ // Also return the log probability of the evidence

Subroutine: $[\mathbf{v}, Z] =$ **normalize**$(\mathbf{u})$ : $Z = \sum_j u_j$; $v_j = u_j/Z$;

- The **forwards-backwards algorithm (FB)** (MLPP 17.4.3) is to compute the smoothed marginals $\mathcal{P}(z_t = j|\mathbf{x}_{1:T})$. The key to the algorithm is that you can decompose this into two probabilities:

$$\mathcal{P}(z_t = j|\mathbf{x}_{1:T}) \propto \mathcal{P}(z_t = j, \mathbf{x}_{t+1:T}|\mathbf{x}_{1:t}) \propto \mathcal{P}(z_t = j|\mathbf{x}_{1:t})\mathcal{P}(\mathbf{x}_{t+1:T}|z_t = j, \cancel{\mathbf{x}_{1:T}}) \tag{16.18}$$

We already compute $\alpha_t(j) \triangleq \mathcal{P}(z_t = j|\mathbf{x}_{1:t})$ using the forwards algorithm. We define: $\beta_t(j) \triangleq \mathcal{P}(\mathbf{x}_{t+1:T}|z_t = j)$, which is the conditional likelihood of future evidence given that the hidden state at time $t$ is $j$. We also define $\gamma_j(j) \triangleq \mathcal{P}(z_t = j|\mathbf{x}_{1:T})$, which is the desired output. Of course, $\gamma_j(j) \propto \alpha_t(j)\beta_t(j)$. We go in reverse order: given $\beta_t$ we compute $\beta_{t-1}$:

$$\begin{aligned}
\beta_{t-1}(j) &= \mathcal{P}(\mathbf{x}_{t:T}|z_{t-1} = j) && (16.19)\\
&= \sum_i \mathcal{P}(\mathbf{x}_{t+1:T}|z_t = i)\,\mathcal{P}(\mathbf{x}_t|z_t = i, \cancel{z_{t-1} = j})\,\mathcal{P}(z_t = i|z_{t-1} = j) && (16.20)\\
&= \sum_i \beta_t(i)\,\psi_t(i)\,A(j, i) && (16.21)\\
\boldsymbol{\beta}_{t-1} &= \mathbf{A}(\boldsymbol{\psi}_t \odot \boldsymbol{\beta}_t) && (16.22)
\end{aligned}$$

Where the base case is: $\beta_T(i) = \mathcal{P}(\mathbf{x}_{T+1:T}|z_t = i) = \mathcal{P}(\emptyset|z_t = i) = 1$. Hence, we can adjust the Algorithm 5 to compute backward messages, by just having a loop which goes from $T$ to $t$, and computes $\boldsymbol{\beta}_\tau$ from $\boldsymbol{\beta}_{\tau+1}$ at every step. FB algorithm takes $\mathcal{O}(K^2T)$ time to compute.

- The **Viterbi algorithm** (MLPP 17.4.4) can be used to compute the most probable sequence of states in a chain-structured graphical model i.e. $\mathbf{z}^* = \arg\max_{\mathbf{z}_{1:T}} \mathcal{P}(\mathbf{z}_{1:T}|\mathbf{x}_{1:T})$. This is equivalent to computing a shortest path through the **trellis diagram**, where the nodes are possible states at each time step, and the node and edge weights are log probabilities. That is the weight for the path of states $z_1, \ldots, z_T$ is given by:

$$\log \pi_1(z_1) + \log \psi_1(z_1) + \sum_{t=2}^{T} [\log A(z_{t-1}, z_t) + \log \psi_t(z_t)] \tag{16.23}$$

Note: *The jointly most probably sequence of states (joint MAP) is not necessarily the same as the sequence of marginally most probable states.* The former is given by the Viterbi algorithm, whereas the latter is given by the maximizer of the posterior marginals or **MPM**:

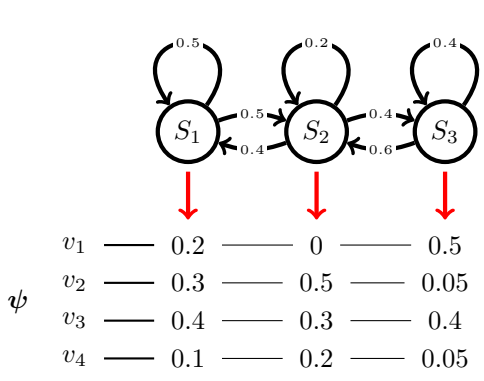
$$\hat{\mathbf{z}} = \left(\arg\max_{z_1} \mathcal{P}(z_1|\mathbf{x}_{1:t}), \ \ldots\ , \arg\max_{z_T} \mathcal{P}(z_T|\mathbf{x}_{1:T})\right) \tag{16.24}$$

Viterbi works well because joint MAP gives a single plausible path between all the states. On the other hand MPM estimates states which are individually most likely given all the data. This is more robust than Viterbi because it estimates each node averaging over its neighbors, rather than conditioning on specific value of its neighbors. Note that in Viterbi, when we estimate $z_t$, we "max-out" the other variables:

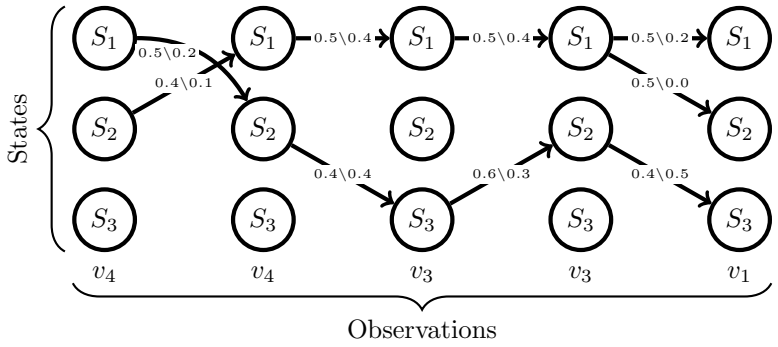
$$z_t^* = \arg\max_{z_t} \max_{\mathbf{z}_{1:t-1}, \mathbf{z}_{t+1:T}} \mathcal{P}(\mathbf{z}_{1:t-1},\, z_t,\, \mathbf{z}_{t+1:T} \,|\, \mathbf{x}_{1:T}) \tag{16.25}$$

- It is tempting to think that we can implement Viterbi by just replacing the sum-operator in the forwards-backwards with a max-operator in Equation 16.14. With the sum operator the algorithm is called **sum-product**, and with the max-operator it is called **max-product**. In general this can lead to incorrect if there are multiple equally probable joint assignments. The Viterbi algorithm uses max-product for the forward pass, but the backward pass uses a traceback procedure to recover the most probably path through the trellis of states. In more detail, we define:

$$\delta_t(j) \triangleq \max_{z_1, \ldots, z_{t-1}} \mathcal{P}(\mathbf{z}_{1:t-1},\, z_t = j \,|\, \mathbf{x}_{1:t}) \tag{16.26}$$




(a) State transition diagram with emission probabilities for $K = 4$ states.




(b) Illustration of the trellis diagram for viterbi algorithm applied to sequence $v_4, v_4, v_3, v_3, v_1$. Each edge from $i$ to $j$ indicates $A(i,j) = \mathcal{P}(z_t = j|z_{t-1} = i)$ followed by $\psi_t(j) = \mathcal{P}(x_t = v_t|z_t = j)$

Figure 12: A viterbi algorithm example for HMMs. Note that we don't incorporate initial state distributions $\boldsymbol{\pi}$ in this example.

This is the probability of ending up in state $j$ at time $t$, given that we take the most probable path. The key insight is that the most probable path to state $j$ at time $t$ must consist of the most probable path to some other state $i$ at time $t-1$, followed by a transition from $i$ to $j$. Hence,

$$\delta_t(j) = \max_i \delta_{t-1}(i)\, A(i,j)\, \psi_t(j) \tag{16.27}$$

We also keep track of the most likely previous state, for each possible state that we end up in. We initialize by setting $\delta_1(j) = \pi_j \psi_1(j)$. The Figure 12a gives an example, for a case where a person sitting behind a curtain has a choice of 3 urns $S_1, S_2, S_3$ to choose from, and each urn has 4 different colored balls $v_1, v_2, v_3, v_4$. After selecting an urn, they randomly pick the ball and display it across the curtain. They do this five times, and the balls displayed are $v_4, v_4, v_3, v_3, v_1$, and now the goal is to choose the urn $z_t$ for each time step. Figure 12b shows the trellis for the Viterbi for getting the MAP solution given these observations. The time complexity for Viterbi is $\mathcal{O}(K^2T)$ and the space complexity is $\mathcal{O}(KT)$.

- Viterbi can be extended to return the top $N$ paths. This is called the **N-best list** (MLPP 17.4.4.5).
- What if you want to learn the parameters for an HMM $\boldsymbol{\theta} = (\boldsymbol{\pi}, \mathbf{A}, \boldsymbol{\psi})$, where $\pi(j) \triangleq \mathcal{P}(z_1 = j)$ is the initial state distribution, $A(i,j) \triangleq \mathcal{P}(z_t = j|z_{t-1} = i)$ is the transition matrix, and $\boldsymbol{\psi}_t(j) \triangleq \mathcal{P}(\mathbf{x}_t|z_t = j)$ is the emission probabilities / local evidence vectors. There are two cases in parameter estimation: (1) when you know $\mathbf{z}_{1:T}$ in the training set; (2) when these states are hidden.
- (MLPP 17.5.1) If we observe the hidden state sequences, we can compute the MLEs for $\mathbf{A}$ and $\boldsymbol{\pi}$ like in Equation 16.5. The details for extracting $\boldsymbol{\psi}$ depends on the observation model in use. The technique is very similar to fitting a generative classifier. For example, if each state has a multinoulli distribution associated with it, with parameters $\psi(j)_l = \mathcal{P}(X_t = l \,|\, z_t = j)$, where $l \in \{1, \ldots, L\}$ represents the observed symbol, then the MLE is given by:

$$\hat{\psi}(j)_l = \frac{N_{jl}^X}{N_j}, \quad N_{jl}^X \triangleq \sum_{i=1}^{N} \sum_{t=1}^{T_i} \mathbb{I}(z_{i,t} = j,\, x_{i,t} = l) \tag{16.28}$$

The result is quite intuitive: we simply add up the number of times we are in state $j$ and we see a symbol $l$, and divide by the number of times we are in state $j$. Analogous results can be derived for other kinds of distributions.

- (MLPP 17.5.2) If we don't observe $z_t$ we are in a situation similar to fitting a mixture model. The most common approach is to use the EM algorithm to find the MLE or MAP parameters. In the E step the expected counts are computed: $\mathbb{E}[N_k^1]$, $\mathbb{E}[N_{jk}]$, and $\mathbb{E}[N_j]$. In the M step, for $\mathbf{A}$ and $\boldsymbol{\pi}$, is just to normalize the expected counts - which is just adding up the expected number of transitions from $j$ to $k$, and dividing by the number of times we transition from $j$ to anything else. For $\boldsymbol{\psi}$, the M step just adds the expected number of times we are in state $j$ and we see symbol $l$, and divide by the expected number of times we are in state $j$.
- Good initialization for EM can be done by using a smaller fully labeled subset (which has no hidden states); random restarts; or ignoring markov dependencies and estimating the observation parameters using the standard mixture model estimation like k-means or EM. These techniques are useful because EM can get stuck in local minima.
- HMMs can be used as the class conditional density inside a generative classifier (MLPP 17.5.4).
- Model selection in HMM is important too:
  - Selecting number of hidden states $K$ is like estimating the number of mixtures in a mixture model (MLPP 17.5.5.1). One way is to do grid search with an objective function which computed cross-validation likelihood, or BIC score. Another way is to use variational Bayes to extinguish unwanted components. Or we can also use an infinite HMM which is based on hierarchical Dirichilet process.
  - Another model to select in HMM is the sparse transition structure in the state transition model (MLPP 17.5.5.2). Most of the method employ hueristics and alternate between parameter estimation and some kind of split-merge method. One can also pose this problem as MAP estimation using a minimum entropy prior.
- Other form of HMMs:
  - In a **semi-Markov model** to predict the next state, you not only need to know the past state, but we also need to know how long we've been in that state (by going over the self-loop transition). When the state space is not observed directly, the result is called a **hidden semi-Markov model (HSMM)** (MLPP 17.6.1).
  - A **hierarchical HMM (HHMM)** (MLPP 17.6.2) is an extension of HMM that is designed to model domains with hierarchical structure.
  - In standard HMM we suppose that observations are conditionally independent given the hidden states. However, if we add arcs from $\mathbf{x}_{t-1}$ to $\mathbf{x}_t$, this is called **auto-regressive HMM (AR-HMM)** (MLPP 17.6.4). This essentially combines two Markov chains, one on the hidden variables, to capture long range dependencies, and one on the observed variables to capture short range dependencies.
  - If we replace $K$ hidden states by binary representation it is called a **factorial HMM** (MLPP 17.6.5). For instance, if we have $2^{10} = 1024$ states, this could be represented by $C = 10$ binary variables $z_{c,t} \in \{0, 1\}$, where now the observation $\mathbf{x}_\tau$ is now dependent on all $z_{c,\tau} : c = \{1, \ldots, C\}$. Here, we will have $C$ chains for the hidden states, which might capture different aspects of the signal.
  - **Dynamic Bayesian Network (DBN)** (MLPP 17.6.7) is just a way to represent a stochastic process using a directed graphical model (dynamic refers to the dynamical nature of the system).

# 17 State Space Models

- A **State Space Model (SSM)** is just like an HMM except that the hidden states are continuous (MLPP 18.1). The model can be written in the following generic form:

$$\mathbf{z}_t = g(\mathbf{z}_{t-1}, \mathbf{u}_t, \boldsymbol{\epsilon}_t) \tag{17.1}$$

$$\mathbf{x}_t = h(\mathbf{z}_t, \mathbf{u}_t, \boldsymbol{\delta}_t) \tag{17.2}$$

where $\mathbf{z}_t$ is the hidden state, $\mathbf{u}_t$ is an optional input or control signal, $\mathbf{x}_t$ is the observation, $g(\cdot)$ is the **transition model / temporal model**, $h(\cdot)$ is the **observation model / measurement model**, $\boldsymbol{\epsilon}_t$ is the system noise at time $t$, and $\boldsymbol{\delta}_t$ is the observation noise at time $t$. We assume that all the parameters of the model, $\boldsymbol{\theta}$, are known; if not, we can include into the hidden state. One of the primary goals in using SSMs is to recursively estimate the belief state:

$$\mathcal{P}(\mathbf{z}_t \,|\, \mathbf{x}_{1:t},\, \mathbf{u}_{1:t},\, \boldsymbol{\theta}) \tag{17.3}$$

We usually write this as $\mathcal{P}(\mathbf{z}_t \,|\, \mathbf{x}_{1:t})$ for brevity. Another important question is how to convert our beliefs about the hidden state into predictions about future observables by computing the posterior predictive $\mathcal{P}(\mathbf{x}_{t+1} \,|\, \mathbf{x}_{1:t})$.

- The observation model describes the relationship between the measurements $\mathbf{x}_t$ and the state $\mathbf{z}_t$ at time $t$ (CVPrince 19.1). We treat this as a generative process, and model the likelihood $\mathcal{P}(\mathbf{x}_t|\mathbf{z}_t)$. We assume that $\mathbf{x}_t$ is conditionally independent of $\mathbf{z}_{1:t-1}$ given $\mathbf{z}_t$.
- The transition model describes the relationship between states. Typically, we make the Markov assumption: we assume that $\mathbf{z}_t$ is conditionally independent of the states $\mathbf{z}_{1:t-2}$ given its immediate predecessor $\mathbf{z}_{t-1}$, and just model the relationship $\mathcal{P}(\mathbf{z}_t|\mathbf{z}_{t-1})$.
- An important special case for SSMs is when all the CPDs are linear-Gaussian. In other words we assume:
  - The transition model is a linear function: $\mathbf{z}_t = \mathbf{A}_t\mathbf{z}_{t-1} + \mathbf{B}_t\mathbf{u}_t + \boldsymbol{\epsilon}_t$
  - The observation model is a linear function: $\mathbf{x}_t = \mathbf{C}_t\mathbf{z}_{t-1} + \mathbf{D}_t\mathbf{u}_t + \boldsymbol{\delta}_t$
  - The system and observation noise are both Gaussian: $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0},\, \mathbf{Q}_t), \quad \boldsymbol{\delta}_t \sim \mathcal{N}(\mathbf{0},\, \mathbf{R}_t)$

  This model is called **Linear Gaussian SSM (LG-SSM)** or a **Linear Dynamical System (LDS)**. If the parameters $\boldsymbol{\theta}_t \triangleq \{\mathbf{A}_t, \mathbf{B}_t, \mathbf{C}_t, \mathbf{D}_t, \mathbf{Q}_t, \mathbf{R}_t\}$ are independent of time, the model is called **stationary**. The LG-SSM is important because it supports exact inference through the famous **Kalman Filter**. In particular if the initial belief state is Gaussian, $\mathcal{P}(\mathbf{z}_1) = \mathcal{N}(\boldsymbol{\mu}_{1|0}, \boldsymbol{\Sigma}_{1|0})$, then all subsequent belief states will also be Gaussian; we will denote them by $\mathcal{P}(\mathbf{z}_t \,|\, \mathbf{x}_{1:t}) = \mathcal{N}(\boldsymbol{\mu}_{t|t}, \boldsymbol{\Sigma}_{t|t})$. The notation denotes $\boldsymbol{\mu}_{t|\tau} \triangleq \mathbb{E}[\mathbf{z}_t|\mathbf{x}_{1:\tau}]$, and similarly for $\boldsymbol{\Sigma}_{t|t}$; thus $\boldsymbol{\mu}_{t|0}$ denotes the prior for $\mathbf{z}_1$ before seeing any data.
- One application of Kalman Filtering is **tracking objects** (MLPP 18.2.1) from noisy measurements (think of tracking aircraft from radar measurements). Given an object with $z_{1t}, z_{2t}$ position and velocity of $\dot{z}_{1t}$, $\dot{z}_{2t}$, we can represent $\mathbb{R}^4$ state as follows:

  $$\mathbf{z}_t^\mathsf{T} = (z_{1t},\, z_{2t},\, \dot{z}_{1t},\, \dot{z}_{2t}) \tag{17.4}$$

  If the object is moving with constant velocity but is perturbed by random Gaussian noise $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{Q})$ (e.g. due to wind), the system dynamics can be written as:

  $$\mathbf{z}_t = \mathbf{A}_t \mathbf{z}_{t-1} + \boldsymbol{\epsilon}_t \tag{17.5}$$

  $$\begin{bmatrix} z_{1,t} \\ z_{2,t} \\ \dot{z}_{1,t} \\ \dot{z}_{2,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta & 0 \\ 0 & 1 & 0 & \Delta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} z_{1,t-1} \\ z_{2,t-1} \\ \dot{z}_{1,t-1} \\ \dot{z}_{2,t-1} \end{bmatrix} + \begin{bmatrix} \epsilon_{1t} \\ \epsilon_{2t} \\ \epsilon_{3t} \\ \epsilon_{4t} \end{bmatrix} \tag{17.6}$$

  where $\Delta$ is the sampling period. Since there is noise added to the velocity, this is called **random accelerations model**. This is like saying that the object moves according to Newton's laws but is subject to random changes in velocity. Now suppose we can observe the location of the object but not the velocity. Let $\mathbf{x}_t \in \mathbb{R}^2$ represent

the observation, which we assume is subject to Gaussian noise $\boldsymbol{\delta}_t \sim \mathcal{N}(\mathbf{0}, \mathbf{R})$. We can model this as:

$$\mathbf{x}_t = \mathbf{C}_t \mathbf{z}_t + \boldsymbol{\delta}_t \tag{17.7}$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \\ \dot{z}_{1,t} \\ \dot{z}_{2,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} z_{1,t} \\ z_{2,t} \\ \dot{z}_{1,t} \\ \dot{z}_{2,t} \end{bmatrix} + \begin{bmatrix} \delta_{1t} \\ \delta_{2t} \\ \delta_{3t} \\ \delta_{4t} \end{bmatrix} \tag{17.8}$$

We can specify our prior (initial) belief about the state of the object, by assuming it Gaussian, $\mathcal{P}(\mathbf{z}_1) = \mathcal{N}(\mathbf{z}_1 \mid \boldsymbol{\mu}_{1|0}, \boldsymbol{\Sigma}_{1|0})$. We can represent our prior ignorance by setting $\boldsymbol{\Sigma}_{1|0} = \infty\mathbf{I}$. We now have fully specified the model and can perform sequential Bayesian updating to compute $\mathcal{P}(\mathbf{z}_t \mid \mathbf{x}_{1:t})$ using the Kalman Filter.

- We can perform *online* Bayesian inference for the parameters of various statistical models using SSMs (MLPP 18.2.3). The basic idea is to let the hidden state represent the regression parameters, and to let the (time-varying) observation model represent the current data vector. In more detail, define the prior to be $\mathcal{P}(\boldsymbol{\theta} \mid \boldsymbol{\theta}_0, \boldsymbol{\Sigma}_0)$ (if we want to do online ML estimate, we can just set $\boldsymbol{\Sigma}_0 = \infty\mathbf{I}$. Let the hidden state be $\mathbf{z}_t = \boldsymbol{\theta}$). Let the hidden state be $\mathbf{z}_t = \boldsymbol{\theta}$; if we assume the regression parameters do not change we can set $\mathbf{A}_t = \mathbf{I}$ and $\mathbf{Q}_t = 0\mathbf{I}$, so:

  $$\mathcal{P}(\boldsymbol{\theta}_t \mid \boldsymbol{\theta}_{t-1}) = \mathcal{N}(\boldsymbol{\theta}_t | \boldsymbol{\theta}_{t-1}, , 0\mathbf{I}) = \delta_{\boldsymbol{\theta}_{t-1}}(\boldsymbol{\theta}_t) \tag{17.9}$$

  Let $\mathbf{C}_t = \mathbf{x}_t^\mathsf{T}$, and $\mathbf{R}_t = \sigma^2$, so the (non-stationary) observation model has the form:

  $$\mathcal{N}(\mathbf{x}_t \mid \mathbf{C}_t \mathbf{z}_t, \mathbf{R}_t) = \mathcal{N}(\mathbf{x}_t \mid \mathbf{x}_t^\mathsf{T} \boldsymbol{\theta}_t, \sigma^2) \tag{17.10}$$

  Applying the Kalman filter to this model provides a way to update our posterior beliefs about the parameters as the data streams in. This is known as the **recursive least squares (RLS)** algorithm.

- **SSM for time series forecasting** (MLPP 18.2.4) where the basic idea is to create a generative model of the data in terms of latent processes, which capture different aspects of the signal. We can then integrate out the hidden variables to compute the posterior predictive of the visibles.

- In inference (i.e. state estimation) of temporal models, the goal is to compute the marginal posterior distribution $\mathcal{P}(\mathbf{z}_t | \mathbf{x}_{1:t})$ over the world state $\mathbf{z}_t$, at time $t$, given all the measurements $\mathbf{x}_{1:t}$ up until this time (CVPrince 19.1.1). To initially compute the posterior distribution given only $\mathbf{x}_1$, our posterior is entirely dependent on this datum:

  $$\mathcal{P}(\mathbf{z}_1 \mid \mathbf{x}_1) = \frac{\mathcal{P}(\mathbf{x}_1 | \mathbf{z}_1)\mathcal{P}(\mathbf{z}_1)}{\int \mathcal{P}(\mathbf{x}_1 | \mathbf{z}_1)\mathcal{P}(\mathbf{z}_1) d\mathbf{z}_1} \tag{17.11}$$

  The distribution $\mathcal{P}(\mathbf{z}_1)$ contains our prior knowledge about the initial state. At time $t = 2$ we will have a second measurement $\mathbf{x}_2$. Our posterior distribution would become:

  $$\mathcal{P}(\mathbf{z}_2 \mid \mathbf{x}_1, \mathbf{x}_2) = \frac{\mathcal{P}(\mathbf{x}_2 | \mathbf{z}_2)\mathcal{P}(\mathbf{z}_2 | \mathbf{x}_1)}{\int \mathcal{P}(\mathbf{x}_2 | \mathbf{z}_2)\mathcal{P}(\mathbf{z}_2 | \mathbf{x}_1) d\mathbf{z}_2} \tag{17.12}$$

  The prior $\mathcal{P}(\mathbf{z}_2 | \mathbf{x}_1)$ is now based on what we have learned from the previous measurement; the possible values of the state at this time depend on our knowledge of what happened at the previous time and how these are affected by the temporal model. Generalizing we have:

  $$\mathcal{P}(\mathbf{z}_t \mid \mathbf{x}_{1:t}) = \frac{\mathcal{P}(\mathbf{x}_t | \mathbf{z}_t)\mathcal{P}(\mathbf{z}_t | \mathbf{x}_{1:t-1})}{\int \mathcal{P}(\mathbf{x}_t | \mathbf{z}_t)\mathcal{P}(\mathbf{z}_t | \mathbf{x}_{1:t-1}) d\mathbf{z}_t} \tag{17.13}$$

  To evaluate this, we must compute $\mathcal{P}(\mathbf{z}_t | \mathbf{x}_{1:t-1})$, which represents our prior knowledge about $\mathbf{z}_t$ before we look at the associated measurement $\mathbf{x}_t$. This prior depends on our knowledge $\mathcal{P}(\mathbf{z}_{t-1} | \mathbf{x}_{1:t-1})$ of the state at the previous time step and the transition model $\mathcal{P}(\mathbf{z}_t | \mathbf{z}_{t-1})$, and is computed recursively as:

  $$\mathcal{P}(\mathbf{z}_t \mid \mathbf{x}_{1:t-1}) = \int \mathcal{P}(\mathbf{z}_t | \mathbf{z}_{t-1})\mathcal{P}(\mathbf{z}_{t-1} | \mathbf{x}_{1:t-1}) d\mathbf{z}_{t-1} \tag{17.14}$$

which is known as the **Chapman-Kolmogorov relation**. Hence, inference consists of two alternating steps: In the **prediction step**, we compute the prior $\mathcal{P}(\mathbf{z}_t|\mathbf{x}_{1:t-1})$ in Equation 17.14. In the **measurement incorporation step**, we combine this prior with the new information from the measurement $\mathbf{x}_t$, as given in Equation 17.13.

- **Kalman Filtering** (CVPrince 19.2) For this section we will ignore the control inputs $\mathbf{u}_t$ and hence $\mathbf{B}$ and $\mathbf{D}$ would be absent from the analysis. Let's define our transition model $\mathbf{z}_t = \mathbf{A}_t\mathbf{z}_{t-1} + \boldsymbol{\epsilon}_t$. This can be written in a probabilistic form:

$$\mathcal{P}(\mathbf{z}_t|\mathbf{z}_{t-1}) \;=\; \mathcal{N}(\mathbf{z}_t \,|\, \mathbf{A}_t\mathbf{z}_{t-1},\, \mathbf{Q}_t) \tag{17.15}$$

The observation model relates the data $\mathbf{x}_t$ at time $t$ to the state $\mathbf{x}_t = \mathbf{C}_t\mathbf{z}_t + \boldsymbol{\delta}_t$. This can also be written in a probabilistic form:

$$\mathcal{P}(\mathbf{x}_t|\mathbf{z}_t) \;=\; \mathcal{N}(\mathbf{x}_t \,|\, \mathbf{C}_t\mathbf{z}_t,\, \mathbf{R}_t) \tag{17.16}$$

Notice that this model is quite similar to the factor analysis model in Equation 11.1, which was the relation between the data and the hidden variable. In the context of the Kalman filter, often the dimension of the state $\mathbf{z}$, $D_z$ is often larger than the dimension of the measurement $\mathbf{x}$, $D_{\mathbf{x}}$. Note that the basic form of both the observation and the transition model is the same. This is chosen by design because it ensures if the marginal posterior $\mathcal{P}(\mathbf{z}_{t-1}|\mathbf{x}_{1:t-1})$ at time $t-1$ was normal, then so is the marginal posterior $\mathcal{P}(\mathbf{z}_t|\mathbf{x}_{1:t})$ at time $t$. Hence, the inference procedure consists of a recursive updating of the means and variances of these normal distributions. We will represent the marginal posterior at time $t$ as:

$$\mathcal{P}(\mathbf{z}_t \,|\, \mathbf{x}_{1:t}) = \mathcal{N}(\mathbf{z}_t \,|\, \boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t) \tag{17.17}$$

- In **prediction step** we compute the prior at time $t$ using the Chapman-Kolmogorov equation:

$$\begin{aligned}
\mathcal{P}(\mathbf{z}_t \,|\, \mathbf{x}_{1:t-1}) &= \int \mathcal{P}(\mathbf{z}_t \,|\, \mathbf{z}_{t-1})\mathcal{P}(\mathbf{z}_{t-1} \,|\, \mathbf{x}_{1:t-1}) d\mathbf{z}_{t-1} && (17.18)\\
&= \int \mathcal{N}(\mathbf{z}_t \,|\, \mathbf{A}_t\mathbf{z}_{t-1},\, \mathbf{Q}_t)\, \mathcal{N}(\mathbf{z}_{t-1} \,|\, \boldsymbol{\mu}_{t-1},\, \boldsymbol{\Sigma}_{t-1})\, d\mathbf{z}_{t-1} && (17.19)\\
&= \mathcal{N}(\mathbf{z}_t \,|\, \mathbf{A}_t\boldsymbol{\mu}_{t-1},\, \mathbf{Q}_t + \mathbf{A}_t\boldsymbol{\Sigma}_{t-1}\mathbf{A}_t^\mathsf{T}) && (17.20)\\
&\triangleq \mathcal{N}(\mathbf{z}_t \,|\, \boldsymbol{\mu}_{t|t-1},\, \boldsymbol{\Sigma}_{t|t-1}) && (17.21)
\end{aligned}$$

- In the **measurement incorporation step** we compute:

$$\begin{aligned}
\mathcal{P}(\mathbf{z}_t \,|\, \mathbf{x}_{1:t}) &= \frac{\mathcal{P}(\mathbf{x}_t|\mathbf{z}_t)\mathcal{P}(\mathbf{z}_t|\mathbf{x}_{1:t-1})}{\mathcal{P}(\mathbf{x}_{1:t})} && (17.22)\\
&= \frac{\mathcal{N}(\mathbf{x}_t \,|\, \mathbf{C}_t\mathbf{z}_t,\, \mathbf{R})\, \mathcal{N}(\mathbf{z}_t \,|\, \boldsymbol{\mu}_{t|t-1},\, \boldsymbol{\Sigma}_{t|t-1})}{\mathcal{P}(\mathbf{x}_{1:t})} && (17.23)\\
&= \mathcal{N}\left( \left( \mathbf{C}_t^\mathsf{T}\mathbf{R}_t^{-1}\mathbf{C}_t + \boldsymbol{\Sigma}_{t|t-1}^{-1} \right)^{-1} \left( \mathbf{C}_t^\mathsf{T}\mathbf{R}_t^{-1}\mathbf{x}_t + \boldsymbol{\Sigma}_{t|t-1}^{-1}\boldsymbol{\mu}_{t|t-1} \right),\, \left( \mathbf{C}_t^\mathsf{T}\mathbf{R}_t^{-1}\mathbf{C}_t + \boldsymbol{\Sigma}_{t|t-1}^{-1} \right)^{-1} \right) && (17.24)\\
&= \mathcal{N}(\mathbf{z}_t \,|\, \boldsymbol{\mu}_t,\, \boldsymbol{\Sigma}_t) && (17.25)
\end{aligned}$$

It can be shown that the mean of the posterior is a weighted sum of the values predicted by the measurements and the prior knowledge, and the covariance is smaller than either. Although the form of this equation is quite messy. Moreover, $\boldsymbol{\mu}_t$ and $\boldsymbol{\Sigma}_t$ contain an inversion that is of size $D_z \times D_z$. If the world state is much higher dimensional than the observed state, then it would be more efficient to reformulate this as an inversion of size $D_{\mathbf{x}} \times D_{\mathbf{x}}$. To this end we define the **Kalman gain** as:

$$\mathbf{K}_t \;=\; \boldsymbol{\Sigma}_{t|t-1}\mathbf{C}_t^\mathsf{T} \left( \mathbf{R}_t + \mathbf{C}_t\boldsymbol{\Sigma}_{t|t-1}\mathbf{C}_t^\mathsf{T} \right)^{-1} \tag{17.26}$$

We will modify the expressions for $\boldsymbol{\mu}_t$ and $\boldsymbol{\Sigma}_t$:

$$\boldsymbol{\mu}_t = \boldsymbol{\mu}_{t|t-1} + \mathbf{K}_t(\mathbf{x}_t - \mathbf{C}_t\boldsymbol{\mu}_{t|t-1}) \tag{17.27}$$

$$\boldsymbol{\Sigma}_t = (\mathbf{I} - \mathbf{K}_t\mathbf{C}_t)\boldsymbol{\Sigma}_{t|t-1} \tag{17.28}$$

In equation 17.27 the expression in the brackets is known as the **innovation**, because it is the difference between the actual measurements $\mathbf{x}_t$ and the predicted measurements $\mathbf{C}_t\boldsymbol{\mu}_{t|t-1}$ based on the prior estimate of the state. It is easy to see why $\mathbf{K}$ is termed the Kalman gain: it determines the amount that the measurements contribute to the new estimate in each direction in state space. If the Kalman gain is small in a given direction, then this implies that the measurements are unreliable relative to the prior and should not influence the mean of the state too much. If the Kalman gain is large in a given direction, then this suggests that the measurements are more reliable than the prior and should be weighted highly (CVPrince 19.2.2).

The interpretation of Equation 17.28 is also clear: the posterior covariance is equal to the prior covariance less a term that depends on the Kalman gain: we are always more certain about the state after incorporating information due to the measurement, and the Kalman gain modifies how much more certain we are. When measurements are more reliable the Kalman gain is high, and the covariance decreases more.

- In an offline setting we can look at all the data and then make predictions: $\mathcal{P}(\mathbf{z}_t \mid \mathbf{x}_{1:T})$. This is called **Kalman smoothing** algorithm or RTS smoother (MLPP 18.3.2). By conditioning on the past and future data, our uncertainty will be significantly reduced. The algorithm is quite similar to forwards-backwards algorithm for HMMs. Kalman filtering can be regarded as message passing on a graph from left to right. When the messages have reached the end of the graph, we have successfully computed $\mathcal{P}(\mathbf{z}_T|\mathbf{x}_{1:T})$. Now we work backwards, from right to left, sending information from the future back to the past and then combining the two information sources. We do this by having a backwards Kalman Gain matrix $\mathbf{J}_t$:

$$\mathcal{P}(\mathbf{z}_t \mid \mathbf{x}_{1:T}) = \mathcal{N}(\boldsymbol{\mu}_{t|T}, \boldsymbol{\Sigma}_{t|T}) \tag{17.29}$$

$$\boldsymbol{\mu}_{t|T} = \boldsymbol{\mu}_{t|t} + \mathbf{J}_t(\boldsymbol{\mu}_{t+1|T} - \boldsymbol{\mu}_{t+1|t}) \tag{17.30}$$

$$\boldsymbol{\Sigma}_{t|T} = \boldsymbol{\Sigma}_{t|t} + \mathbf{J}_t(\boldsymbol{\Sigma}_{t+1|T} - \boldsymbol{\Sigma}_{t+1|t})\mathbf{J}_t^\mathsf{T} \tag{17.31}$$

$$\mathbf{J}_t \triangleq \boldsymbol{\Sigma}_{t|t}\mathbf{A}_{t+1}^\mathsf{T}\boldsymbol{\Sigma}_{t+1|t}^{-1} \tag{17.32}$$

The algorithm can be initialized from $\boldsymbol{\mu}_{T|T}$ and $\boldsymbol{\Sigma}_{T|T}$ from the Kalman filter. Note that this backward pass does not need access to the observations $\mathbf{x}_{1:T}$, saving us some memory.

- Another smoothing is fixed lag smoothing (CVPrince 19.2.6), $\mathcal{P}(\mathbf{z}_{t-\tau} \mid \mathbf{x}_{1:t})$. Basically the state vector will be $[\mathbf{w}_{t-\tau}, \ldots, \mathbf{w}_t]^\mathsf{T}$, which fits the original Kalman filter form.
- **Learning for LG-SSM** (MLPP 18.4.1) here we discuss how to learn $\mathbf{A}$ and $\mathbf{C}$. In this case we can set $\mathbf{Q} = \mathbf{I}$ without loss of generality, since an arbitrary noise covariance can be modeled appropriately by changing $\mathbf{A}$. We can also require $\mathbf{R}$ to be diagonal w.l.o.g.. Doing this increases numerical stability and reduces the number of free parameters. Another constraint we can impose is that the eigenvalues of $\mathbf{A}$ be less than 1. This is important because in $\mathbf{z}_t = \mathbf{A}^t\mathbf{z}_1$ for large $t$, $\mathbf{z}_t$ will blow up in magnitude.

  If we observe the hidden state sequences we can fit the model by computing the MLEs for the parameters by solving a multivariate linear regression problem for $\mathbf{z}_{t+1} \to \mathbf{z}_t$ and for $\mathbf{z}_t \to \mathbf{y}_t$. That is, we can estimate $\mathbf{A}$ by solving the least squares problem $J(\mathbf{A}) = \sum_{t=1}^{2}(\mathbf{z}_t - \mathbf{A}\mathbf{z}_{t-1})^2$, and similarly for $\mathbf{C}$. We can estimate the system noise covariance $\mathbf{Q}$ from the residuals in predicting $\mathbf{z}_t$ from $\mathbf{z}_{t-1}$, and estimate the observation noise covariance $\mathbf{R}$ from the residuals in predicting $\mathbf{y}_t$ from $\mathbf{z}_t$.

  If we only observe the output sequence, we can compute ML or MAP estimates of the parameters using EM.
- There are two notable limitations of Kalman filter: firstly it requires the transition and the observation model to be linear; secondly the marginal posterior $\mathcal{P}(\mathbf{z}_t \mid \mathbf{x}_{1:t})$ is unimodal and can be well captured by a mean and covariance; hence, it can only ever have one hypothesis about the position of the object (CVPrince 19.2.8). The solution for former is given by EKF or UKF and the latter can be solved by particle filtering.

- The **Extended Kalman Filter (EKF)** (MLPP 18.5.1) (CVPrince 19.3) is designed to cope with more general transition models, where the relationship between the states at time $t$ is an arbitrary nonlinear function $g(\cdot)$ of the state of the previous time step and a stochastic contribution $\boldsymbol{\epsilon}_t$:

$$\mathbf{z}_t = g(\mathbf{z}_{t-1}) + \mathcal{N}(\mathbf{0}, \mathbf{Q}_t) \tag{17.33}$$

  where $\mathbf{Q}_t$ is the covariance of the noise term $\boldsymbol{\epsilon}_t$ as before. Similarly, it can cope with a nonlinear relationship $h(\cdot)$ between the state and the measurements:

$$\mathbf{x}_t = h(\mathbf{z}_t) + \mathcal{N}(\mathbf{0}, \mathbf{R}_t) \tag{17.34}$$

  where $\mathbf{R}_t$ is the covariance of the noise term $\boldsymbol{\delta}_t$ as before. $g(\cdot)$ and $h(\cdot)$ are nonlinear but differentiable functions. We linearlize these functions about the previous state estimate using a first order Taylor series expansion and then applying the standard Kalman filter equations. Thus we approximate the stationary non-linear dynamical system with a non-stationary linear dynamical system. There are two cases when EKF works poorly. The first is when the prior covariance is large. In this case, the prior distribution is broad, so we end up spending a lot of probability mass through different parts of the function that are far from the mean, where the function has been linearized. The other setting where the EKF works poorly is when the function is highly nonlinear near the current mean.

- The **unscented Kalman filter (UKF)** (MLPP 18.5.2) (CVPrince 19.4) is a derivative-free approach to better handle non-linear function. The key intuition is this: it is easier to approximate a Gaussian than to approximate function. So instead of performing a linear approximation to the function, and passing a Gaussian through it, instead pass a deterministically chosen set of points, known as **sigma points**, through the function, and fit a Gaussian to the resulting transformed points. This is known as the **unscented transform**. The UKF uses the unscented transform twice, once to approximate passing through the transition model $g$ and once to approximate passing through the measurement model $h$. As for the EKF, the predicted state $\mathcal{P}(\mathbf{z}_t|\mathbf{x}_{1:t-1})$ in the UKF is a normal distribution. However, this normal distribution is a provably better approximation to the true distribution than that provided by the EKF.

- See Section 22 for Particle Filtering.

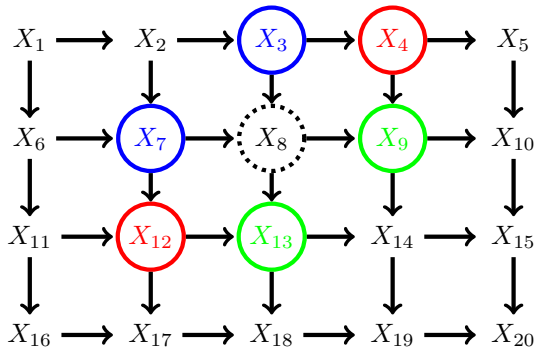
- Many systems contain both discrete and continuous latent variables; these known as **hybrid systems** (MLPP 18.6). For example, the discrete variable may indicate whether a measurement sensor is faulty or not, or which "regime" the system is in. A special case of a hybrid system is when we combine an HMM and an LG-SSM. This is called a **switching linear dynamical system (SLDS)**, a **jump Markov linear system (JMLS)**. More precisely, we have a discrete latent variable, $q_t \in \{1, \ldots, K\}$, a continuous latent variable, $\mathbf{z}_t \in \mathbb{R}^L$, a continuous observed response $\mathbf{x}_t \in \mathbb{R}^D$. We then assume that the continuous variables have linear Gaussian CPDs, conditional on the discrete cases.

  Unfortunately inference (i.e. state estimation) in hybrid models, including the switching LG-SSM model is intractable. This is because of the dependency of $\mathbf{z}_t$ on the discrete variable $q_t$. Various approximate inference methods have been proposed: (1) prune off low probability trajectories in the discrete tree; which is the basis of **multiple hypothesis tracking**; (2) Use monte carlo sampling where we sample discrete trajectories, and apply an analytical filter to the continuous variables conditional on a trajectory.

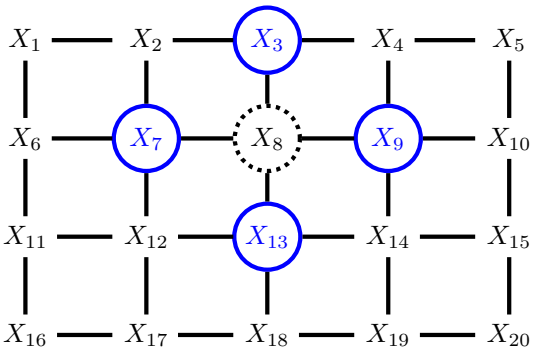
- **Data association** and **multi-target tracking** as an application of hybrid SSM at (MLPP 18.6.2).

# 18 Undirected Graphical Models (Markov Random Fields)

- (MLPP 19.1) In some domains, choosing the direction of dependence doesn't make intuitive sense - for instance if a graph is based on the pixel on an image, you might want to enforce correlation between two neighboring pixels - the Markov blanket becomes un-natural (see Figure 13a). An alternative is to use an **undirected**




(a) A 2d lattice given as a DAG. $X_8$ is independent of all other (black) nodes given its Markov blanket, which includes its parents (blue), children (green), and co-parents (red).




(b) Same model as left given as a UGM. The dotted $X_8$ node is independent of the other black nodes given its neighbors (blue).

Figure 13

**graphical model (UGM)**, also called a **Markov Random Field (MRF)** or **Markov Network**. In this case we don't need to specify the direction of the edges, which is more natural for some problems like image analysis and spatial statistics. An example is given in Figure 13b; where the Markov blanket for each node is just its nearest neighbors - which in case of $X_8$ are $\{X_3, X_7, X_9, X_{13}\}$.

- The main advantages of a UGM over a DGM are: (1) the model is symmetric because of no directional edges - which is useful for spatial problems; (2) discriminative UGM (Conditional Random Fields), which define the conditional densities of the form $\mathcal{P}(\mathbf{y}|\mathbf{x})$, work better than discriminative DGMs.
- The main disadvantage of a UGM over a DGM are: (1) the parameters are less interpretable and less modular; (2) parameter estimation is more expensive.
- UGMs define conditional-independence (CI) relationships via simple graph separation as follows: for sets of nodes $A$, $B$, and $C$, we say $\mathbf{x}_A \perp_G \mathbf{x}_B \,|\, \mathbf{x}_C$ iff $C$ separates $A$ from $B$ in graph $G$. This means when we remove all the nodes in $C$, if there are no paths connecting any node in $A$ to any node in $B$, then the CI property holds. This called the **global Markov property** for UGMs (MLPP 19.2.1). An example is in Figure 14b where we have $\{1, 2\} \perp \{6, 7\} \,|\, \{3, 4, 5\}$.
- The smallest set of nodes that renders a node $t$ conditionally independent of all the other nodes in the graph is called $t$'s **Markov blanket**; we will denote this as $\mathsf{mb}(t)$. Formally the Markov blanket satisfies the following property:

$$t \perp \mathcal{V} \backslash \mathsf{cl}(t) \,|\, \mathsf{mb}(t) \tag{18.1}$$

where $\mathsf{cl}(t) \triangleq \mathsf{mb}(t) \cup \{t\}$ is the **closure** of node $t$. It can be shown that in a UGM, a node's Markov blanket is its set of immediate neighbors. This is called the **undirected local Markov property**. For instance in Figure 14b, we have $\mathsf{mb}(5) = \{2, 3, 4, 6, 7\}$. From the local property, we can easily see that two nodes are conditionally independent given the rest if there is no direct edge between them. This is called the **pairwise Markov property**. In symbols this is written as:

$$s \perp t \,|\, \mathcal{V} \backslash \{s, t\} \iff (s, t) \notin E \tag{18.2}$$

Using the three Markov properities we can derive the following CI properties (amongst others) from the UGM in Figure 14b:

  - **Pairwise**: $1 \perp 7 \,|\, \text{rest}$

- **Local**: $1 \perp \text{rest} \,|\, \{2,3\}$
- **Global**: $\{1,2\} \perp \{6,7\} \,|\, \{3,4,5\}$

It is obvious that the global Markov implies local Markov which implies pairwise Markov. What is less obvious is that pairwise implies global if $\mathcal{P}(x) > 0$, and hence all these Markov properties are essentially the same. The importance of this is that it empirically is more easier to assess pairwise CI; which can be used to construct a graph from which global CI statements can be extracted.

- How do we determine CI relationships for a DGM using a UGM (MLPP 19.2.2)? It is tempting to simply convert the DGM to a UGM by dropping the orientation of the edges, but this is clearly incorrect, since a v-structure $A \to B \leftarrow C$ has quite different CI properties than the corresponding undirected chain $A - B - C$. The latter graph incorrectly states that $A \perp C \,|\, B$. To avoid such incorrect CI statements, we can add edges between the "unmarried" parents $A$ and $C$, and then drop the arrows from the edges, forming (in this case) a fully connected undirected graph. This process is called **moralization**. Figure 14b gives an example where we had to interconnect 2 and 3 because they had a common child, and we had to interconnect 4, 5, and 6 because they had a common child 7.

- Unfortunately moralization loses some CI information, and therefore we cannot reconstruct the original DGM from a moralized UGM. For instance in Figure 14a, using d-separation we see that $4 \perp 5 \,|\, 2$; but adding a moralization arc $4 - 5$ loses this fact. Note that this was because we needed to insert arc $4 - 5$ because of the common child 7. This suggests another technique to determine $A \perp B \,|\, C$. We first form an **ancestral graph** $G$ with respect to $U \triangleq A \cup B \cup C$ - which means $G$ has only all the nodes in $U$ and all the ancestors of $U$. We then moralize this ancestral graph, and apply the simple graph separation rules for UGMs. For Figure 14a, if $U = \{2,4,5\}$, the graph $G$ only has nodes $1,2,3,4,5$, which if moralized will show that $4 \perp 5 \,|\, 2$.

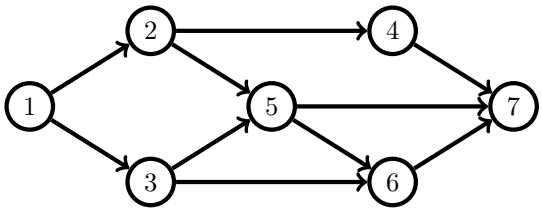
- Let's consider a **perfect map** of $p$ $I(G) = I(p)$ (see Section 9) i.e. $G$ exactly makes all (and only) the CI assumptions as those given by $p$. The natural question is that are DGMs or UGMs are more powerful in representing CI assumptions. It turns out that both DGMs and UGMs are perfect maps for different sets of distributions. As an example for CI relationships which can be represented by DGM but not by UGM is of the v-structure $A \to C \leftarrow B$. This asserts that $A \perp B$, and $A \not\perp B \,|\, C$. Converting this to a UGM with $A - C - B$, it asserts $A \perp B \,|\, C$ and $A \not\perp B$ which is incorrect. In fact there is no UGM which can precisely represent all and only the two CI statements encoded in the v-structure. In general, CI properties in a UGMs are monotonic, in the following sense: if $A \perp B \,|\, C$ then $A \perp B \,|\, (C \cup D)$. But, in DGMs, CI properties can be non-monotonic, since conditioning on extra variables can eliminate conditional independencies due to explaining away.

  As an example of some CI relationships that can be perfectly modeled by a UGM but not by a DGM, is a 4-cycle graph $\{(A - B), (B - C), (C - D), (D, A)\}$. There is no DGM which can represent all (and only) CI statements encoded by this UGM.

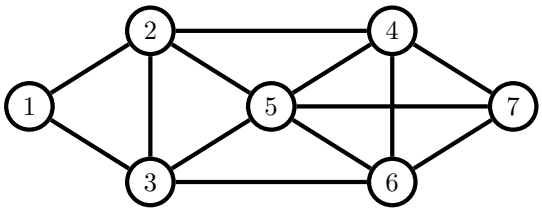
  Some distributions can be perfectly mapped to either UGMs or DGMs - these resulting graphs are called **decomposable** or **chordal**. Roughly speaking this means means that if we collapse together all the variables in each maximal clique, to make "mega-variables," the resulting graph will be a tree.

- Although the CI properties of UGM are simpler than DGMs, representing the joint distribution of a UGM is less natural than a DGM (MLPP 19.3).

- (MLPP 19.3.1) Since there is no topological ordering in a UGM, we can't use the chain rule to represent $\mathcal{P}(\mathbf{y})$. So instead of associating CPDs with each node, we associate **potential functions** or **factors** with each maximal clique in the graph. We will denote the potential function for clique $c$ by $\psi_c(\mathbf{y}_c|\boldsymbol{\theta}_c)$. The potential function can be any non-negative function of its arguments. *The joint distribution is then defined to be proportional to the product of clique potentials.* Rather surprisingly, one can show that any positive distribution whose CI properties can be represented by a UGM can be represented in this way. This theorem follows:




(a) A DGM




(b) Moralized version of Figure 14a represented as a UGM.

Figure 14

**Theorem 18.1.** Hammersley-Clifford Theorem*: A positive distribution $\mathcal{P}(\mathbf{y}) > 0$ satisfies the CI properties of an undirected graph $G$ iff $p$ can be represented as a product of factors, one per maximal clique i.e.:*

$$\mathcal{P}(\mathbf{y} \mid \boldsymbol{\theta}) \;=\; \frac{1}{Z(\boldsymbol{\theta})} \prod_{c \in \mathcal{C}} \psi_c(\mathbf{y}_c | \boldsymbol{\theta}_c) \tag{18.3}$$

*where $\mathcal{C}$ is the set of all maximal cliques of $G$, and $Z(\boldsymbol{\theta})$ is the* ***partition function*** *given by:*

$$Z(\boldsymbol{\theta}) \;\triangleq\; \sum_{\mathbf{y}} \prod_{c \in \mathcal{C}} \psi_c(\mathbf{y}_c | \boldsymbol{\theta}_c) \tag{18.4}$$

*Note that the partition function is what ensures that the overall distribution sums to 1.*

For instance for the MRF given in Figure 14b, if $p$ satisfies the CI properties of this graph then we can write $p$ as follows:

$$\mathcal{P}(\mathbf{y} \mid \boldsymbol{\theta}) \;=\; \frac{1}{z(\boldsymbol{\theta})} \psi_{123}(y_1, y_2, y_3)\, \psi_{235}(y_2, y_3, y_5)\, \psi_{245}(y_2, y_4, y_5)\, \psi_{356}(y_3, y_5, y_6)\, \psi_{4567}(y_4, y_5, y_6, y_7) \tag{18.5}$$

- We can equivalently write Equation 18.3 as a model known as the **Gibbs distribution** which is as follows:

$$\mathcal{P}(\mathbf{y} | \boldsymbol{\theta}) \;=\; \frac{1}{Z(\boldsymbol{\theta})} \exp\left( - \sum_{c \in \mathcal{C}} E(\mathbf{y}_c | \boldsymbol{\theta}_c) \right) \tag{18.6}$$

where $E(\mathbf{y}_c) > 0$ is the energy associated with the variables in clique $c$. $E(\cdot)$ is also sometimes known as the cost function. We can convert this to a UGM by defining:

$$\psi_c(\mathbf{y}_c | \boldsymbol{\theta}_c) \;=\; \exp\left(-E(\mathbf{y}_c | \boldsymbol{\theta}_c)\right) \tag{18.7}$$

Here high probability states correspond to low energy configurations. Models of this form are known as **energy based models**, and are commonly used in physics and biochemistry.

- Note that we are free to restrict the parameterization to the edges of the graph, rather than the maximal cliques. This is called **pairwise MRF**. For Figure 14b, we get:

$$\begin{aligned} \mathcal{P}(\mathbf{y} \mid \boldsymbol{\theta}) \;&=\; \frac{1}{z(\boldsymbol{\theta})} \psi_{12}(y_1, y_2)\, \psi_{13}(y_1, y_3)\, \psi_{23}(y_2, y_3)\; \psi_{25}(y_2, y_5)\, \psi_{35}(y_3, y_5) \\ &\qquad \psi_{24}(y_2, y_4)\, \psi_{25}(y_2, y_5)\, \psi_{45}(y_4, y_5)\; \psi_{35}(y_3, y_5)\, \psi_{36}(y_3, y_6)\, \psi_{56}(y_5, y_6) \\ &\qquad \psi_{45}(y_4, y_5)\, \psi_{46}(y_4, y_6)\, \psi_{47}(y_4, y_7)\, \psi_{56}(y_5, y_6)\, \psi_{57}(y_5, y_7)\, \psi_{67}(y_6, y_7) \qquad (18.8) \\ &\propto \prod_{s \sim t} \psi_{st}(y_s, y_t) \qquad (18.9) \end{aligned}$$

This form is widely used because of its simplicity - but it does not generalize to all cases.

- If the variables are discrete, we can represent the potential or energy functions as tables of (non-negative) numbers, just as we did with CPTs. However, the potentials are not probabilities. Rather, they represent the relative "compatibility" between the different assignments to the potential. A more general approach is to define the log potentials as a linear function for the parameters (using a Gibbs distribution):

$$\log \psi_c(\mathbf{y}_c|\boldsymbol{\theta}_c) \triangleq \boldsymbol{\phi}_c(\mathbf{y}_c)^\mathsf{T}\boldsymbol{\theta}_c \tag{18.10}$$

  where $\phi_c(\mathbf{y}_c)$ is a feature vector derived from the values of the variables $\mathbf{y}_c$. The resulting log probability has the form:

$$\log \mathcal{P}(\mathbf{y}|\boldsymbol{\theta}) = \sum_c \boldsymbol{\phi}_c(\mathbf{y}_c)^\mathsf{T}\boldsymbol{\theta}_c - \log Z(\boldsymbol{\theta}) \tag{18.11}$$

  This is also known as a **maximum entropy** or a **log-linear** model.

  For example if we consider an MRF where the number of states are $K$, then for each edge, we associate a feature vector of length $K^2$ as follows:

$$\boldsymbol{\phi}_{st} = [\dots, \mathbb{I}(y_s = j, y_t = k), \dots] \tag{18.12}$$

  If we have a weight for each feature, we can convert this into a $K \times K$ potential function as follows:

$$\psi_{st}(y_s = j,\, y_t = k) = \exp\left(\left[\boldsymbol{\theta}_{st}^\mathsf{T}\boldsymbol{\phi}_{st}\right]_{jk}\right) = \exp(\theta_{st}(j,k)) \tag{18.13}$$

  So we see that we can easily represent tabular potentials using a log-linear form.

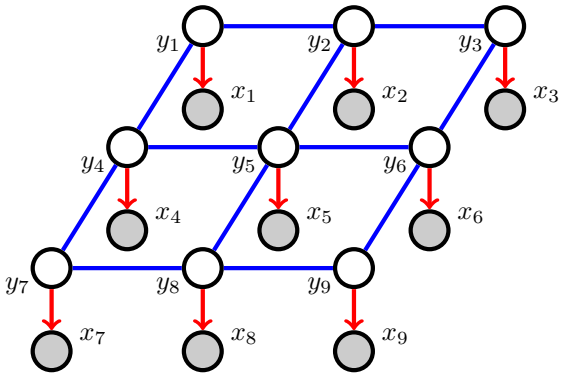
- Examples of MRFs:
  - The **Ising Model** (MLPP 19.4.1) is an example of an binary MRF that arose from statistical physics. We create a 2D or 3D lattice of the form given in Figure 13b. We define the following pairwise clique potential:

$$\psi_{st}(y_s, y_t) = \begin{bmatrix} e^{w_{st}} & e^{-w_{st}} \\ e^{-w_{st}} & e^{w_{st}} \end{bmatrix} \tag{18.14}$$

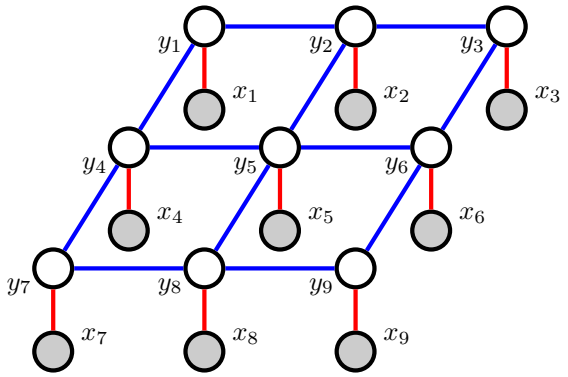
    Here $w_{st}$ is the coupling strength between nodes $s$ and $t$. If two nodes are not connected in the graph, $w_{st} = 0$. Also, we take that weight matrix $\mathbf{W}$ is symmetric, so $w_{st} = w_{ts}$. Often we assume all edges have the same strength, so $w_{st} = J$ (assuming that the edge exists). If all the weights are positive, $J > 0$, the model will encourage neighboring nodes to be in the same state. This is an example of an **associate Markov network**. If the weights are sufficiently strong, the corresponding probability distribution will have two modes, corresponding to the all 1's state and the all 0's state. These are called the **ground states** of the system.

    If all of the weights are negative, $J < 0$, then the states want to be different from their neighbors; and results in a **frustrated system**, in which not all the constraints can be satisfied at the same time. The corresponding probability distribution will have multiple modes. Interestingly, computing the partition function $Z(J)$ can be done in polynomial time for associative Markov networks, but is NP-hard in general.

    There is also an interesting relationship between Ising models and Gaussian graphical models.
  - A **Hopfield network** (MLPP 19.4.2) is a fully connected Ising model with a symmetric weight matrix, $\mathbf{W} = \mathbf{W}^\mathsf{T}$. These weights, can be learned from training data using (approximate) maximum likelihood. The main application of Hopfield networks is as an *associative memory* or *content addressable memory*. The idea is this: suppose we train on a set of fully observed bit vectors, corresponding to patterns we want to memorize. Then, at test time, we present a partial pattern to the network. We would like to estimate the missing variables; this is called *pattern completion*. Since exact inference is intractable in this model, it is standard to use a **coordinate descent** algorithm known as **iterative conditional**




(a) Potts model: A grid-structured MRF with local evidence nodes. Here the measurements / observed data are conditionally dependent on $\mathbf{y}$ (red edges). Note that the variable $y_5$ contributes to the cliques $c_{45}$, $c_{25}$, $c_{56}$, and $c_{58}$.




(b) Conditional Random Field (CRF), where the posterior probability of labels $\mathbf{y}$ is an MRF for fixed data $\mathbf{x}$.

Figure 15

**modes (ICM)**, which just sets each node to its most likely (lowest energy) state, given all its neighbors. A **Boltzmann machine** generalizes the Hopfield / Ising model by including some hidden nodes, which makes the model representationally more powerful. Inference in such models often uses Gibbs sampling, which is a stochastic version of ICM.

– You can generalize the Ising model to multiple discrete states, $y_t \in \{1, 2, \ldots, K\}$. For example if $K = 3$, we use a potential function of the following form:

$$\psi_{st}(y_{st})(y_s, y_t) = \begin{bmatrix} e^{w_{st}} & e^0 & e^0 \\ e^0 & e^{w_{st}} & e^0 \\ e^0 & e^0 & e^{w_{st}} \end{bmatrix} \tag{18.15}$$

This is called the **Potts model** (MLPP 19.4.3). As before, we often assume tied weights of the form $w_{st} = J$. If $J > 0$, then neighboring nodes are encouraged to have the same label. For a small value of $J$ we see many small clusters and for large values we see large clusters. There is a critical value of $J$, for which there is a mix of small and large clusters. This rapid change in the behavior as we vary the parameter is called a *phase transition*.

The potts model can be used as a prior for image segmentation, since it says that neighboring pixels are likely to have the same discrete label and hence belong to the same segment. We can combine this prior with a likelihood term as follows:

$$\mathcal{P}(\mathbf{y}, \mathbf{x} \,|\, \boldsymbol{\theta}) \;=\; \mathcal{P}(\mathbf{y} \,|\, J) \prod_s \mathcal{P}(x_s|y_s, \boldsymbol{\theta}) \;=\; \left[\frac{1}{Z(J)} \prod_{s \sim t} \psi(y_s, y_t;\, J)\right] \prod_s \mathcal{P}(x_s|y_s, \boldsymbol{\theta}) \tag{18.16}$$

where $\mathcal{P}(x_s|y_s = k, \boldsymbol{\theta})$ is the probability of observing pixel $x_s$ given that the corresponding segment belong to class $k$. The corresponding graphical model is a mix of undirected and directed edged as shown in Figure 15a. The undirected 2d lattice represents the prior $\mathcal{P}(\mathbf{y})$; in addition, there are directed edges from $y_s$ to its corresponding $x_s$, representing **local evidence**. Technically speaking this combination of an undirected and directed graph is called a **chain graph**. However, since $x_s$ nodes are observed they can be "absorbed" into the model, thus leaving behind an undirected "backbone." The model is a 2d analog of an HMM and could be a partially observed MRF. As in an HMM, the goal is to perform posterior inference i.e. to compute $\mathcal{P}(\mathbf{y} \,|\, \mathbf{x}, \boldsymbol{\theta})$. Unfortunately the 2d case is provably much harder than the 1d case.

  - Gaussian MRFs (MLPP 19.4.4).
  - **Markov Logic Networks** (MLPP 19.4.5) is a way to combine first order logic with probabilistic measurement - where logic rules are used to define potential functions in an unrolled UGM.

- An important question is whether ML or MAP **parameter estimation for MRFs** can be performed. This happens to be quite computationally expensive. For this reason, it is rare to perform Bayesian inference for the parameters of MRFs.
- **Training maxent models for parameter learning** (MLPP 19.5.1): Consider an MRF in log-linear form:

$$\mathcal{P}(\mathbf{y} \mid \boldsymbol{\theta}) \;=\; \frac{1}{Z(\boldsymbol{\theta})} \exp\left(\sum_c \boldsymbol{\theta}_c^\mathsf{T} \boldsymbol{\phi}_c(\mathbf{y})\right) \tag{18.17}$$

  where $c$ indexes cliques. The scaled log-likelihood is given by:

$$\ell(\boldsymbol{\theta}) \;\triangleq\; \frac{1}{N}\sum_i \log \mathcal{P}(\mathbf{y}_i|\boldsymbol{\theta}) \;=\; \frac{1}{N}\sum_i \left[\sum_c \boldsymbol{\theta}_c^\mathsf{T} \boldsymbol{\phi}_c(\mathbf{y}_i) - \log Z(\boldsymbol{\theta})\right] \tag{18.18}$$

  *Since MRFs are in the exponential family, we know that this function is convex in $\boldsymbol{\theta}$ so it has a unique global maximum which we find using gradient-based optimizers.* In particular the derivative for the weights of a particular clique, $c$, given by:

$$\frac{\partial \ell}{\partial \boldsymbol{\theta}_c} \;=\; \frac{1}{N}\sum_i \left[\boldsymbol{\phi}_c(\mathbf{y}_i) - \frac{\partial}{\partial \boldsymbol{\theta}_c} \log Z(\boldsymbol{\theta})\right] \tag{18.19}$$

  Here, we can show that $\frac{\partial}{\partial \boldsymbol{\theta}_c} \log Z(\boldsymbol{\theta}) = \mathbb{E}[\boldsymbol{\phi}_c(\mathbf{y}) \mid \boldsymbol{\theta}] = \sum_{\mathbf{y}} \phi_c(\mathbf{y}) \mathcal{P}(\mathbf{y} \mid \boldsymbol{\theta})$. This would let us learn parameters when you have observed all the data. There are also methods when you have partially observed data (MLPP 19.5.2).

- When fitting a UGM there is (in general) no closed form solution for the ML or the MAP estimate of the parameters, so we need to use gradient-based optimizers. The gradient requires inference. In models where inference is intractable, learning also becomes intractable. We can combine approximate inference with (gradient based) learning, but results are not always guaranteed to be good depending on which inference algorithm is used. This has motivated various computationallly faster alternatives to ML/MAP estimation, and a few examples are:
  - An alternative to MLE is to maximize the **pseudo likelihood**, which is defined as follows:

$$\ell_{PL}(\boldsymbol{\theta}) \;=\; \frac{1}{N}\sum_{i=1}^{N}\sum_{d=1}^{D} \log \mathcal{P}(y_{id}|\mathbf{y}_{i,-d}, \boldsymbol{\theta}) \tag{18.20}$$

    That is we optimizae the product of the full conditionals, also known as the **composite likelihood**. Compare this to the object for maximum likelihood in Equation 18.18. Here, in the case of Gaussian MRFs, PL is equivalent to ML, but this is not true in general. In the PL approach we learn to predict each node, given all of its neighbors. This objective is generally fast to compute since each full conditional $\mathcal{P}(y_{id}|\mathbf{y}_{i,-d}, \boldsymbol{\theta})$ only requires summing over the states of a single node, $y_{id}$, in order to compute the local normalization constant. The PL approach is similar to fitting each full conditional separately, except that, in PL, the parameters are tied between adjacent nodes.
  - For a fully or partially observed MRF we can approximate model expectations ($\mathbb{E}[\boldsymbol{\phi}_c(\mathbf{y}) \mid \boldsymbol{\theta}]$) using Monte Carlo sampling. We can combine this with stochastic gradient descent (Section 6), which takes samples from the empirical distribution. By some adjustments on where to start MCMC, we can get exact MLE of the parameters modulo MCMC errors. This is called **stochastic maximum likelihood (SML)** (MLPP 19.5.5) .

- MRFs require a good set of features. One unsupervised way to learn such features, known as **feature induction**, is to start with a base set if features, and then to continually create new feature combinations out of old ones, greedily adding the best ones to the model (MLPP 19.5.6).
- **Iterative Proportional Fitting (IPF)** (MLPP 19.5.7) is a fixed point algorithm for enforcing the moment matching constraints and is guaranteed to converge to the global optimum.

- We saw in Equation 18.3 (and in the Potts model - Equation 18.16) that MRF is a generative model. If we now condition on the data (i.e. we assume is fixed), then we can use the relation $\mathcal{P}(\mathbf{y} \,|\, \mathbf{x}) \propto \mathcal{P}(\mathbf{y}, \mathbf{x})$ to write:

$$\mathcal{P}(\mathbf{y} \,|\, \mathbf{x}, \mathbf{w}) \;=\; \frac{1}{Z(\mathbf{x}, \mathbf{w})} \prod_c \psi_c(\mathbf{y}_c \,|\, \mathbf{x}, \mathbf{w}) \tag{18.21}$$

  This is called a **Conditional Random Field (CRF)**. An example can be seen in Figure 15b. There are two sets of cliques in this model: (1) neighboring labels; (2) each label to its associated input / measurement. It can be thought of as a **structured output** extension of logistic regression, where we model the correlation amongst the output labels conditioned on the input features. We will usually assume a log-linear representation of the potentials:

$$\psi_c(\mathbf{y}_c \,|\, \mathbf{x}, \mathbf{w}) \;=\; \exp\left(\mathbf{w}_c^\mathsf{T} \boldsymbol{\phi}(\mathbf{x}, \mathbf{y}_c)\right) \tag{18.22}$$

  where $\boldsymbol{\phi}(\mathbf{x}, \mathbf{y}_c)$ us a feature vector derived from the global inputs $\mathbf{x}$ and the local set of labels $\mathbf{y}_c$.

- The advantages of CRF over MRF: (1) same advantages of discriminative classifiers over generative classifiers - instead of spending resources on modeling things that we always observe, we can focus on modeling what we care about, namely the distribution of labels given the data; (2) we can make the potentials of the model data-dependent - for instance we can "turn-off" the label smoothing between two neighboring nodes $i$ and $j$ if there is an observed discontinuity in the image intensity between $i$ and $j$; (3) The graph structure itself can change depending on the input.

- The disadvantages of CRF over MRF is they require labeled training data, and they are slower to train. This is analogous to the strengths and weaknesses of logistic regression vs naïve Bayes.

- (MLPP 19.6.1) One of the widely used CRFs uses a chain-structured model. But, in the case of HMMs, given in Equation 16.10, the model was written in a generative form:

$$\mathcal{P}(\mathbf{y}_{1:T}, \mathbf{x}_{1:T} \,|\, \mathbf{w}) \;=\; \prod_{t=1}^{T} \mathcal{P}(y_t \,|\, y_{t-1}, \mathbf{w}) \mathcal{P}(x_t \,|\, y_t, \mathbf{w}) \tag{18.23}$$

  where the first term, $\mathcal{P}(y_1)$ is excluded for simplicity. We need to replace this by a discriminative model to be used as an CRF. One way to do this is to reverse the arrows, which would now go from $x_t$ to $y_t$. This defines a directed discriminative model of the form:

$$\mathcal{P}(\mathbf{y}_{1:T} \,|\, \mathbf{x}_{1:T}, \mathbf{w}) \;=\; \prod_t \mathcal{P}(y_t \,|\, y_{t-1}, x_t, \mathbf{w}) \tag{18.24}$$

  If we break $\mathbf{x}$ in a set of local and global features, this is called a **maximum entropy Markov model (MEMM)**. This suffers from **label bias** problem since information cannot flow from $x_t$ back to $y_{t-1}$, because they are v-separated by the v-structure $y_t$. This can be fixed by a discriminative chain-structured undirected CRF, with a model of this form:

$$\mathcal{P}(\mathbf{y} \,|\, \mathbf{x}, \mathbf{w}) \;=\; \frac{1}{Z(\mathbf{x}, \mathbf{w})} \prod_{t=1}^{T} \psi(y_t \,|\, \mathbf{x}, \mathbf{w}) \prod_{t=1}^{T-1} \psi(y_t, y_{t+1} \,|\, \mathbf{x}, \mathbf{w}) \tag{18.25}$$

This no longer has the label bias problem because $y_t$ does not block the information from $x_t$ from reaching to other nodes $y_{t'}$. The label bias problem in MEMMs occurs because directed models **locally normalized**, meaning each CPD sums to 1. By contrast, MRFs and CRFs are **globally normalized**, which means that local factors do not need to sum to 1, since the partition function $Z$, which sums over all joint configurations, will ensure the model defines a valid distribution. However the price we pay is that we do not have a valid distribution over $\mathbf{y}$ until we have seen the whole sequence, because only then we can normalize. Consequently, CRFs are not as useful as DGMs (whether discriminative or generative) for online or real-time inference. Furthermore, the fact that $Z$ depends on all nodes, and hence all their parameters, makes CRFs much slower to train than DGMs.

- **CRF Parameter Training** (MLPP 19.6.3): We can modify the gradient based optimization of MRFs in Section 18 to the CRF case in straightforward way:

$$\ell(\mathbf{w}) \triangleq \frac{1}{N}\sum_i \log \mathcal{P}(\mathbf{y}_i \,|\, \mathbf{x}_i, \mathbf{w}) = \frac{1}{N}\sum_i \left[\sum_c \mathbf{w}_c^\mathsf{T} \boldsymbol{\phi}_c(\mathbf{y}_i, \mathbf{x}_i) - \log Z(\mathbf{w}, \mathbf{x}_i)\right] \tag{18.26}$$

and the gradient becomes:

$$\frac{\partial \ell}{\partial \mathbf{w}_c} = \frac{1}{N}\sum_i \left[\boldsymbol{\phi}_c(\mathbf{y}_i, \mathbf{x}_i) - \frac{\partial}{\partial \mathbf{w}_c}\log Z(\mathbf{w}, \mathbf{x}_i)\right] \tag{18.27}$$

$$= \frac{1}{N}\sum_i \left[\boldsymbol{\phi}_c(\mathbf{y}_i, \mathbf{x}_i) - \mathbb{E}\left[\boldsymbol{\phi}_c(\mathbf{y}_i, \mathbf{x}_i)\right]\right] \tag{18.28}$$

Note that we now have to perform inference for every single training case inside teach gradient step, which is $\mathcal{O}(N)$ times slower than the MRF case. This is because the partition function depends on the inputs $\mathbf{x}_i$.

In most CRFs. the size of the graph structure can vary. Hence we need to use parameter tying to ensure we can define a distribution of arbitrary size. In the pairwise case, we can write the model as follows:

$$\mathcal{P}(\mathbf{y} \,|\, \mathbf{x}, \mathbf{w}) = \frac{1}{Z(\mathbf{w}, \mathbf{x})}\exp(\mathbf{w}^\mathsf{T}\boldsymbol{\phi}(\mathbf{y}, \mathbf{x})) \tag{18.29}$$

where $\mathbf{w} = [\mathbf{w}_n, \mathbf{w}_e]$ are the node and edge parameters, and:

$$\boldsymbol{\phi}(\mathbf{y}, \mathbf{x}) \triangleq \left[\sum_t \boldsymbol{\phi}_t(y_t, \mathbf{x}), \sum_{s\sim t} \boldsymbol{\phi}_{st}(y_s, y_t, \mathbf{x})\right] \tag{18.30}$$

are the summed node and edge features (these are the sufficient statistics). The gradient expression is easily modified to handle this case. In practice it is more important to use a prior/regularization to prevent overfitting. If we use $\ell_1$ regularization for edge weights $\mathbf{w}_e$ to learn sparse graph structure, and $\ell_2$ for the node weights $\mathbf{w}_n$, the likelihood function would become:

$$\ell(\mathbf{w}) \triangleq \frac{1}{N}\sum_i \log \mathcal{P}(\mathbf{y}_i \,|\, \mathbf{x}_i, \mathbf{w}) - \lambda_1\|\mathbf{w}_e\|_1 - \lambda_2\|\mathbf{w}_n\|_2^2 \tag{18.31}$$

Unfortunately, the optimization algorithms are more complicated when we use $\ell_1$ (see Section 12), although the problem is still convex. To handle large datasets, we can use SGD (see Section 6). To handle cases where exact inference is intractable, we can use SML (see Section 18), which combines MCMC inference with SGD parameter learning.

- Training a CRF requires inference, in order to compute the expected sufficient statistics needed to evaluate the gradient. For certain models, computing a joint MAP estimate of the states is provably simpler than computing marginals. Here we discuss a way to train structured output classifiers that leverages the existence

of fast MAP solvers. (to differentiate this from MAP estimation of parameters, we will call MAP estimation of states as **decoding**). These methods are known as **Structural SVM (SSVM)** (MLPP 19.7). Up until now we have looked at fitting models using MAP parameter estimation i.e., by minimizing functions of the form:

$$R_{\text{MAP}}(\mathbf{w}) \;=\; -\log \mathcal{P}(\mathbf{w}) - \sum_{i=1}^{N} \log \mathcal{P}(\mathbf{y}_i \,|\, \mathbf{x}_i,\, \mathbf{w}) \tag{18.32}$$

However at test time, we pick the label so as to minimize the posterior expected loss:

$$\hat{\mathbf{y}}(\mathbf{x} \,|\, \mathbf{w}) \;=\; \arg\min_{\hat{\mathbf{y}}} \sum_{\mathbf{y}} \mathcal{L}(\mathbf{y},\, \hat{\mathbf{y}}) \mathcal{P}(\mathbf{y} \,|\, \mathbf{x},\, \mathbf{w}) \tag{18.33}$$

where $\mathcal{L}(\mathbf{y}^*,\, \hat{\mathbf{y}})$ is the loss we incur when we estimate $\hat{\mathbf{y}}$ but the truth is $\mathbf{y}^*$. It therefore seems reasonable to take the loss function into account when performing parameter estimation. Let us define $L(\mathbf{y}_i,\, \mathbf{y}) = \log \mathcal{L}(\mathbf{y}_i,\, \mathbf{y})$. Since the objective was had to optimize, because the loss is not differentiable, we construct a convex upper bound instead. This gives us SSVM, which minimizes the posterior expected loss on the training set by:

$$R_{\text{SSVM}}(\mathbf{w}) \;=\; \frac{1}{2}\|\mathbf{w}\|^2 \;+\; C \sum_{i=1}^{N} \left[ \max_{\mathbf{y}} \{ L(\mathbf{y}_i,\, \mathbf{y}) \;+\; \mathbf{w}^{\mathsf{T}} \boldsymbol{\phi}(\mathbf{x}_i,\, \mathbf{y}) \} \;-\; \mathbf{w}^{\mathsf{T}} \boldsymbol{\phi}(\mathbf{x}_i,\, \mathbf{y}_i) \right] \tag{18.34}$$

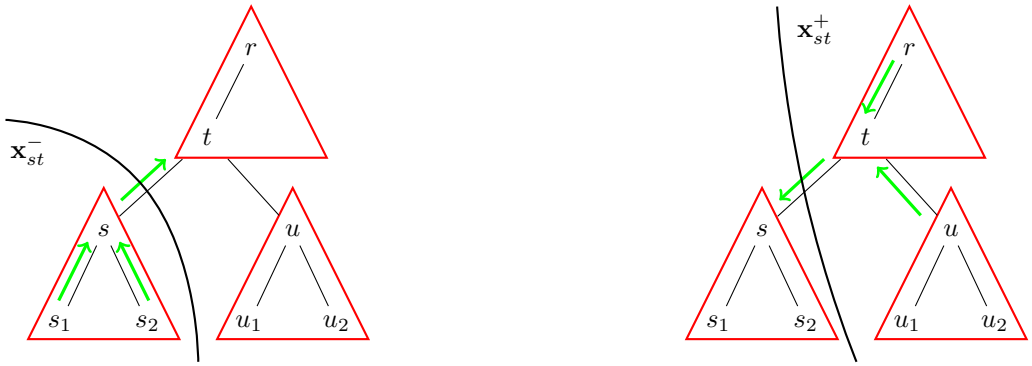
- Even though the SSVM objectives are simple quadratic programs, they have $\mathcal{O}(N|\mathcal{Y})$ constraints. This is intractable, since the domain of $\mathbf{y}$, $\mathcal{Y}$ is usually exponentially large.
- To find the weights $\mathbf{w}$ in SSVM, we use the **cutting plane** algorithm (MLPP 19.7.3). The goal of this algorithm is minimize a convex not-necessarily smooth function $L(\mathbf{w})$. The idea is to incrementally construct a lower approximation $L^{(t)}(\mathbf{w})$. At each iteration, minimize the latter to obtain $\mathbf{w}_t$ and add a cutting plane at that point. We start with an initial guess $\mathbf{w}$ and no constraints. At each iteration, we then do the following: for each example $i$, we find the "most violated" constraint involving $\mathbf{x}_i$ and $\hat{\mathbf{y}}_i$. If the loss-augmented margin violation exceeds the current value of the slack terms $\xi_i$ by more than $\epsilon$, we add $\hat{\mathbf{y}}_i$ to the working set of constraints for the training case $\mathcal{W}_i$, and then solve the resulting QP to find the new $\mathbf{w}$, $\xi$. See Figure (MLPP 19.21). Since at each step we only add one new constraint, we can warm-start the QP solver.

# 19 Exact inference for Graphical Models

- We saw in Section 6 the forward-backwards algorithm can exactly compute the posterior marginals $\mathcal{P}(y_t \,|\, \mathbf{x}_t, \boldsymbol{\theta})$ in any chain-structured graphical model, where $\mathbf{y}$ are the hidden variables (assumed discrete) and $\mathbf{x}$ are the visible variables. This algorithm can be modified to compute posterior mode and posterior samples. In Section 17 we saw Kalman smoothing which is a similar algorithm for linear-Gaussian chains. Our goal now is to generalize these exact inference algorithms for arbitrary graphs. These methods apply to both directed and undirected models.
- **Belief Propagation (BP)** (also called **sum-product algorithm**) (MLPP 20.2.1) is a generalization of forwards-backwards algorithm (which was originally for chains) to trees. We initially assume that the model is a pairwise MRF (or CRF), i.e.:

$$\mathcal{P}(\mathbf{y} \,|\, \mathbf{x}) \;=\; \frac{1}{Z(\mathbf{x})} \prod_{s \in \mathcal{V}} \psi_s(y_s) \prod_{(s,t) \in \mathcal{E}} \psi_{s,t}(y_s, y_t) \tag{19.1}$$

where $\psi_s$ is the local evidence for node $s$, and $\psi_{st}$ is the potential for edge $s - t$. One way to implement BP for undirected trees is as follows. (1) Pick an arbitrary node and call it the root $r$ - and imagine you




(a) Collect-to-root phase (b) Distribute-from-root $r$ phase

Figure 16: Belief Propagation: Message passing on a tree

picked the graph by $r$ and all the nodes are dangling below it - and gives us parent child relationships. (2) We send messages up from the leaves to the root (the **collect evidence phase**, and (3) then send messages back down from the root (the **distribute evidence** phase), in a manner analogous to forwards-backwards on chains. Suppose we want to compute the belief state at node $t$, as illustrated in Figure 16a. We will initially condition the belief only on evidence that is at or below $t$ in the graph i.e., we want to compute $\mathsf{bel}^-_t(y_t) \triangleq \mathcal{P}(y_t|\mathbf{x}^-_t)$, where $\mathbf{x}^-_t$ are all the observations / evidence at or below node $t$ in the tree. We will call this a "bottom-up belief state". Suppose, by induction, that we have computed "messages" from $t$'s two children, summarizing what they think $t$ should know about the evidence in their subtrees, i.e. we have computed $m^-_{s\to t}(y_t) = \mathcal{P}(y_t|\mathbf{x}^-_{st})$ where $\mathbf{x}^-_{st}$ is all the evidence on the downstream side of the $s-t$ edge, and similarly we have computed $m^-_{u\to t}(y_t)$. Then we can compute the bottom-up belief state at $t$ as follows:

$$\mathsf{bel}^-_t(y_t) \triangleq \mathcal{P}(y_t|\mathbf{x}^-_t) = \frac{1}{Z_t}\psi_t(y_t)\prod_{c\in\mathsf{ch}(t)} m^-_{c\to t}(y_t) \tag{19.2}$$

where $\mathsf{ch}(t) = \{s, u\}$ in this case. Also $\psi_t(y_t) \propto \mathcal{P}(y_t|\mathbf{x}_t)$ is proportional to the local evidence for node $t$, and $Z_t$ is the local normalization constant. In words, we multiply all the incoming messages from our children, as well as the incoming message from our local evidence, and then normalize. Now how do we compute the messages $m_{\cdot\to\cdot}(y_t)$ themselves? Consider computing $m^t_{s\to t}(y_t)$, whre $s$ is one $t$'s children. Assume by recursion, that we have already computed $\mathsf{bel}^-_s(y_s)$. Then we can compute the message as follows:

$$m^-_{s\to t}(y_t) = \sum_{y_s}\psi_{st}(y_s, y_t)\,\mathsf{bel}^-_s(y_s) \tag{19.3}$$

Essentially we convert beliefs about $y_s$ into beliefs about $y_t$ by using the edge potential $\psi_{st}$. We continue in this way up the tree until we reach the root. Once at the root, we have "seen" all the evidence in the tree, so we can compute our local belief state at the root $\mathsf{bel}_r(y_r)$. This completes the end of the upwards pass, which is analogous to the forwards pass in an HMM. If we can compute all normalization constants $Z_t$, we can compute the probability of the evidence: $\mathcal{P}(\mathbf{x}) = \prod_t Z_t$.

We can now pass messages down from the root. For example, consider node $s$, with parent $t$, as shown in Figure 16b. To compute the belief state for $s$, we need to combine the bottom-up belief for $s$ together with the a top-down message from $t$, which summarizes all the information in the reset of the graph $m^+_{t\to s}(y_s) \triangleq \mathcal{P}(y_t \,|\, \mathbf{x}^+_{st})$, where $\mathbf{x}^+_{st}$ is all the evidence on the upstream (root) side of the $s-t$ edge, as shown in the figure. We then have:

$$\mathsf{bel}_s(y_s) \triangleq \mathcal{P}(x_s|\mathbf{y}) \propto \mathsf{bel}^-_s(y_s)\prod_{p\in\mathsf{pa}(s)} m^+_{p\to s}(y_s) \tag{19.4}$$

where $\mathsf{pa}(s) = t$ in this case. How do we compute these downward messages? For example, consider the message from $t$ to $s$. Suppose $t$'s parent is $r$, and $t$'s children are $s$ and $u$, as shown in Figure 16b. We want to include in $m^+_{t\to s}$ all the information that $t$ has received, except for information that $s$ sent it:

$$m^+_{t\to s}(y_s) \triangleq \mathcal{P}(y_s|\mathbf{x}^+_{st}) = \sum_{y_t} \psi_{st}(y_s, y_t) \frac{\mathsf{bel}_t(y_t)}{m^-_{s\to t}(y_t)} \tag{19.5}$$

Rather than dividing out the message sent up to $t$, we can plug in the equation of $\mathsf{bel}_t$ to get:

$$m^+_{t\to s}(y_s) = \sum_{y_t} \psi_{st}(y_s, y_t)\, \psi_t(y_t) \prod_{c\in \mathsf{ch}(t)\setminus s} m^-_{c\to t}(y_t) \prod_{p\in\mathsf{pa}(t)} m^+_{p\to t}(y_t) \tag{19.6}$$

In other words, we multiply together all the messages coming into $t$ from all nodes except for the recipient $s$, combine together, and then pass through the edge potential $\psi_{st}$. The summation symbol indicates that we are summing over all the states of of node $y_t$. In the case of a chain, $t$ only has one child $s$ and one parent $p$, and hence both products are replaced simply by $m^+_{p\to t}(y_t)$.

- The version of BP in which we use division is called **belief updating**, which is analogous to Kalman smoothing (see Section 17): the top-down messages $m^+$ depend on the bottom-up messages $m^-$ as well as the filtered belief state $\mathsf{bel}_t$. In the other version where we multiply all-but-one of the messages is called **sum-product**, which is analogous to how we formulated the backwards algorithm (see Section 6), where in case of a chain, the top-down (backward) messages $m^+$ are completely independent of the bottom-up (forward) messages $m^-$, and do not depend on the filtered belief states.

- For a parallel implementation (MLPP 20.2.2) all nodes receive messages from their neighbors in parallel, they then update their belief states, and finally they send new messages back out to their neighbors. The process repeats until convergence. This kind of computing method is called a **systolic array**.

  More precisely, we initialize all messages to all 1's vector. Then, in parallel, each node absorbs messages from all its neighbors using:

$$\mathsf{bel}_t(y_t) \propto \psi_t(y_t) \prod_{s\in\mathsf{nbr}(t)} m_{s\to t}(y_t) \tag{19.7}$$

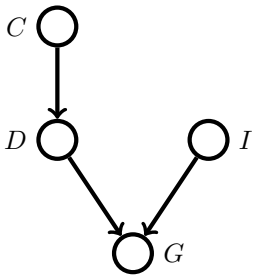
  Then, in parallel, each node sends messages to each of its neighbors:

$$m_{t\to s}(y_s) = \sum_{y_t} \left( \psi_t(y_t)\, \psi_{st}(y_s, y_t) \prod_{u\in\mathsf{nbr}(t)\setminus s} m_{u\to t}(y_t) \right) \tag{19.8}$$

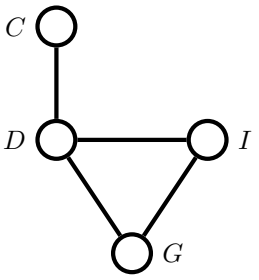
  The $m_{t\to s}$ message is computed by multiplying together all incoming messages, except the one sent by the recipient, and then passing through $\psi_{st}$ potential. See Algorithm 6 for pseudo-code. An iteration $T$ of the algorithm, $\mathsf{bel}_t(y_t)$ represents the posterior belief of $y_s$ conditioned on the evidence that is $T$ steps away in the graph. After $D(G)$ steps, where $D(G)$ is the **diameter**, of the graph (the largest distance between any two pairs of nodes), every node has obtained information from all the other nodes. Its local belief state is then the correct posterior marginal. Since the diameter of a tree is at most $|\mathcal{V}| - 1$, the algorithm converges in linear number of steps.

- Gaussian belief propagation (MLPP 20.2.3).

- (MLPP 20.2.4.1) It is possible to devise a **max-product** version of the BP algorithm by replacing the $\sum$ operator with the max operator. We can then compute the local MAP marginal of each node. However if there are ties, this might not be globally consistent (see Section 6).




(a) Example DGM




(b) Moralized version of Figure 17a

Figure 17

- We have seen how BP can be used to compute exact marginals on chains and trees. But how to compute them on any kind of graph? Here we discuss the **variable elimination (VE) algorithm**.

  The DGM in Figure 17a is given by: $\mathcal{P}(C, D, I, G) = \mathcal{P}(C)\mathcal{P}(D|C)\mathcal{P}(I)\mathcal{P}(G|C, D)$. We can moralize this graph by adding edge $D - I$, and dropping the arrows. Even though this is not a necessary step but it will give a more unified presentation (variable elimination works both on DGMs and UGMs/MRFs). The computational complexity of solving is the same before or after the moralization. Now we define a potential or factor for every CPD, giving $\mathcal{P}(C, D, I, G) = \psi_C(C)\psi_D(D, C)\psi_I(I)\psi_G(G, C, D)$. Since all the potentials are locally normalized, since they are CPDs, there is no need for a global normalization, so $Z = 1$. The corresponding undirected graph is shown in Figure 17b. Considering all variables were binary, what if we wanted to compute $\mathcal{P}(G = 1)$, the marginal probability of having $G = 1$. We could simply sum over the rest of the variables: $\mathcal{P}(G) = \sum_C \sum_D \sum_I \mathcal{P}(C, D, I, G)$. However this takes $\mathcal{O}(2^3)$ time which will grow exponentially with the number of variables. The key idea behind variable elimination is to *push sums inside products*. In our example it would be:

$$\begin{aligned}\mathcal{P}(G) = \sum_{C,D,I} \mathcal{P}(C, D, I, G) &= \sum_{C,D,I} \psi_C(C)\psi_D(D, C)\psi_I(I)\psi_G(G, C, D) \\ &= \sum_C \left( \psi_C(C) \sum_D \left( \psi_D(D, C) \sum_I \left( \psi_I(I)\psi_G(G, C, D) \right) \right) \right)\end{aligned}$$

  We can evaluate this expression right to left. First we multiply together all terms in the scope of a sum and create a temporary factor. Then we marginalize the variable over which we are summing to get a new factor:

$$\begin{aligned}\mathcal{P}(G) &= \sum_C \left( \psi_C(C) \sum_D \left( \psi_D(D, C) \underbrace{\sum_I \left( \psi_I(I)\psi_G(G, C, D) \right)}_{\tau_1(G,C,D)} \right) \right) \\ &= \sum_C \left( \psi_C(C) \sum_D \left( \psi_D(D, C) \underbrace{\sum_I \tau_1'(I, G, C, D)}_{\tau_1(G,C,D)} \right) \right) \qquad \text{replacement with temporary factor} \\ &= \sum_C \left( \psi_C(C) \underbrace{\sum_D \left( \psi_D(D, C)\tau_1(G, C, D) \right)}_{\tau_2(G,C)} \right) \\ &= \sum_C \left( \psi_C(C)\tau_2(G, C) \right) = \tau_3(G)\end{aligned}$$

Furthermore if we need to compute a conditional, we can take a ratio of two marginals, where the visible variables have been clamped to their known values (and hence don't need to be summed over). For instance: $\mathcal{P}(G = g \,|\, I = 1) = \frac{\mathcal{P}(G=g\, I=1)}{\sum_g' \mathcal{P}(G=g'\, I=1)}$. In general:

$$\mathcal{P}(\mathbf{y}_q \,|\, \mathbf{y}_v) = \frac{\sum_{\mathbf{y}_h} \mathcal{P}(\mathbf{y}_h, \mathbf{y}_q, \mathbf{y}_v)}{\sum_{\mathbf{y}_h} \sum_{\mathbf{y}'_q} \mathcal{P}(\mathbf{y}_h, \mathbf{y}'_q, \mathbf{y}_v)} \tag{19.9}$$

The normalization constant in the denominator, $\mathcal{P}(\mathbf{y}_v)$ us called the **probability of the evidence**. If we want to use VE for a MAP estimate $\mathbf{y}^* = \arg\max_{\mathbf{y}} \prod_c \psi_c(\mathbf{y}_c)$, we can just replace the sums with max. Like Viterbi (see Section 6) we need to have a traceback step. In general VE can be applied to any **commutative semi-ring**. This is a set $K$ with two binary operations + and $\times$ with the axioms (1) + is commutative and associative, and $k{+}0 = k$; (2) $\times$ is commutative and associative, and $k{+}1 = k$; and (3) $(a{\times}b){+}(a{\times}c) = a{\times}(b{+}c)$ holds for all triples $(a, b, c) \in K$.

- Note that the running time of VE is exponential in the size of the largest factor, since we have to sum over all of the corresponding variables (MLPP 20.3.2). Hence, the **elimination order**, the order in which we carry out summations, is extremely important. We can see the size of the largest factor graphically. When we eliminate a variable $y_t$, we add a **fill-in edge** between all the variables that share the factor $y_t$ (to reflect the temporary factor $\tau'_t$). The temporary factors generated by VE correspond to maximal cliques in graph $G(\pi)$ where $\pi$ is the elimination ordering. Given the set of maximal cliques $\mathcal{C}$, and $K$ states for each variable, the time complexity of VE is:

$$\sum_{c \in \mathcal{C}(G(\pi))} K^{|c|} \tag{19.10}$$

  Hence we would like to find an elimination ordering where the size of largest clique in the induced graph is the smallest possible (the **treewidth**). The problem of finding the best possible elimination ordering is NP-hard. But in case of chains and trees, we can make an ordering which doesn't induce new edges. For chains we should work forward or backward in time; for trees we should work from leaves to the root. In the case of chains and trees VE takes $\mathcal{O}(VK^2)$ time. This is one reason why Markov chains and trees are used so often. Unfortunately for other graphs, the treewidth can be huge. For instance for a 2D lattice it would be $\mathcal{O}(\min\{m, n\})$, making the running time $\mathcal{O}(K^{\min\{m,n\}})$.

- Another weakness of VE is apparent when you want to compute multiple queries on the same evidence (MLPP 20.3.3). For instance computing all marginals $\{\mathcal{P}(y_1|\mathbf{x}), \ldots, \mathcal{P}(y_T|\mathbf{x})\}$. If we had to do this on a chain structure, the forwards-backwards algorithm will reuse messages computed on the forward pass. In trees, BP would do something similar by reusing messages for compute marginals on one node to compute marginals on another node. VE on the other hand does not reuse such messages, making it slower.

- **Junction Tree Algorithm (JTA)** (MLPP 20.4.1) generalizes BP from trees to arbitrary graphs for exact inference. It works for variables with discrete distributions and Gaussian distributions.

- The basic idea behind JTA is this. WE first run the VE algorithm "symbolically", adding fill-in edges as we go, according to the given elimination ordering. The resulting graph will be a **chordal graph**, which means that every undirected cycle $X_1 - X_2 \cdots X_k - X_1$ of length $k \geq 4$ has a chord i.e. an edge connects $X_i$, $X_j$ for all non-adjacent node $i, j$ in the cycle. The largest loop in a chordal graph is of length 3. For this reason they are sometimes called **triangulated**. Having created a chordal graph, we can extract the maximal cliques. Note that if the original graphical model was already chordal, the elimination process would not add any extra fill-in edges (assuming the optimal elimination ordering was used). We call such models **decomposable** since they break into little pieces defined by the cliques. The cliques of a chordal graph can be arranged into a special kind of tree known as **junction tree**. The nodes of the junction tree contain the variables in a clique, whereas edges contain the variables shared between two cliques. The junction tee enjoys the **running intersection property (RIP)**, which means that any subset of nodes containing a given variable forms a

connected component. *One can show that if a tree that satisfies RIP, then applying BP to this tree (as given below) will return the exact values of $\mathcal{P}(\mathbf{y}_c|\mathbf{x})$ for each node c in the tree (i.e. clique in the induced graph).* From this, we can easily extract the node and edge marginals, $\mathcal{P}(y_t|\mathbf{x})$ and $\mathcal{P}(y_s, y_t|\mathbf{x})$ from the original model, by marginalizing the clique distributions.

- Once the junction tree is constructed we can do inference, in a process similar to BP on a tree (MLPP 20.4.2). Here we will discuss the **belief updating** (also known as the **Hugin**) form of the algorithm. We assume the original model has the form $\mathcal{P}(\mathbf{y}x) = \frac{1}{Z}\prod_{c\in\mathcal{C}(G)}\psi_c(\mathbf{y}_c)$ where $\mathcal{C}(G)$ are the cliques of the original graph. On the other hand the tree defines a distribution of the form:

$$\mathcal{P}(\mathbf{y}) \;=\; \frac{\prod_{c'\in\mathcal{C}(T)}\psi_{c'}(\mathbf{y}_{c'})}{\prod_{s\in\mathcal{S}(T)}\psi_s(\mathbf{y}_s)} \tag{19.11}$$

where $\mathcal{C}(T)$ are nodes of the junction tree (which are the cliques of the chordal graph), and $\mathcal{S}(T)$ are the separators of the tree. To make this equal to the original model, we start by initializing $\psi_s = 1$ and $\psi_{c'} = 1$ for all separators and cliques respectively. Then, for each clique in the original model, $c \in \mathcal{C}(G)$, we find a clique in the tree $c' \in \mathcal{C}(T)$ which contains it, $c \subseteq c'$. To equate the two models we set $\psi_{c'} := \psi_{c'}\psi_c$. After doing this for all cliques in the original graph, we have:

$$\prod_{c'\in\mathcal{C}(T)}\psi_{c'}(\mathbf{y}_{c'}) \;=\; \prod_{c\in\mathcal{C}(G)}\psi_c(\mathbf{y}_c) \tag{19.12}$$

We now send messages from leaves to root and back, as sketched in Figure 16a and 16b. In the collect-to-root phase (Figure 16a), node $i$ sends to its parent $j$ the following message:

$$m_{i\to j}(S_{ij}) = \sum_{C_i\setminus S_{ij}}\psi_i(C_i) \tag{19.13}$$

That is, we marginalize out the variable that node $i$ "knows about" which are irrelevant to $j$, and then we send what is left over. Once a node has received messages from all its children, then it updates its belief state:

$$\psi_j(C_j) \propto \psi_j(C_j)\prod_{i\in\mathsf{ch}(j)} m_{i\to j}(S_{ij}) \tag{19.14}$$

At the root $\psi_r(C_r)$ represents $\mathcal{P}(\mathbf{y}_{C_r}|\mathbf{x})$, which is the posterior over the nodes in the clique $C_r$ conditioned on all the evidence. Its normalization constant is $\mathcal{P}(\mathbf{x})/Z_0$, where $Z_0$ is the normalization constant for the unconditional prior, $\mathcal{P}(\mathbf{y})$. (We have $Z_0 = 1$ if the original model was a DGM).

In the downwards pass, also known as the distribute-from-root phase (Figure 16b), node $i$ sends to its children $j$ the following message:

$$m_{i\to j}(S_{ij}) = \frac{\sum_{C_i\setminus S_{ij}}\psi_i(C_i)}{m_{j\to i}(S_{ij})} \tag{19.15}$$

We divide out by what $j$ sent to $i$ to avoid double counting of evidence - other than that, this message is similar to the upward pass. This requires that we store messages from the upward pass. Once a node has received a top-down message from its parent, it can compute its final belief state using:

$$\psi_j(C_j) \propto \psi_j(C_j)m_{i\to j}(S_{ij}) \tag{19.16}$$

This is exactly the same as updating the belief state in the upward pass, except that we receive only one message i.e. from the parent, because we are working on a tree. Another way of looking at the same process is based on computing messages inside the separator potentials, and updating the recipient potential. This process is known as junction tree **calibration**.

- If all nodes are discrete with $K$ states each, JTA would take $\mathcal{O}(|\mathcal{C}|K^{w+1})$ time and space, where $|\mathcal{C}|$ is the number of cliques and $w$ is the treewidth of the graph, i.e. the size of the largest clique minus 1. As discussed before, choosing a triangulation so as to minimize the treewidth is NP-hard.
- (MLPP 20.5) VE and JTA take time that is exponential in the treewidth of a graph. Since the treewidth can be $\mathcal{O}(|\mathcal{V}|)$ in the worst case, this means these algorithms can be exponential in the problem size. It is easy to show that exact inference is NP-hard. Hence, we need to turn to **approximate inference**.

# 20 Variational Inference

- Given that we are using some distribution which is not discrete or Gaussian, how do we perform inference? We look toward deterministic approximate inference algorithms based on **variational inference** (MLPP 21.1). Here the basic idea is to pick an approximation $q \triangleq \mathcal{Q}(\mathbf{y})$ from some tractable family, and then try to make the approximation as close as possible to the true posterior $p^* \triangleq \mathcal{P}^*(\mathbf{y}) \triangleq \mathcal{P}(\mathbf{y}|\mathcal{D})$, usually by minimizing the KL divergence from $p^*$ to $q$. This reduces the inference to an optimization problem. By relaxing the constraints that $q$ is a proper distribution, and/or by approximating the KL objective function, we can trade accuracy for speed.
- (MLPP 21.2) To make $q$ look "similar" to $p^*$, the intractable true distribution. The obvious cost function to try to minimize is the forward KL divergence (also known as **moment projection**):

$$\mathbb{KL}(p^* \| q) = \sum_{\mathbf{y}} p^*(\mathbf{y}) \log \frac{p^*(\mathbf{y})}{q(\mathbf{y})} \tag{20.1}$$

  But we know this is hard to compute, since taking expectations wrt $p^*$ is assumed to be intractable. A natural alternative is the reverse KL divergence (also called the **information projection**):

$$\mathbb{KL}(q \| p^*) = \sum_{\mathbf{y}} q(\mathbf{y}) \log \frac{q(\mathbf{y})}{p^*(\mathbf{y})} \tag{20.2}$$

  The main advantage here is that computing expectations wrt $q$ is tractable. The former usually over-estimates the support of $p$, while the latter under-estimates it. Unfortunately even the latter form is not tractable, because $p^*(\mathbf{y}) = \mathcal{P}(\mathbf{y}|\mathcal{D})$ pointwise is hard, since it requires evaluating the intractable normalization constant $Z = \mathcal{P}(\mathcal{D})$. However, usually the un-normalized distribution $\tilde{p}(\mathbf{y}) \triangleq \mathcal{P}(\mathbf{y}x, \mathcal{D}) = Zp^*(\mathbf{y})$. is tractable to compute/ We therefore define our new objective function is follows:

$$J(q) \triangleq \mathbb{KL}(q \| \tilde{p}) = \mathbb{KL}(q \| p^*) - \log Z \tag{20.3}$$

  Since $Z$ is constant, by minimizing $J(q)$, we will force $q$ to become close to $p^*$. $J(q)$ is an upper bound on the NLL, which we would like to minimize.
- A popular method for variational inference is called the **mean field approximation** (MLPP 21.3). In this approach, we assume the posterior is a fully factorized approximation of the form:

$$q(\mathbf{y}) = \prod_{i=1}^{D} q_i(\mathbf{y}_i) \tag{20.4}$$

  Our goal is to solve this optimization problem:

$$\min_{q_1,\ldots,q_D} \mathbb{KL}(q \| p) \tag{20.5}$$

  where we optimize over the parameters of each marginal distribution $q_i$. We can solve this by coordinate descent technique. We can show that each step we make the following update:

$$\log q_j(y_j) = \mathbb{E}_{-q_j}[\log \tilde{p}(\mathbf{y})] + \text{const} \tag{20.6}$$

Here $\tilde{p}(\mathbf{y}) \triangleq \mathcal{P}(\mathbf{y}, \mathcal{D})$ is the unnormalized posterior and the notation $\mathbb{E}_{-q_j}[f(\mathbf{y})]$ means to take the expectation over $f(\mathbf{y})$ with respect to all the variables except $y_j$. For examples if we have three variables, then $\mathbb{E}_{-q_2}[f(\mathbf{y})] = \sum_{y_1} \sum_{y_3} q_1(y_1) q_3(y_3) f(y_1, y_2, y_3)$.

When updating $q_j$, we only need to reason about the variables which share a factor with $x_j$, i.e. the terms in $j$'s Markov blanket; the other terms are absorbed into the constant term. Since we are replacing the neighboring values by their mean value, the method is called **mean field**. This is similar to Gibbs sampling, except instead of sending sampled values between neighboring node, we send mean values between node. This tends to be more efficient, since the mean can be a proxy for a large number of samples. In mean field, of course, updating one distribution $q_i$ at a time can be slow, since it is a form of coordinate descent. It is important to note that the mean field method can be used to infer discrete or continuous latent quantities, using a variety of parametric form for $q_i$.

- The assumption that all variables in the posterior are independent of each other is a strong assumption. At times we can exploit substructures in our problem, so we can efficiently handle some kinds of dependencies. This is called **structured mean field** (MLPP 21.4). The approach is same as before, except we group sets of variables together, and update them simultaneously. This follows by treating all variables in the $i$'th group as a single "mega-variable," and then deriving the form for Equation 20.6. As long as we can perform efficient inference in each $q_i$, the method is tractable overall.

- So far we have been concentrating on inferring latent variables $\mathbf{y}_i$ assuming the parameters $\boldsymbol{\theta}$ of the model are known. Now suppose we wanted to infer the parameters themselves. If we make a fully factorized (i.e. mean field) approximation, $p(\boldsymbol{\theta}|\mathcal{D}) \approx \prod_k q(\boldsymbol{\theta}_k)$, we get the method known as **variational Bayes (VB)** (MLPP 21.5). If we want to infer both the latent variables and the parameters, and we make an approximation of the form $p(\boldsymbol{\theta}, \mathbf{y}_{1:N}|\mathcal{D}) \approx q(\boldsymbol{\theta}) \prod_i q_i(\mathbf{y}_i)$, we get a method known as **variational Bayes EM (VBEM)**. Although the math to derive these model gets muddy, the upshot is a method as fast as MAP estimation, while enjoying the statistical benefits of the Bayesian approach. In VB, we are maximizing a lower bound on the log marginal likelihood. (Note, usually variational inference underestimates the posterior uncertainty).

- The principle advantage of VBEM over regular EM is that by marginalizing out the parameters we can compute a lower bound on the marginal likelihood, which can be used for model selection (MLPP 21.6). VBEM is also "egalitarian" in the sense that it treats parameters as "first class citizens," just like any other unknown quantity, whereas EM makes an artificial distinction between parameters and latent variables.

- Methods for getting better lower bounds for distributions (MLPP 21.8.2) / (MLPP 21.8.3).

# 21 More Variational Inference

- (MLPP 22.1) We are still aiming for variational inference, where the basic idea is the same: minimize $J(q) = \mathbb{KL}(q\|\tilde{p})$, where $\tilde{p}$ is the exact but un-normalized posterior as before, but where we no longer require $q$ to be factorized. In fact, we do no even require that $q$ to be a globally valid joint distribution. Instead, we only require that $q$ is locally consistent, meaning that the joint distribution of two adjacent nodes agrees with the corresponding marginals.

- There is a very simple approximate inference algorithm for discrete (or Gaussian) graphical models known as **Loopy Belief Propagation (LBP)** (MLPP 22.2). The basic idea is quite simple: we apply the belief propagation algorithm (see Section 19) to the graph, even if it has loops (i.e. even if it is not a tree). This method is simple and efficient, and often works well in practice, outperforming mean field.

- (MLPP 22.2.1) When there are loops in a graph, BP is not guaranteed to give correct results, and may not even converge. This is because the network is no longer singly connected and local propagation schemes will invariably run into trouble ...if we ignore the existence of loops and permit the nodes to continue communicating with each other as if the network we singly connected, messages may circulate indefinitely around the loops and the process may not converge to a stable equilibrium.

- Applying LBP an undirected graphical model / MRF with pairwise factors is simple. Just repeatedly apply Equations 19.7 and 19.8 until convergence:

**Algorithm 6:** Loopy belief propagation (LBP) for a pairwise MRF

**Input**: Node potentials $\psi_s(y_s)$, edge potentials $\psi_{st}(y_s, y_t)$
Initialize messages $m_{s\to t}(y_t) = 1$ for all edges $s - t$
Initialize beliefs $\mathsf{bel}_s(y_s) = 1$ for all nodes $s$
**repeat**
  Send messages on each edge (Equation 19.8):
    $m_{t\to s}(y_s) \;=\; \sum_{y_t} \left(\psi_t(y_t)\, \psi_{st}(y_s, y_t) \prod_{u \in \mathsf{nbr}(t)\setminus s} m_{u\to t}(y_t)\right)$
  Update belief of each node (Equation 19.7):
    $\mathsf{bel}_t(y_t) \;\propto\; \psi_t(y_t) \prod_{s\in\mathsf{nbr}(t)} m_{s\to t}(y_t)$
**until** *beliefs don't change significantly*
**Output**: Return marginal beliefs $\mathsf{bel}_s(y_s)$

- **Factor Graphs** (MLPP 22.2.3) is a graphical representation which makes it easier to deal with models with higher-order clique potentials (this includes directed models where some nodes have more than one parent). Its representation unifies directed and undirected models, which simplifies certain message passing algorithms. More precisely, a factor graph is an undirected bipartite graph with two kinds of nodes. The two groups are round nodes representing variables, and square nodes representing factors, and there is an edge from each variable to every factor that mentions it. For example, consider the MRF in Figure 18a. If we assume one potential per maximal clique, we get the factor graph in Figure 18b, which represents the function:

$$f(y_1, y_2, y_3, y_4) \;=\; f_{123}(y_1, y_2, y_3)\, f_{234}(y_2, y_3, y_4) \tag{21.1}$$

  Or if we assumer one potential per edge, we get the factor graph in Figure 18c, which represents the function

$$f(y_1, y_2, y_3, y_4) \;=\; f_{12}(y_1, y_2)\, f_{13}(y_1, y_3)\, f_{23}(y_2, y_3)\, f_{24}(y_2, y_4)\, f_{34}(y_3, y_4) \tag{21.2}$$

  We can also convert a DGM to a factor graph: just create one factor per CPD, and connect that factor to all the variables that use that CPD. For example Figure 18d can be factorized as Figure 18e, which can be represented by:

$$f(y_1, y_2, y_3, y_4, y_5) \;=\; f_1(y_1)\, f_2(y_2)\, f_{123}(y_1, y_2, y_3)\, f_{34}(y_3, y_4)\, f_{35}(y_3, y_5) \tag{21.3}$$

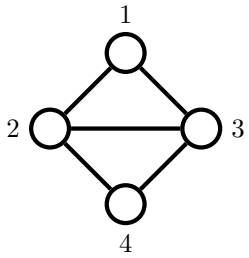
  where we define $f_{123}(y_1, y_2, y_3) \triangleq \mathcal{P}(y_3 \,|\, y_1, y_2)$, etc. If each node has at most one parent (and hence the graph is a chain or a simple tree), then there will be one factor per edge (root nodes can have their prior CPDs absorbed into their children's factors). Such models are equivalent to pairwise MRFs.

- We can perform **BP on a factor graph** (MLPP 22.2.3.2). As mentioned earlier, the intuition is that if the loops are long then the eect of the loops fades out as the messages propagate and the resulting answer is accurate in any case. The method is quite similar to typical BP, where we send message on a factor graph between factors and variables. See Figure 19a for an illustration. The messages from variables to factors are:

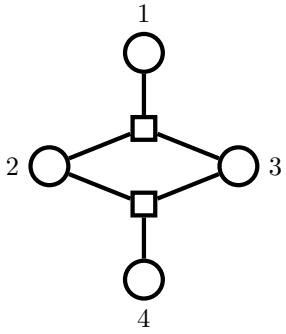
$$m_{y\to f}(y) \;=\; \prod_{f' \in \mathsf{nbr}(y)\,\setminus\,\{f\}} m_{f'\to y}(y) \tag{21.4}$$

  where in Figure 19a the factors neighboring the variable $y$ are, $\mathsf{nbr}(y) \setminus \{f\} \;\triangleq\; \{f_1, \ldots, f_M\}$. The messages from factors to variables are:

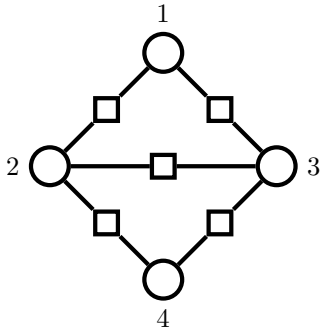
$$m_{f\to y}(y) \;=\; \sum_{y_1} \cdots \sum_{y_N} \left( f(y,\, y_1, \ldots, y_N) \prod_{y' \in \mathsf{nbr}(f)\,\setminus\,\{y\}} m_{y'\to f}(y') \right) \tag{21.5}$$



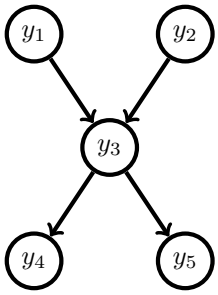

(a) Example MRF



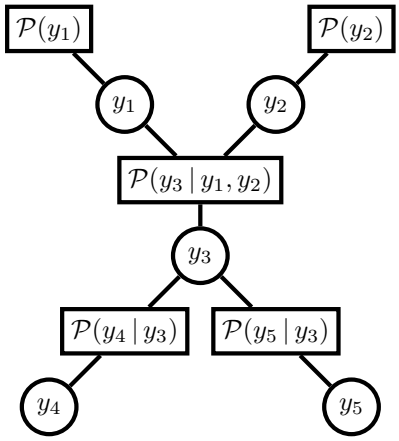

(b) Factor Graph of Figure 18a assuming one potential per maximal clique




(c) Factor Graph of Figure 18a assuming one potential per edge




(d) Example DGM



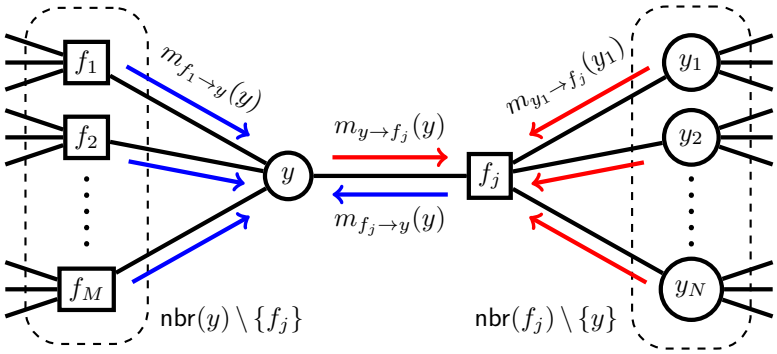

(e) Factor Graph of Figure 18d

Figure 18

Note that $y$ is not included in the set $\{y_1, \ldots, y_N\}$. In the Figure 19a, the variable $y$ is excluded from the set $\{y_1, \ldots, y_N\}$. Also in the figure all variables neighboring factor $f$ are, $\mathsf{nbr}(f) \setminus \{y\} \triangleq \{y_1, \ldots, y_N\}$. At convergence we can compute the final beliefs as a product of incoming messages:

$$\mathsf{bel}(y) \propto \prod_{f \in \mathsf{nbr}(y)} m_{f \to y}(y) \tag{21.6}$$

- LBP does not converge and even when it does, it may converge to wrong answers (MLPP 22.2.4). How can we find when will LBP converge. For this we use an analysis technique based on **computation tree**, which visualizes messages that are passed as the algorithm proceeds (MLPP 22.2.4.1). At the root of the tree you have the node which you are concerned about. At depth 1, you will have nodes which send messages to node at the root at the first iteration. At depth 2, you will have nodes which sent messages to the root node (and their message had travelled through the path specified by the tree). The key insight is that $T$ iterations of LBP is equivalent to exact computation in a computation tree of height $T + 1$. If the strengths of the connections on the edges is sufficiently weak, then the influence of the leaves on the root will diminish over time, and convergence will occur.




(a) Message passing in a bipartite factor graph. Red arrows are messages from variables to factors and blue arrows are messages from factors to variables.

Figure 19

- In practice if LBP is not converging, what should be done? One simple way to reduce the chance of oscillation is use **damping**. This is just a decay term on the message. Indeed, osciallating marginals is sometimes a sign that the LBP approximation is a poor one.
- Another question is how to increase the rate of convergence (MLPP 22.2.4.3). The standard approach in LBP is to perform **synchronous updates**, where all nodes absorb messages in parallel, and then send out messages in parallel. A faster convergence approach is to use **asynchronous updates**, which works in a fixed round-robin fashion, where at iteration $k+1$ the message for edge $i$ is computed using new messages (from iteration $k+1$) from edges edges earlier in the ordering, and using old messages (from iteration $k$) from edges later in the ordering.

  This raises the question what is the right ordering to update the messages. A smarter approach is to pick a set of spanning trees, and then perform an up-down sweep on one tree at a time, keeping all the other messages fixed. This is known as **tree reparameterization (TRP)**. Another technique is to focus more on variables that are most uncertain - this is called **residual belief propagation**. in which messages are scheduled to be sent according to the norm of the difference from their previous value. This scheme usually converges more often, and faster than synchronous, or asynchronous updating with fixed order, or the TRP approach.
- For a graph with a single loop, one can show that the max-product version of LBP will find the correct MAP estimate, if it converges (MLPP 22.2.5)
- **Fast message computation for large state spaces** (MLPP 22.2.6.1) The cost of computing each message in BP (whether in a tree or a loopy graph) is $\mathcal{O}(K^f)$, where $K$ is the number of states, and $f$ is the size of the largest factor ($f = 2$ for pairwise UGMs). For pairwise potential functions of the form $\psi_{st}(y_s, y_t) = \psi_{st}(y_s, -y_t)$, one can compute sum-product messages in $\mathcal{O}(K \log K)$ using FFT. The key insight is that the message computation is just a convolution. If the potential function $\psi(\cdot)$ is a Gaussian-like potential, we can compute the convolution in $\mathcal{O}(K)$ time by sequentially convolving with a small number of box filters (Felzenszwalb and Huttenlocher 2006).
- (MLPP 22.2.6.2) In 2D lattice structures, you can have a coarse-to-fine grid - you can start computing messages on the coarse level and then use them to initialize messages on the next level. This heuristic helps in speeding up BP.
- (MLPP 22.3) UGMs can be represented in the exponential family form:

$$\mathcal{P}(\mathbf{y} \mid \boldsymbol{\theta}, G) = \frac{1}{Z(\boldsymbol{\theta})} \exp\left\{ \sum_{s \in \mathcal{V}} \theta_s(y_s) + \sum_{(s,t) \in \mathcal{E}} \theta_{st}(y_s, y_t) \right\} \tag{21.7}$$

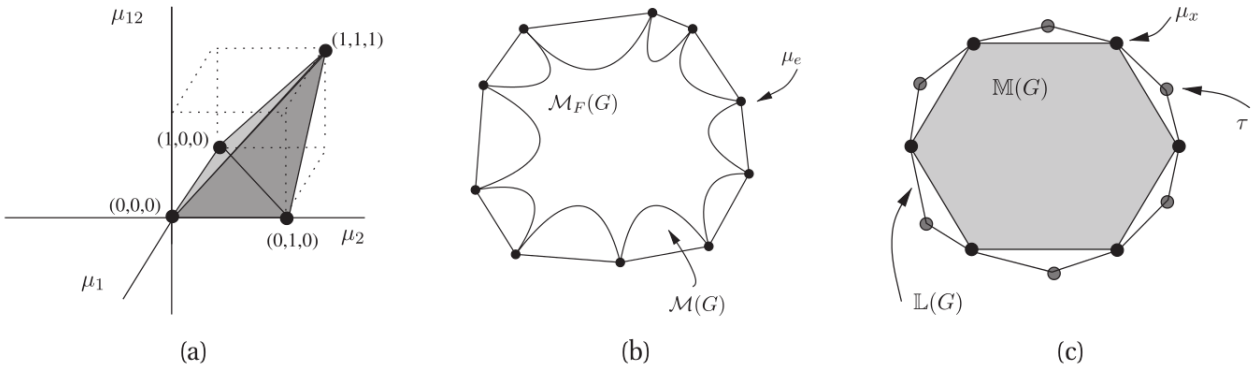



Figure 20: (a) The marginal polytope for an Ising model with two variables. (b) Conceptual illustration of set $\mathbb{M}_F(G)$, which is a nonconvex inner bound on the marginal polytope $\mathbb{M}(G)$. $\mathbb{M}_F(G)$ is used by mean field. (c) Conceptual illustration of the relationship between $\mathbb{M}(G)$ and $\mathbb{L}(G)$, which is used by LBP. The set $\mathbb{L}(G)$ is always an outer bound on $\mathbb{M}(G)$, and the include $\mathbb{M}(G) \subset \mathbb{L}(G)$ is strict whenever $G$ has loops. Taken from (Wainwright and Jordan 2008).

where the graph is $G(\mathcal{V}, \mathcal{E})$. We can rewrite this in the exponential family form (forgetting about the explicit condition on $\boldsymbol{\theta}$ and $G$ because they are known and fixed):

$$\mathcal{P}(\mathbf{y} \,|\, \boldsymbol{\theta}) \;=\; \frac{1}{Z(\boldsymbol{\theta})} \exp\left(-E(\mathbf{y})\right), \qquad\qquad E(\mathbf{y}) \triangleq -\boldsymbol{\theta}^{\mathsf{T}} \boldsymbol{\phi}(\mathbf{y}) \tag{21.8}$$

where $\boldsymbol{\theta} = (\{\theta_{s;j}\}, \{\theta_{s,t;j,k}\})$ are all node and edge parameters (the canonical parameters). On the other hand $\boldsymbol{\phi}(\mathbf{y}) = (\{\mathbb{I}(y_s = j)\}, \{\mathbb{I}(y_s = j,\, y_t = k)\})$ are all the node and edge indicator functions (the sufficient statistics). Note, we are using $s, t \in \mathcal{V}$ to index nodes, and $j, k \in \mathcal{X}$ to index states.

The mean of the sufficient statistics are known as the mean parameters of the model, and are given by:

$$\boldsymbol{\mu} \;=\; \mathbb{E}[\boldsymbol{\phi}(\mathbf{y})] \;=\; (\{\mathcal{P}(y_s = j)\}_s,\, \{\mathcal{P}(y_s = j, y_t = k\}_{s \neq t}) \tag{21.9}$$

$$\;=\; (\{\mu_{s;j}\}_s,\, \{\mu_{st;jk}\}_{s \neq t}) \tag{21.10}$$

This is a vector of length $d = |\mathcal{X}||\mathcal{V}| + |\mathcal{X}|^2|\mathcal{E}|$, containing the node and edge marginals. It completely characterizes the distribution $\mathcal{P}(\mathbf{y}|\boldsymbol{\theta})$, so we sometimes treat $\boldsymbol{\mu}$ as a distribution itself. Equation 21.7 is called the **standard overcomplete representation**. If is called "overcomplete" because it ignores the sum to one constraints. It might be convenient to remove this redundancy, for instance in the potts model, where $\mathcal{X} \in \{0, 1\}$, the minimal parameterization would lead to $d = |\mathcal{V}| + |\mathcal{E}|$.

- The space of allowable $\boldsymbol{\mu}$ vectors is called the **marginal polytope** and is denoted by $\mathbb{M}(G)$, where $G$ is the structure of the graph defining the UGM. This is defined to be the set of all mean parameters for the given model that can be generated from a valid probability distribution:

$$\mathbb{M}(G) \;\triangleq\; \left\{ \boldsymbol{\mu} \in \mathbb{R}^d \,:\, \exists p \quad \text{s.t.} \quad \boldsymbol{\mu} = \sum_{\mathbf{y}} \boldsymbol{\phi}(\mathbf{y}) p(\mathbf{y}) \;\text{ for some }\; p(\mathbf{y}) \geq 0,\, \sum_{\mathbf{y}} \mathcal{P}(\mathbf{y}) = 1 \right\} \tag{21.11}$$

For instance for an Ising model, the constraints on $\boldsymbol{\mu}$ are just a series of half-planes, whose intersection defines a polytope (like an example in Figure 20a). Since $\mathbb{M}(G)$ is obtained by taking the convex combination of the $\boldsymbol{\phi}(\mathbf{y})$ vectors, it can also be written as a **convex hull** of the feature set. The black dots in Figure 20a define that convex hull.

- The exact inference techniques can be viewed as a variational optimization problem (MLPP 22.3.3). This comes down to optimizing:

$$\max_{\boldsymbol{\mu} \in \mathbb{M}(G)} \boldsymbol{\theta}^\mathsf{T} \boldsymbol{\mu} + \mathbb{H}(\boldsymbol{\mu}) = \log Z(\boldsymbol{\theta}) \tag{21.12}$$

  Even though this function seems easy to optimize (since it is the sum of a linear function and the entropy, $\mathbb{H}$ is concave, moreover we are maximizing over a convex set $\mathbb{M}(G)$), this is hard because $\mathbb{M}(G)$ has exponentially many facets. In some cases, there is structure to this polytope that can be exploited by dynamic programming, but in general, exact inference takes exponential time. *Most existing deterministic approximate inference schemes that have been proposed in literature can be seen as different approximations to the marginal polytope.* For instance, mean field inference gives an inner approximation to the marginal polytope (see Figure 20b), where $\mathbb{M}_F(G)$ is a non-convex polytope, which result in multiple local optima (MLPP 22.3.4).

  Similarly when LBP is viewed as a variational inference problem, we get the convex outer approximation $\mathbb{L}(G)$ on $\mathbb{M}(G)$, which is illustrated in Figure 20c (MLPP 22.3.5.1).

- **LBP vs Mean Field (MF)** (MLPP 22.3.6): LBP is exact for trees whereas MF is not. LBP optimizes over node and edge marginals, whereas MF only optimizes over node marginals. Both these facts suggest that LBP should in general be more accurate than MF. MF objective has many more local optima than the LBP objective, so optimizing the MF might be harder. If we initialize MF with BP marginals, we tend to get overconfident yet good results, indicating that the weakness of MF is not the inaccuracy of MF approximations, but rather the severe non-convexity of the MF objective, and the standard coordinate descent optimization used with MF. However, the advantage of MF is that it gives a lower bound on the partition function, unlike BP, which is useful when using it as a subroutine inside a learning algorithm. Also, MF can be extended to handle other distributions beside discrete and Gaussian - since MF works with marginal distributions, which have a single type, rather than having to define pairwise distributions.

- If you consider spanning trees in a graph, we can compute the upper bound on the entropy, by averaging over all trees. This induces the **edge appearance probability** $\rho_{st}$ (MLPP 22.4.2.1) which is the additional term in the edge marginals for variational optimization we can perform for LBP. These edge appearance probabilities live in a space called the **spanning tree polytope**. So long as $\rho_{st} > 0$ for all edges $(s, t)$, the variational optimization is strictly concave with a unique maximum. To get this global optimum, the simplest way is to use **tree reweighted belief propagation (TRW / TRBP)**, which is modification of BP where the message from $t$ to $s$ is now a function of all messages sent from other neighbors $v$ to $t$, as before, but now it is also a function of the message sent from $s$ to $t$. TRBP entropy approximation is an upper bound on the true entropy.

- **Expectation Propagation (EP)** (MLPP 22.5) is a form of belief propagation where the messages are approximated. Here we approximate the posterior at each step using an assumed functional form, such as a Gaussian. The posterior can be computed using moment matching, which locally optimizes $\mathbb{KL}(p\|q)$ for a single term. From this we derive the message to send in the next time step. EP makes multiple passes over the data (making it a batch / offline algorithm), to reduce an a side-effects of the order in which the data was viewed.

- **MAP state estimation** (MLPP 22.6): Now what if we wanted to find the most probable configuration of variables in a discrete-state graphical model, i.e. our goal is to find a MAP assignment of the following form:

$$\mathbf{y}^* = \arg\max_{\mathbf{y} \in \mathcal{X}^m} \mathcal{P}(\mathbf{y} \mid \boldsymbol{\theta}) = \arg\max_{\mathbf{y} \in \mathcal{X}^m} \sum_{i \in \mathcal{V}} \theta_i(y_i) + \sum_{f \in F} \theta_f(\mathbf{y}_f) \tag{21.13}$$

$$= \arg\max_{\mathbf{y} \in \mathcal{X}^m} \boldsymbol{\theta}^\mathsf{T} \boldsymbol{\phi}(\mathbf{y}) \tag{21.14}$$

  where $\theta_i$ are the singleton node potentials, and $\theta_f$ are the factor potentials (which could be pairwise). Note that the partition function $Z(\theta)$ plays no role in MAP estimation. If the treewidth is low, we can solve this problem with the junction tree algorithm, but in general this problem is intractable.

- We cab rewrite the objective in terms of the variational parameters as follows:

$$\arg\max_{\mathbf{y}\in\mathcal{X}^m} \boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}) \;=\; \arg\max_{\boldsymbol{\mu}\in\mathbb{M}(G)} \boldsymbol{\theta}^\mathsf{T}\boldsymbol{\mu} \tag{21.15}$$

  where $\boldsymbol{\phi}(\mathbf{x}) = [\{\mathbb{I}(y_s = j)\}, \{\mathbb{I}(y_f = k)\}]$ and the $\boldsymbol{\mu}$ is a probability vector in the marginal polytope. So instead of optimizing over discrete assignments, we now optimize over probability distributions $\boldsymbol{\mu}$. Even though the equation is linear, and the constraint set $\mathbb{M}(G)$ is convex, the number of facets in $\mathbb{M}(G)$ is exponential in the number of nodes. A standard strategy is to relax the constraints. We now allow $\boldsymbol{\mu}$ to live inside a convex outer bound $\mathbb{L}(G)$:

$$\arg\max_{\mathbf{y}\in\mathcal{X}^m} \boldsymbol{\theta}^\mathsf{T}\boldsymbol{\phi}(\mathbf{x}) \;=\; \arg\max_{\boldsymbol{\mu}\in\mathbb{M}(G)} \boldsymbol{\theta}^\mathsf{T}\boldsymbol{\mu} \;\leq\; \arg\max_{\boldsymbol{\tau}\in\mathbb{L}(G)} \boldsymbol{\theta}^\mathsf{T}\boldsymbol{\tau} \tag{21.16}$$

  If the solution is integral, it is exact; if it is fractional, it is an approximation. This is called (a first order) **linear programming relaxation** (MLPP 22.6.1). How do we solve this optimization? You can use generic linear programming, but that will be slow. We can devise message passing algorithms for solving this optimization problem, which will be faster, as explained next.

- (MLPP 22.6.2) The objective in Equation 21.15 is quite similar to the one in Equation 21.12, $\max_{\boldsymbol{\mu}\in\mathbb{M}(G)} \boldsymbol{\theta}^\mathsf{T}\boldsymbol{\mu} + \mathbb{H}(\boldsymbol{\mu})$, apart from the entropy term. One heuristic way to proceed would be to consider the **zero temperature limit** of the probability distribution $\boldsymbol{\mu}$ - where the entropy becomes zero. We can modify message passing methods to solve this MAP estimation problem. In particular, in the zero temperature limit, the sum operator becomes the max operator, which results in a method called **max-product belief propagation**. We showed before that LBP finds a local optimum of this objective. In the zero temperature limit, this objective is equivalent to the LP relaxation of the MAP algorithm. Unfortunately, max-product LBP does not solve this LP relaxation unless the graph is a tree. However, if we use TRBP / TRW, we have a concave objective, in which case the max-product version of TRBP does solve the above LP relaxation.

  A certain scheduling of this algorithm, known as **sequential TRBP, TRBP-S, or TRW-S**, can be shown to always converge (Kolmogorov 2006), and furthermore, it typically does so faster than the standard parallel updates. The idea is to pick an arbitrary node ordering $y_1, \ldots, y_N$. We then consider a set of trees which is a subsequence of this ordering. At each iteration, we perform max-product BP from $y_1$ towards $y_N$ and back along one of these trees. It can be shown that this monotonically minimizes a lower bound on the energy, and thus is guaranteed to converge to the global optimum of the LP relaxation.

- We now show how to find MAP state estimates, or equivalently, minimum energy configurations, by using the **max flow/min cut** algorithms fir graphs. The best known running times are $\mathcal{O}(|\mathcal{E}||\mathcal{V}|\log|\mathcal{V}|)$ or $\mathcal{O}(|\mathcal{V}|^3)$. The class of these methods is known as **graphcuts** (MLPP 22.6.3). In the restricted case of MRFs with binary nodes and some particular potentials, graphcuts can find the exact global optimum. For the case of multiple states per node, which are assumed to have some underlying ordering we can approximately solve this case by solving a series of binary sub-problems.

- **Graphcuts for the generalized Ising model** (MLPP 22.6.3.1): Let us start by considering a binary MRF where the edge energies have the following form:

$$E_{uv}(y_u, y_v) = \begin{cases} 0 & \text{if } y_u = y_v \\ \lambda_{uv} & \text{if } y_u \neq y_v \end{cases} \tag{21.17}$$

  where $\lambda_{uv} \geq 0$ is the edge cost. This encourages neighboring nodes to have the same value (since we are trying to minimize energy). Since we are free to add any constant we like to the overall energy without affecting the MAP state estimate, let us rescale the local energy terms such that either $E_u(1) = 0$ or $E_u(0) = 0$. Now let us construct a graph which has the same set of nodes as the MRF, plus two distinguished nodes: the source $s$ and the sink $t$. If $E_u(1) = 0$, we add the edge $y_u \to t$ with cost $E_u(0)$. This ensures that if $u$ is not in

partition $\mathcal{X}_t$, meaning $u$ is assigned to state 0, we will pay a cost of $E_u(0)$ in the cut. Similarly, if $E_u(0) = 0$, we add the edge $s \to y_u$ with cost $E_u(1)$. Finally, for every pair of variables that are connected in the MRF, we add edges $y_u \to y_v$ and $y_v \to y_u$, both with the cost $\lambda_{uv} \geq 0$.

Having constructed the graph, we compute a minimal $s-t$ cut. This is a partition of the nodes into two sets, $\mathcal{X}_s$, which are nodes connected to $s$, and $\mathcal{X}_t$, which are nodes connected to $t$. We pick the partitioning / cut which minimizes the sum of the cost of the edges between nodes on different sides of the partition. Minimizing the cost in this graph is equivalent to minimizing the energy in the MRF, which, in turn, maximizing the probability. Hence, nodes that are assigned to $s$ have an optimal state of 0, and the nodes that are assigned to $t$ have an optimal state of 1. One key point to note here is that because nodes which are eventually associated with $t$ are labeled 1, it was essential to set the $E_u(0)$ on the $y_u \to t$ edge, and vice versa. This is like saying because node $u$ doesn't like being associated with label 0, $u$ would put a cost/energy on being broken away from label 1.

- Graphcut construction can be extended to binary MRFs with more general kinds of potential functions. In particular, suppose each pairwise energy satisfies the following condition:

$$E_{uv}(1,1) + E_{uv}(0,0) \leq E_{uv}(1,0) + E_{uv}(0,1) \tag{21.18}$$

which just says that the sum of the diagonal energies is less than the sum of the off-diagonal energies. In this case, we say the energies $E_{uv}$ are **submodular** (Kolmogorov and Zabih 2004) (MLPP 22.6.3.2)[2]. An example of a submodular energy is in the Ising model as given in Equation 21.17, where $\lambda_{uv} > 0$. This is also known as an **attractive MRF** or **associative MRF**, since the model "wants" neighboring states to be the same.

To apply graphcuts to a binary MRF with submodular potentials, we construct the pairwise edge weights as follows:

$$E'_{uv}(0,1) \triangleq \lambda_{uv} = E_{uv}(1,0) + E_{uv}(0,1) - E_{uv}(0,0) - E_{uv}(1,1) \tag{21.19}$$

This is guaranteed to be non-negative by virtue of the submodularity assumption. In addition we construct new local edge weights as follows: first we initialize $E'(u) = E(u)$, and then for each edge pair $(u, v)$, we update these values as follows:

$$E'_u(1) := E'_u(1) + (E_{uv}(1,0) - E_{uv}(0,0)) \tag{21.20}$$
$$E'_v(1) := E'_v(1) + (E_{uv}(1,1) - E_{uv}(1,0)) \tag{21.21}$$

We now construct a graph in a similar way to before. Specifically, if $E'_u(1) > E'_u(0)$, we add the edge $s \to u$ with cost $E'_u(1) - E'_u(0)$, otherwise we add the edge $u \to t$ with cost $E'_u(0) - E'_u(1)$. Finally for every MRF edge for which $E'_{uv}(0,1) > 0$, we add a graphcuts edge $y_u - y_v$ with cost $E'_{uv}(0,1)$. (We don't need to add the edge in both directions). One can show that the min cut in the graph is the same as the minimum energy configuration. Thus we can use max flow/min cut to find the globally optimal MAP estimate.

- **Graphcuts for non-binary metric MRFs** (MLPP 22.6.3.3): here we can use graph-cuts to achieve approximate MAP estimation, where each node can have multiple states. However, we require that the pairwise energies form a metric. We call such a model a **metric MRF**. For example, suppose the states have a natrual ordering, as commonly arises if they are a discretization of some continuous natural space.

  One version of graphcuts is the **alpha expansion**. At each step, it picks one of the available labels or states and calls it $\alpha$, and then solves a binary subproblem where each variable can choose to remain in its current

[2]Submodularity is the discrete analog of convexity. Intuitively, it corresponds to the law of diminishing returns, i.e. the extra value of adding one more element to a set is reduced if the set is already large. More formally, we say that $f : 2^S \to \mathbb{R}$ is submodular if for any $A \subset B \subset S$ and $x \in S$, we have $f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B)$. In Equation 21.18, where $E_{uv}(0,1) \triangleq f(\{b\})$ and $E_{uv}(1,1) \triangleq f(\{a,b\})$, and so on - the equation this translates to $f(\{a,b\}) + f(\emptyset) \leq f(\{a\}) + f(\{b\})$, which is equivalent to $f(\{b\}) - f(\emptyset) \geq f(\{a,b\}) - f(\{a\})$, hence fulfilling submodularity condition, since $\emptyset \subset \{a\}$. If $-f$ is submodular, then $f$ is **supermodular**.

state or become $\alpha$. Alpha expansion can be applied to any metric MRF. At each step of alpha expansion, we find the optimal move from amongst an exponentially large set; thus we reach a **strong local optimum**, of much lower energy than the local optima found by standard greedy label flipping methods such as **iterative conditional models (ICM)**. Once the algorithm can converges, for instance for Potts model, the energy of the resulting solution is at most 2 times the optimal energy.

Another version of graphcuts is called **alpha-beta swap**. At each step, two labels are chosen, call them $\alpha$ and $\beta$. All the nodes labeled $\alpha$ can change to $\beta$ (and vice versa) if it reduces the energy of the solution. The resulting binary subproblem can be solved exactly, even if the energies are only semi-metric (i.e. the triangle inequality need not hold). Even though we can apply it to wider class of energies than $\alpha$-expansion, empirically the latter performs better usually.

- (MLPP 22.6.4) When comparing different methods for MAP estimation on various low-level vision problems, graphcuts and TRW give the best results, with regular max-product BP being much worse. In terms of speed, graphcuts is fastest. with TRW being a close second. Other algorithms, such as ICM, simulated annealing do even worse. (Note sometimes we are not even optimizing on the right energy - as it has been shown that the global minima is lower than the GT estimate - but making the model right and, in the process, more complex - like enforcing long range constraints - makes BP too slow, and/or make the potentials non-submodular - making graphcuts inapplicable.

- **Dual Decompostion** (MLPP 22.6.5): If we are interested in computing Equation 21.13, where $F$ represents a set of factors, we can come with a scheme where we optimize each term independently, but then to introduce constraints that force all the local estimates of the variables' values to agree with each other. The basic idea is to duplicate all variables once for each factor, and then force them to be equal. On top of this we introduce Lagrange multipliers or dual variables to enforce these constraints.

# 22 Monte Carlo Inference

- (MLPP 23.1) Up until now we have discussed deterministic algorithms for posterior inference. The problem with these methods is that they can be rather complicated to derive and their domain of applicability is somewhat limited (e.g. they usually assume conjugate priors and exponential family likelihood). Here we discuss the idea of Monte Carlo approximation. The basic idea is simple: generate some (unweighted) samples from the posterior $\mathbf{x}^{(s)} \sim \mathcal{P}(\mathbf{x}|\mathcal{D})$, and then use these to compute any quantity of interest, such as the posterior marginal $\mathcal{P}(x_1|\mathcal{D})$, or the posterior of the difference of two quantities $\mathcal{P}(x_1 - x_2|\mathcal{D})$, or the posterior predictive $\mathcal{P}(y|\mathcal{D})$, etc. All of these quantities can be approximated by $\mathbb{E}[f|\mathcal{D}] \approx \frac{1}{S}\sum_{s=1}^{S} f(\mathbf{x}^{(s)})$. By generating enough examples we cab achieve any desired level of accuracy. The man question is: how do we efficiently generate sample from a probability distribution - given that it could be even in high dimensions?

- The simplest way to sample from a univariate distribution is based on the **inverse probability transform** (MLPP 23.2.1). Let $F$ be the CDF of some distribution we want to sample, and $F^{-1}$ be its inverse. Then we have the following theorem:

  **Theorem 22.1.** Inverse Probability Transform Theorem*: If $U \sim U(0,1)$ is a uniform random variable, then $F^{-1}(U) \sim F$.*

  Hence we can sample from a univariate distribution as follows: (1) generate a random number $u \sim U(0,1)$; (2) compute $x := F^{-1}(u)$; (3) the generated sample is $x$.

- If the probability density function of a random variable $X$ is given as $f_X(x)$ it is possible to calculate the probability density function of some variable $Y \sim g_Y(y)$. This is called a **change of variable** and is in practice used to generate a random variable of arbitrary shape $f_{g(x)} = f_Y$ using a known (for instance uniform) random

number generator. This follows from the fact that the probability contained in a differential area must be invariant under change of variables, i.e.:

$$|f_Y(y)\,\partial y| \;=\; |f_X(x)\,\partial x|, \qquad f_Y(y) \;=\; f_X(x)\left|\frac{\partial x}{\partial y}\right| \tag{22.1}$$

This is useful in sampling, as we will see below.

- **Box-Muller method** (MLPP 23.2.2) is a method to sample from a Gaussian. The idea is that we sample uniformly from a unit radius circle, and then use change of variables formula to derive samples from a spherical 2d Gaussian. This can be thought of as two samples from a 1d Gaussian. In more detail, sample $z_1, z_2 \in (-1, 1)$ uniformly, and then discard pairs that don't satisfy $z_1^2 + z_2^2 \leq 1$. The result will be points uniformly distributed inside the unit circle, so $\mathcal{P}(\mathbf{z}) = \frac{1}{\pi}\mathbb{I}(z \text{ inside circle})$. Now define $x_i = z_i\sqrt{\frac{-2\log r^2}{r^2}}$, $i = 1:2$, where $r^2 = z_1^2 + z_2^2$. Using the multivariate change of variables formula we have:

$$\mathcal{P}(x_1, x_2) \;=\; \mathcal{P}(z_1, z_2)\left|\frac{\partial(z_1, z_2)}{\partial(x_1, x_2)}\right| \;=\; \left[\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{1}{2}x_1^2\right)\right]\left[\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{1}{2}x_2^2\right)\right] \tag{22.2}$$

  Hence $x_1$ and $x_2$ are two independent samples from a univariate Gaussian.

  To sample from a multivariate Gaussian, we first compute the Cholesky decomposition of its convariance matrix, $\mathbf{\Sigma} = \mathbf{L}\mathbf{L}^\mathsf{T}$, where $\mathbf{L}$ is lower triangular. Next we sample $\mathbf{x} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ using the Box-Muller method. Finally we set $\mathbf{y} = \mathbf{L}\mathbf{x} + \boldsymbol{\mu}$.

- 

- rejectsample An alternative to the inverse CDF method is **rejection sampling** (MLPP 23.3). In rejection sampling, we create a **proposal distribution** $q(x)$ which satisfies $Mq(x) \geq \tilde{p}(x)$, for some constant $M$, where $\tilde{p}(x)$ is an unnormalized version of $p(x)$ (i.e. $p(x) = \tilde{p}(x)/Z_p$ for some possibly unknown constant $Z_p$). The function $Mq(x)$ provides an upper envelope for $\tilde{p}$. We then sample $x \sim q(x)$, and then we sample $u \sim U(0, 1)$, which corresponds to pick a random height under the envelope. If $u > \frac{\tilde{p}(x)}{Mq(x)}$, we reject the sample, otherwise accept it. You can verify this by observing if the CDF of this sampling distribution is equivalent to $p(x)$. We would want to choose $M$ as small as possible while still satisfying $Mq(x) \geq \tilde{p}(x)$, because that will increase the probability of a sample being accepted.

- (MLPP 23.3.3) We can apply rejection sampling to Bayesian statistics as follows: suppose we want to draw (unweighted) samples from the posterior $\mathcal{P}(\boldsymbol{\theta}|\mathcal{D}) = \mathcal{P}(\mathcal{D}|\boldsymbol{\theta})\mathcal{P}(\boldsymbol{\theta})/\mathcal{P}(\mathcal{D})$. We can use rejection sampling with $\tilde{p}(\boldsymbol{\theta}) = \mathcal{P}(\mathcal{D}|\boldsymbol{\theta})\mathcal{P}(\boldsymbol{\theta})$ as the target distribution and $q(\boldsymbol{\theta}) = \mathcal{P}(\boldsymbol{\theta})$ as our proposal, and $M = \mathcal{P}(\mathcal{D}|\hat{\boldsymbol{\theta}})$, where $\hat{\boldsymbol{\theta}} = \arg\max \mathcal{P}(\mathcal{D}|\boldsymbol{\theta})$ is the MLE. We accept points with probability:

$$u \;>\; \frac{\tilde{p}(\boldsymbol{\theta})}{Mq(\boldsymbol{\theta})} \;=\; \frac{\mathcal{P}(\mathcal{D}|\boldsymbol{\theta})}{\mathcal{P}(\mathcal{D}|\hat{\boldsymbol{\theta}})} \tag{22.3}$$

  Thus samples from the prior that have high likelihood are more likely to be retained in the posterior. Of course, if there is a big mismatch between the prior and posterior (which will be the case if the prior is vague and the likelihood is informative), this procedure is very inefficient.

- How to get a tighter envelope for $q(x)$. If we have a log concave density $p(x)$, we can upper bound this log density by piecewise linear function. We choose the initial locations for the pieces based on a fixed grid over the support of the distribution. We then compute the gradient of the log density at these locations and make the likes be tangent to these points. Sampling from this envelope is simple. If the sample $x$ is rejected, we create new grid point at $x$, and thereby refining the envelope. The tightness of the envelope improves with with the number of grid points, and the rejection rate goes down. This is known as **adaptive rejection sampling** (MLPP 23.3.4).

- Rejection sampling does extremely poorly in higher dimensions (MLPP 23.3.5).
- Another Monte Carlo method is known as **importance sampling** (MLPP 23.4) for approximating integrals of the form:

$$I \;=\; \mathbb{E}[f] \;=\; \int f(\mathbf{x})\mathcal{P}(\mathbf{x})d\mathbf{x} \tag{22.4}$$

Importance sampling is actually not sampling (because you are not sampling the distribution $\mathcal{P}(\mathbf{x})$, but its a variant of Monte Carlo approximation - where we try to approximate $\mathbb{E}[f] \approx \frac{1}{n}\sum_{i=1}^{n} f(\mathbf{x}^{(i)})$, and here $\mathbf{x}^{(i)} \sim p$. This is a big assumption that you can sample from $p$ as before. Now we need to correct for this assumption that we can sample from $p$. The idea is to draw samples $\mathbf{x}^{(s)}$ in regions which have high probability, $\mathcal{P}(\mathbf{x}^{(s)})$, but also where $|f(\mathbf{x}^{(s)})|$ is large. The result can be efficient, meaning it needs less samples than if were to sample directly from distribution $p(\mathbf{x})$. The reason is that the samples are focused on the important parts of space. For instance, suppose we want to estimate the probability of a rate event $E$. Define $f(\mathbf{x}^{(s)}) = \mathbb{I}(\mathbf{x}^{(s)} \in E)$. Then it is better to sample from a proposal distribution of the form $q(\mathbf{x}) \propto f(\mathbf{x})p(\mathbf{x})$ than to sample from $p(\mathbf{x})$ itself.

Importance sampling samples from any proposal $q(\mathbf{x})$. These samples are used to estimate the integral:

$$\mathbb{E}[f] \;=\; \int f(\mathbf{x})\mathcal{P}(\mathbf{x})d\mathbf{x} \;=\; \int f(\mathbf{x})\frac{\mathcal{P}(\mathbf{x})}{q(\mathbf{x})}q(\mathbf{x})d\mathbf{x} \;\approx\; \frac{1}{S}\sum_{s=1}^{S} w_s f(\mathbf{x}^{(s)}) \;=\; \hat{I} \tag{22.5}$$

where now we sample $\mathbf{x}^{(s)} \sim q$, and $w_s \triangleq \frac{\mathcal{P}(\mathbf{x}^{(s)})}{q(\mathbf{x}^{(s)})}$ are the **importance weights**. The second equality holds for any $q$, as long as $q(\mathbf{x}) = 0 \;\Rightarrow\; \mathcal{P}(\mathbf{x}) = 0$. Note that unlike rejection sampling we use all the samples. Now the question is how should you choose the proposal $q(\mathbf{x})$? A natural criterion is to minimize the variance of the estimate $\hat{I} = \sum_s \frac{\mathcal{P}(\mathbf{x}^{(s)})f(\mathbf{x}^{(s)})}{q(\mathbf{x}^{(s)})}$. Minimizing this, leads to a lower bound which holds for any $q$:

$$q^*(\mathbf{x}) \;=\; \frac{|f(\mathbf{x})|\mathcal{P}(\mathbf{x})}{\int |f(\mathbf{x}')|\mathcal{P}(\mathbf{x}')d\mathbf{x}'} \tag{22.6}$$

If the distributions are unnormalized $\frac{Z_q}{Z_p}\frac{\tilde{p}(\mathbf{x})}{\tilde{q}(\mathbf{x})} = \frac{p(\mathbf{x})}{q(\mathbf{x})}$. Now the weights would be unnormalized too, $\hat{w}^{(s)} = \frac{\tilde{p}(\mathbf{x}^{(s)})}{\tilde{q}(\mathbf{x}^{(s)})}$. We can estimate the ratio $\frac{Z_q}{Z_p}$ as the mean of the weights $\frac{1}{S}\sum_{s=1}^{S}\tilde{w}_s$, moreover $w_s \triangleq \frac{\tilde{w}_s}{\sum_{s'}\tilde{w}_{s'}}$.

- If we have no evidence, how do we use importance sampling to generate samples from a DGM (MLPP 23.4.3)? We sample from the root nodes, then sample their children (given what we sampled from the root nodes), then sample their children and so on. This is called **ancestral sampling**. Now suppose we have some evidence, i.e. some nodes are "clamped" to observed values, and we want to sample from the posterior $\mathcal{P}(\mathbf{x}|data)$. If all the variables are discrete, we can perform ancestral sampling, but as soon as we sample a value that is inconsistent with an observed value, we reject the whole sample and start over. This is called **logic sampling**.

  Needless to say, Logic sampling is inefficient, plus the variables are constrained to be discrete. Rather than sampling over the observed variables, we just accept their observed value, and adjust the weights accordingly. This is called **likelihood weighting**. Now the weights would be:

$$w_s \;=\; \prod_{t \in E} \mathcal{P}(x_t^{(s)}|\mathbf{x}_{\mathsf{pa}(t)}^{(s)}) \tag{22.7}$$

  where $E$ is the set of observed nodes.

- We can draw unweighted samples from $\mathcal{P}(\mathbf{x})$ by first using importance sampling (with proposal $q$) to generate a distribution of the form:

$$\mathcal{P}(\mathbf{x}) \;\approx\; \sum_{s=1}^{S} w_s \delta_{\mathbf{x}^{(s)}}(\mathbf{x}) \tag{22.8}$$

where $w_s$ are the normalized importance weights. We then sample with replacement from this equation, where the probability that we pick $\mathbf{x}^{(s)}$ is $w_s$. This is known as **Sampling Importance Resampling (SIR)**. The result is an un-weighted approximation of the form:

$$\mathcal{P}(\mathbf{x}) \approx \frac{1}{S'} \sum_{s=1}^{S'} \delta \left[ \mathbf{x} - \mathbf{x}^{(s)} \right] \tag{22.9}$$

Note that typically $S' \ll S$.

- The EKF (Section 17) and UKF (Section 17) can partially cope with nonlinear temporal and measurement models. However, they both represent the uncertainty over the state as a normal distribution. They are hence ill-equipped to deal with situations where the true probability distribution over the state is $\mathcal{P}(\mathbf{z}_t|\mathbf{x}_t)$ is multimodal. In this situation neither EKF nor the UKF suffice: the EKF models only one of the resulting clusters and the UKF tries to model both with a single normal model, assigning a large probability to the empty region between clusters. **Particle filtering** (CVPrince 19.5) (MLPP 23.5) resolves this problem by representing the probability density as a set of particles in the state space. Each particle can be thought of as representing a hypothesis about the possible state. When the state is tightly contrained by the data, all of these particles will lie close to each other. In more ambiguous cases, they will be widely distributed or clustered into groups of competing hypotheses. The particles can be evolved through time or projected down to simulate measurements regardless of how nonlinear the functions are. This leads to particle filters being able to model multimodal distributions.

  The probability distribution $\mathcal{P}(\mathbf{z}_{t-1} \,|\, \mathbf{x}_{1:t-1})$ is represented by a weighted sum of $J$ weighted particles:

$$\mathcal{P}(\mathbf{z}_t \,|\, \mathbf{x}_{1:t}) \;=\; \sum_{j=1}^{J} \hat{w}_t^{(j)} \delta \left[ \mathbf{z}_t - \hat{\mathbf{z}}_t^{(j)} \right] \tag{22.10}$$

  where $\hat{\mathbf{z}}_t^{(j)}$ represents the $j$'th particle. Here the normalized weights $\hat{\mathbf{w}}_t$ which are positive and sum to one. Each particle represents a hypothesis about the state and the weight of the particle indicates our confidence in that hypothesis. Our goal is to compute the probability distribution $\mathcal{P}(\mathbf{z}_{t+1} \,|\, \mathbf{x}_{1:t+1})$ at the next time step, which will be represented in a similar fashion.

  The simplest way to do particle filtering is through **sequential importance sampling**. Let's say the proposal distribution has the form $q(\mathbf{z}_{1:t}|\mathbf{x}_{1:t})$. For each old sample $s$ we propose a new sample from the proposal distribution $\mathbf{z}_t^{(s)} \sim q(\mathbf{z}_t|\mathbf{z}_{t-1}^{(s)}, \mathbf{x}_t)$. This new particle is given the weight $w_t^{(s)}$ using the formula:

$$w_t^{(s)} \;\propto\; w_{t-1}^{s} \frac{\mathcal{P}(\mathbf{x}_t|\mathbf{z}_t^{(s)})\, \mathcal{P}(\mathbf{z}_t^{(s)}|\mathbf{z}_{t-1}^{(s)})}{q(\mathbf{z}_t^{(s)}|\mathbf{z}_{t-1}^{(s)}, \mathbf{x}_t)} \tag{22.11}$$

  These weights are normalized to $\hat{w}_t^{(s)} = w_t^{(s)} / \sum_{s'} w_t^{(s')}$.

- One problem with sequential importance sampling is that the algorithm fails after a few steps because most of the particles will have negligible weight. This is called the **degeneracy problem**, and occurs because we are sampling in a high dimensional space. The two solutions to the degeneracy problem is to add a resampling step, and use a good proposal distribution.

- Another solution to the degeneracy problem is to set the proposal distribution as sampling from the prior:

$$q(\mathbf{z}_t|\mathbf{z}_{t-1}^{(s)}, \mathbf{x}_t) \;=\; \mathcal{P}(\mathbf{z}_t|\mathbf{z}_{t-1}^{(s)}) \tag{22.12}$$

  Now the weight update simplifies to $w_t^{(s)} \propto w_{t-1}^{(s)} \mathcal{P}(\mathbf{x}_t|\mathbf{z}_t^{(s)})$. This can be thought of as a "generate and test" approach: we sample values from the dynamic model, and then evaluate how good they are after we

see the data. This is applied to a particle filtering method called the **conditional density propagation** or **condensation algorithm**. As usual this process is divided into a time evolution and a measurement incorporation step.

**Time evolution**: In the time evolution step, we create $J$ predictions $\hat{\mathbf{z}}_+^{(j)}$ for the time evolved state. Each is represented by an unweighted particle:

1. Sample an index $n \in \{1, \dots, J\}$ of the original weighted particles according to the weights. In other words we draw a sample from $\text{Cat}(n \mid \mathbf{w})$, and
2. Draw the sample $\hat{\mathbf{z}}_+^{(j)}$ from the temporal update distribution $\mathcal{P}(\mathbf{z}_t \mid \mathbf{z}_{t-1} = \hat{\mathbf{z}}_{t-1}^{(n)})$.

In this process, the final unweighted particles $\hat{\mathbf{z}}_+^{(j)}$ are created from the original weighted particles $\hat{\mathbf{z}}_{t-1}^{(j)}$ according to the weights $\mathbf{w} = [w_1, \dots, w_J]$. Hence the highest weighted original particles may contribute repeatedly to the final set, and the lowest weighted ones may not contribute at all.

**Measurement incorporation**: Here we weight the new set of particles according to how well they agree with the observed data:

1. Pass the particles through the measurement model $\hat{\mathbf{x}}_+^{(j)} = h[\hat{\mathbf{z}}_+^{(j)}]$.
2. weight the particles according to their agreement with the observation density. For example, with a Gaussian measurement model, we could use: $w_j \propto \mathcal{P}(\mathbf{x}_t \mid \hat{\mathbf{z}}_+^{(j)}) = \mathcal{N}(\mathbf{x}_t \mid \hat{\mathbf{x}}_+^{(j)}, \mathbf{R})$
3. Ensure $\{w_j\}_{j=1}^J$ sums to one.
4. Finally we set the new states $\hat{\mathbf{z}}_t^{(j)}$ to the predicted states $\hat{\mathbf{z}}_+^{(j)}$ and the new weights to $w_j$.

The main disadvantage of particle filters is their cost: in high dimensions, a very large number of particles may be required to get an accurate representation of the true distribution over the state.

- **Rao-Blackwellised particle filtering (RBPF)** (MLPP 23.6).

# 23 Markov chain Monte Carlo (MCMC) inference

- The Monte Carlo methods we discussed before don't fare well in higher dimensional spaces. The most popular method for sampling from high-dimensional distributions is **Markov chain Monte Carlo (MCMC)** (MLPP 24.1).
- The basic idea behind MCMC is to construct a Markov chain on the state space $\mathcal{X}$ whose stationary distribution is the target density $\mathcal{P}^*(\mathbf{x})$ of interest (this may be a prior or a posterior). That is, we perform random walk on the state space, in such a way that the fraction of time we spend in each state $\mathbf{x}$ is proportional to $\mathcal{P}^*(\mathbf{x})$. By drawing (correlated) samples $\mathbf{x}_0$, $\mathbf{x}_1$, $\mathbf{x}_2, \dots$, from the chain, we can perform Monte Carlo integration wrt $\mathcal{P}^*$.
- It is worth briefly comparing MCMC to variational inference. The advantages of variational inference are (1) for small to medium problems, it is usually faster; (2) it is deterministic; (3) is it easy to determine when to stop; (4) it often provides a lower bound on the log likelihood. The advantages of sampling are: (1) it is often easier to implement; (2) it is applicable to a broader range of models, such as models whose size or structure changes depending on the values of certain variables (e.g., as happens in matching problems), or models without nice conjugate priors; (3) sampling can be faster than variational methods when applied to really huge models or datasets.
- A popular MCMC method is known as **Gibbs sampling** (MLPP 24.2). This is the MCMC analog of coordinate descent. The idea behind Gibbs sampling is that we sample each variable in turn, conditioned on the values of all the other variables in the distribution. That is, given a joint sample $\mathbf{x}^{(s)}$ of all the variables,

we generate a new sample $\mathbf{x}^{(s+1)}$ by sampling each component in turn, based on the most recent values of the other variables. For example, if we have $D = 3$ variables, we could sample in the following order:

1. $x_1^{(s+1)} \sim \mathcal{P}(x_1 \mid x_2^{(s)}, x_3^{(s)})$
2. $x_2^{(s+1)} \sim \mathcal{P}(x_2 \mid x_1^{(s+1)}, x_3^{(s)})$
3. $x_3^{(s+1)} \sim \mathcal{P}(x_3 \mid x_2^{(s+1)}, x_3^{(s+1)})$

This readily generalizes to $D$ variables. If $x_i$ is a visible variable, we do not sample it, since the value is already known. The expression $\mathcal{P}(x_i|\mathbf{x}_{-i})$ is called the **full conditional** for variable $i$. In general, $x_i$ may only depend on some of the other variables. If we represent $\mathcal{P}(\mathbf{x})$ as a graphical model, we can infer the dependencies by looking at $i$'s Markov blanket, which are its neighbors in the graph. Thus to sample $x_i$, we only need to know the values of $i$'s neighbors. In this sense, Even though Gibbs sampling seems like a distributed algorithm, it is not a parallel algorithm, since the samples must be generated sequentially.

- Gibbs sampling can be quite slow, since it only updates one variable at a time (so-called **single site updating**), if the variables are highly correlated, it will take a long time to move away from the current state. If the variables are highly correlated, the algorithm will move very slowly through the state space. In particular, the size of the moves is controlled by the variance of the conditional distributions. In some cases we can efficiently sample groups of variables at a time. This is called **blocking Gibbs sampling** or **blocked Gibbs sampling** (MLPP 24.2.8), and can make much bigger moves through the state space.

- When the distribution's corresponding graphical model has no useful Markov structure, Gibbs sampling cannot do much - since we sample the full conditional $\mathcal{P}(x_i|\mathbf{x}_{-i})$. In addition Gibbs sampling can be quite slow. A more general MCMC algorithm is **Metropolis Hastings (MH)** (MLPP 24.3).

  The basic idea in MH is that at each step, we propose to move from the current state $\mathbf{x}$ to a new state $\mathbf{x}'$ with probability $q(\mathbf{x}'|\mathbf{x})$, where $q$ is the proposal distribution (sometimes also called the **kernel**). Note, the meaning of this proposal distribution $q$, is somewhat different than the one used for rejection or importance sampling. You are free to choose any kind of proposal distribution you want, subject to some conditions. This flexibility makes MH quite powerful. A commonly used proposal is a symmetric Gaussian distribution centered on the current state $q(\mathbf{x}' \mid \mathbf{x}, \mathbf{\Sigma})$; this is called a **Random walk Metropolis algorithm**. If we use a proposal of the form $q(\mathbf{x}'|\mathbf{x}) = q(\mathbf{x}')$, where the new state is independent of the old state, we get a method known as the **independence sampler**, which is similar to importance sampling (see Section 22).

  Having proposed a move to $\mathbf{x}'$, we then decide whether to accept this proposal or not according to a condition which ensure that the fraction of time spent in each state is proportional to $\mathcal{P}^*(\mathbf{x})$. If the proposal is accepted, the new state is $\mathbf{x}'$, otherwise we remain at the old/current state $\mathbf{x}$, which is equivalent to repeating the sample. If the proposal is symmetric, so $q(\mathbf{x}'|\mathbf{x}) = q(\mathbf{x}|\mathbf{x}')$, the acceptance probability is given by the following formula:

$$r = \min\left(1, \frac{\mathcal{P}^*(\mathbf{x}')}{\mathcal{P}^*(\mathbf{x})}\right) \tag{23.1}$$

  where if $\mathbf{x}'$ is more probable than $\mathbf{x}$, we definitely move to the new state, but even if it is not, there is some chance given by the ratio $\frac{\mathcal{P}^*(\mathbf{x}')}{\mathcal{P}^*(\mathbf{x})}$. So instead of greedily moving to only more probable states, we occasionally allow "downhill" moves to less probable states. You can prove that this scheme ensures that the fraction of time spent in each state $\mathbf{x}$ is proportional to $\mathcal{P}^*(\mathbf{x})$. If the proposal is assymetric, i.e. $q(\mathbf{x}'|\mathbf{x}) \neq q(\mathbf{x}|\mathbf{x}')$, we need the **Hastings correction**, which is given by:

$$r = \min\left(1, \frac{\mathcal{P}^*(\mathbf{x}')/q(\mathbf{x}'|\mathbf{x})}{\mathcal{P}^*(\mathbf{x})/q(\mathbf{x}|\mathbf{x}')}\right) \tag{23.2}$$

This correction is needed to compensate for the fact that the proposal distribution itself might favor certain states. Note that here $\mathcal{P}^*$ could be an unnormalized distribution.

**Algorithm 7:** Metropolis Hastings (MH) Algorithm

**Input**: The initial state $\mathbf{x}^{(0)}$

**for** $s := \{0, 1, 2, \dots\}$ **do**

- Sample $\mathbf{x}' \sim q(\mathbf{x}' \mid \mathbf{x}^{(s)})$
- Compute acceptance probability: $\alpha = \frac{\mathcal{P}^*(\mathbf{x}')/q(\mathbf{x}'|\mathbf{x}^{(s)})}{\mathcal{P}^*(\mathbf{x}^{(s)})/q(\mathbf{x}^{(s)}|\mathbf{x}')}$
- Compute: $r = \min(1, \alpha)$
- Sample: $u \sim U(0, 1)$
- Set new sample to:

$$\mathbf{x}^{(s+1)} = \begin{cases} \mathbf{x}' & \text{if } u < r \\ \mathbf{x}^{(s)} & \text{if } u \geq r \end{cases}$$

**Output**: Return the set of samples $\{\mathbf{x}^{(0)}, \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots\}$

- It can be shown that Gibbs sampling is a special case of MH (MLPP 24.3.2). It turns out that this would be the case when the proposal distribution is:

$$q(\mathbf{x}'|\mathbf{x}) = \mathcal{P}(x_i'|\mathbf{x}_{-i})\, \mathbb{I}(\mathbf{x}'_{-i} = \mathbf{x}_{-i}) \tag{23.3}$$

That is, we move to a new state where $x_i$ is sampled from its full conditional, but $\mathbf{x}_{-i}$ is left unchanged. You can also prove that the acceptance rate of each proposal is 1. Although that doesn't mean that Gibbs will converge rapidly since it updates one coordinate at a time.

- (MLPP 24.3.3) For a given target distribution $\mathcal{P}^*$, a proposal distribution $q$ is valid or admissible if it gives a non-zero probability of moving to the states that have non-zero probability in the target. Formally we can write this:

$$\mathsf{supp}(\mathcal{P}^*) \subseteq \mathsf{supp}(q(\cdot|x)) \tag{23.4}$$

For example, a Gaussian random walk proposal has non-zero probability density on the entire state space. and hence is a valid proposal for any continuous state space.

- **Why MH works?** (MLPP 24.3.6): To prove that the MH procedure samples from $\mathcal{P}^*$, we need to see that it defines a Markov chain with the following transition matrix:

$$\mathcal{P}(\mathbf{x}'|\mathbf{x}) = \begin{cases} q(\mathbf{x}'|\mathbf{x})\, r(\mathbf{x}'|\mathbf{x}) & \text{if } \mathbf{x}' \neq \mathbf{x} \\ q(\mathbf{x}|\mathbf{x}) + \sum_{\mathbf{x}' \neq \mathbf{x}} q(\mathbf{x}'|\mathbf{x})(1 - r(\mathbf{x}'|\mathbf{x})) & \text{otherwise} \end{cases} \tag{23.5}$$

This follows from a case analysis: if you move to $\mathbf{x}'$ from $\mathbf{x}$, you must have proposed it (with probability $q(\mathbf{x}'|\mathbf{x})$) and it must have been accepted (with probability $r(\mathbf{x}'|\mathbf{x})$); otherwise you stay in state $\mathbf{x}$, either because that is what you proposed (with probability $q(\mathbf{x}|\mathbf{x})$), or because you proposed something else (with probability $q(\mathbf{x}'|\mathbf{x})$) but it was rejected (with probability $1 - r(\mathbf{x}'|\mathbf{x})$).

Let us analyze this Markov chain (see Section 16). A chain satisfies **detailed balance** if:

$$\mathcal{P}(\mathbf{x}'|\mathbf{x})\, \mathcal{P}^*(\mathbf{x}) = \mathcal{P}(\mathbf{x}|\mathbf{x}')\, \mathcal{P}^*(\mathbf{x}') \tag{23.6}$$

We also showed that if a chain satisfies detailed balance, then $\mathcal{P}^*$ is its stationary distribution. We can prove that the MH algorithm defines a transition function that satisfies detailed balance and hence that $\mathcal{P}^*$ is its stationary distribution. Furthermore, from Theorem 16.1, this distribution is unique, since the chain is ergodic and irreducible.

- Suppose we have a set of models with different numbers of parameters, e.g. mixture models in which the number of mixture components is unknown. The difficulty with this approach arises when we move between models of different dimensionality. The trouble is that when we compute the MH acceptance ratio, we are comparing densities defined in different dimensionality spaces, which is meaningless. It is like trying to compare a sphere with a circle. One solution is to use **reversible jump MCMC (RJMCMC)** (MLPP 24.3.7), where you augment the low dimensional space with extra random variables so that the two spaces have a common measure.
- We start MCMC from an arbitrary initial state. Only when the chain has "forgotten" where it started from will the samples be coming from the chain's stationary distribution. Samples collected before the chain has reached its stationary distribution do not come from $\mathcal{P}^*$, and are usually thrown away. The initial period, whose samples will e ignored is called the **burn-in phase** (MLPP 24.4.1). The amount of time it takes for a Markov chain to converge to the stationary distribution, and forget its initial state, is called the **mixing time**. It is difficult to diagnose when the chain has burned in - this is one of the fundamental weaknesses of MCMC.
- A natural question to ask is: how many chains should we run (MLPP 24.4.5)? We could either run one long chain to ensure convergence, and then collect samples spaced far apart, or we could run many short chains, but that wastes the burning time. In practice, it is common to run a medium number of chain (say 3) of medium length (say 100,000 steps), and to take samples from each after discarding the first half of the samples. If we initialize at a local mode, we may be able to use all samples, and not wait for burn-in.
- Sometimes we can improve the efficiency of MCMC sampling by introducing dummy **auxiliary variables** (MLPP 24.5), in order to reduce correlation between the original variables. If the original variables are denoted by $\mathbf{x}$, and the auxiliary variables by $\mathbf{z}$, we require that $\sum_{\mathbf{z}} \mathcal{P}(\mathbf{x}, \mathbf{z}) = \mathcal{P}(\mathbf{x})$, and that $\mathcal{P}(\mathbf{x}, \mathbf{z})$ is easier to sample from than just $\mathcal{P}(\mathbf{x})$. If we meet these two conditions, we can sample in the enlarged model, and then throw away the sampled $\mathbf{z}$ values, thereby recovering samples from $\mathcal{P}(\mathbf{x})$.
- One method using auxiliary variables is **slice sampling** (MLPP 24.5.2). Let's consider sampling from a univariate, but multimodal distribution $\mathcal{P}(x)$. Slice sampling will improve our ability to make large moves by adding an auxiliary variable $z$. We define a joint distribution as follows:

$$\tilde{\mathcal{P}}(x, z) \;=\; \begin{cases} 1/Z_p & \text{if } 0 \le z \le \tilde{\mathcal{P}}(x) \\ 0 & \text{otherwise} \end{cases} \tag{23.7}$$

  where $Z_p = \int \tilde{\mathcal{P}}(x)dx$, i.e. $Z_p$ is the normalization factor for the distribution $\mathcal{P}(x)$. The marginal distribution over $x$ is given by:

$$\int \tilde{\mathcal{P}}(x, z)dz \;=\; \int_0^{\tilde{\mathcal{P}}(x)} \frac{1}{Z_p} dz \;=\; \frac{\tilde{\mathcal{P}}(x)}{Z_p} \;=\; \mathcal{P}(x) \tag{23.8}$$

  so we can sample from $\mathcal{P}(x)$ by sampling from $\tilde{\mathcal{P}}(x, z)$ and then ignoring $z$. The full conditionals have the form:

$$\mathcal{P}(z|x) \;= U_{[0,\tilde{\mathcal{P}}(x)]}(z) \tag{23.9}$$

$$\mathcal{P}(x|z) \;= U_A(x) \tag{23.10}$$

  where $A = \{x \,:\, \tilde{\mathcal{P}}(x) \ge z\}$ is the set of points on or above the chosen height $z$. This corresponds to a slice through the distribution, hence the term slice sampling.

  In practice, it can be difficult to identify the set $A$. So we can use the following approaches: construct as interval $x_{\min} \le x \le x_{\max}$ around the current sample $x^{(s)}$ of some width. We then test to see if each end point lies within the slice. If it does, we keep extending in the direction until it lies outside the slice. This is called

**stepping out**. A candidate value $x'$ is then chosen uniformly from this region. If lies within the slice, it is kept, so $x^{(s+1)} := x'$. Otherwise, we shrink the region such that $x'$ forms one end and such that the region still contains $x^{(s)}$. Then another sample is drawn. We continue in this way until a sample is accepted. To apply to a multivariate distributions, we can sample one extra auxiliary variable for each dimension. The advantage of slice sampling over Gibbs is that it does not need a specification of the full conditionals, just the unnormalized joint. The advantage of slice sampling over MH is that it does not need a user-specified proposal distribution.

- For an Ising model of the form $\mathcal{P}(\mathbf{x}) = \frac{1}{Z}\prod_e f_e(\mathbf{x}_e)$, where the edge factor $f_e$ is defined by $\begin{bmatrix} e^J & e^{-J} \\ e^{-J} & e^J \end{bmatrix}$, where $J$ is the edge strength. Gibbs sampling in such models can be slow when $J$ is large in absolute value, because neighboring states can be highly correlated. The **Swendsen Wang** algorithm (MLPP 24.5.3) is a auxiliary variable MCMC sampler, which mixes much faster, at least for the case of attractive models with $J > 0$. The key idea is to split the model into:

$$g_e(\mathbf{x}_e, z_e = 0) = \begin{bmatrix} e^{-J} & e^{-J} \\ e^{-J} & e^{-J} \end{bmatrix}, \qquad g_e(\mathbf{x}_e, z_e = 1) = \begin{bmatrix} e^J - e^{-J} & 0 \\ 0 & e^J - e^{-J} \end{bmatrix}, \qquad \sum_{z_e=0}^{1} g_e(\mathbf{x}_e, z_e) = f_e(\mathbf{x}_e) \tag{23.11}$$

  So if we sample from $g_e$ and then throw away $\mathbf{z}$ samples, we can get valid $\mathbf{x}$ samples from the original distribution. This is simply done by first collecting connecting components with the same label. Each connected component is split into multiple components by setting edges to 0 with the probability $p = e^{-2J}$. We randomly select one of these sub-components and uniformly at random switch the state $\pm 1$ for all the variables in that component. This would be a new sample $\mathbf{x}^{(s+1)}$.

- **Simulated Annealing (SA)** (MLPP 24.6.1) is a stochastic algorithm that attempts to find the global optimum of a black-box function $f(\mathbf{x})$. It is closely related to the MH algorithm for generating samples from a probability distribution. SA can be used for generating samples for both discrete and continuous distribution. The key quantity is the **Boltzmann distribution** which specifies that the probability of being in any particular state $\mathbf{x}$ is given by:

$$\mathcal{P}(\mathbf{x}) \propto \exp\left(-f(\mathbf{x})/T\right) \tag{23.12}$$

  where $f(\mathbf{x})$ is the "energy" of the system and $T$ is the **temperature**. As the temperature approaches 0 (so the system is cooled), the system spends more and more time in its minimum energy (most probable) state. At high temperatures $R \gg 1$, the surface approximately flat, and hence it is easy to move around (i.e. to avoid local optima). As the temperature cools, the largest peaks become larger, and the smallest peaks disappear. By cooling slowly enough, it is possible to "track" the largest peak, and thus find the global optimum. This is an example of a **continuation method**.

  We can generate an algorithm from this as follows. At each step, sample a new state according to some proposal distribution $\mathbf{x}' \sim q(\cdot|\mathbf{x}^{(s)})$. For real-valued parameters, this is often simply a random walk proposal, $\mathbf{x}' = \mathbf{x}^{(s)} + \epsilon_s$, where $\epsilon_s \sim \mathcal{N}(\mathbf{0}, \mathbf{\Sigma})$. For discrete optimization, other kinds of local moves must be defined. Having proposed a new state, we compute:

$$\alpha = \exp\left(\left(f(\mathbf{x}^{(s)}) - f(\mathbf{x}')\right)/T\right) \tag{23.13}$$

  We then accept the new state (i.e. $\mathbf{x}^{(s+1)} := \mathbf{x}'$) with probability $\min(1, \alpha)$, otherwise we stay in the current (i.e. $\mathbf{x}^{(s+1)} := \mathbf{x}^{(s)}$). This means that if the new state has lower energy (is more probable), we will definitely accept it, but if it has higher energy (i.e. less probable), we might still accept, depending on the current temperature. Thus the algorithm allows "down-hill" moves in probability (up-hill in energy space), but less frequently as the temperature drops.

The rate at which the temperature changes over time is called the **cooling schedule**. It has been shown that if one cools sufficiently slowly, the algorithm will provably find the global optimum. Finding the best cooling schedule for a problem is the major drawback of simulated annealing. For MAP state estimation, we just return the state with the largest value found once the algorithm converges.

# 24 Decision Theory

- (AI R&N 16.1) The agent's preferences are captured by a **utility function**, $U(s)$, which expresses desirability of a given state $s$. The **expected utility** can be given by:

$$EU(a|\mathbf{e}) \;=\; \sum_{s'} \mathcal{P}(R(a) = s' \,|\, a, \mathbf{e})\, U(s') \tag{24.1}$$

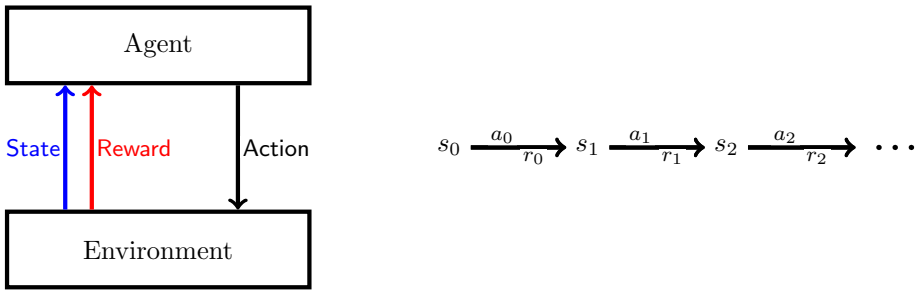
  where $\mathcal{P}(R(a) = s' \,|\, a, \mathbf{e})$ is the probability of outcome $s'$, given evidence observations $\mathbf{e}$, and the action $a$. The principle of **maximum expected utility (MEU)** says that a rational agent should choose the action that maximizes the agent's expected utility:

$$a \;=\; \arg\max_a EU(a|\mathbf{e}) \tag{24.2}$$

- Axioms of utility theory (AI R&N 16.2.1)
- (AI R&N 16.3.3) Usually we are really working with estimates $\hat{EU}(a|\mathbf{e})$ of the true expected utility. We will assume, kindly perhaps, that the estimates are **unbiased**, that is, the expected value of the error, $\mathbb{E}(EU(a|\mathbf{e}) - \hat{EU}(a|\mathbf{e}))$, is zero. In that case, it still seems reasonable to choose the action with the highest estimated utility and to expect to receive that utility, on average, when the action is executed.
- (AI R&N 16.5) **Decision networks** (or **influence diagrams**) combine Bayesian networks with additional node types for actions and utilities. In its most general form, a decision network represents information about the agents current state, its possible actions, the state that will result from the agents action, and the utility of that state. **Chance nodes** (oval) represent random variables; **Decision nodes** (rectangle) represents where we have a choice of actions; **Utility nodes** (diamond) represents the agent's utility function.
- **Information value theory** (AI R&N 16.6) enables an agent to choose what information to acquire.
- (AI R&N 16.6) Sometimes, solving a problem involves finding more information before making a decision. The **value of information** is defined as the expected improvement in utility compared with making a decision without the information.

# 25 Complex Decisions and Reinforcement Learning

- Reinforcement learning (RL) addresses the question of how an autonomous **agent** that senses and acts in its environment can learn to choose optimal actions to achieve its goals (Mitchell 13). Each time the agent performs an action in its environment, a trainer may provide a **reward** or **penalty** to indicate the desirability of the resulting state. The task of the agent is to learn from this indirect, delayed reward, to choose sequences of actions that produce the greatest cumulative reward. **Q-learning** is one algorithm which can acquire optimal control strategies from delayed reward, even when the agent has no prior knowledge of the effects of its actions on the environment.
- The agent can do two things: (1) observe the **state of its environment**; or (2) perform a set of **actions to alter this state**. The aim of the agent would be to learn a **policy** (also called **control strategy**) for choosing actions that achieve its goals. We assume that the goals of the agent can be defined by a **reward function** that assigns a numerical value – an immediate payoff – to each distinct action the agent may take




(a) An agent interacting with its environment. It can perform any action $a \in \mathcal{A}$ given its state $s \in \mathcal{S}$. Each time it reaches $a_t$, the trainer provides a reward $r_t$ for ending up in state $s_{t+1}$ - this indicates the desirability of the state-action transition. The agent's task is to learn a control policy $\pi : \mathcal{S} \to \mathcal{A}$, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.

Figure 21

from each distinct state. In general, we are interested in any type of agent that must learn to choose actions that alter the state of its environment and where a cumulative reward function is used to define the quality of any given action. Here we will consider settings in which the actions have deterministic or non-deterministic outcomes, and settings in which the agent has or does not have prior knowledge about the effects of its actions on the environment.

- The target function to be learned in RL is a control policy $\pi : \mathcal{S} \to \mathcal{A}$, that outputs an appropriate action $a$ from the set $\mathcal{A}$, given the current state $s$ from the set $\mathcal{S}$ (see Figure 21a) (Mitchell 13.1). But note that this is different from other function approximation tasks in several ways:

  1. **Delayed reward**: We are trying to learn the function $\pi$, where given a state, you are trying to learn an optimal action $a = \pi(s)$. Previously in such settings, training is in the form of $(s_1, a_1), (s_2, a_2), \ldots, (s_n, a_n)$. In RL, however, training information is not available in this form, rather the trainer provides only a sequence of immediate reward values as the agent executes its sequence of actions. In other words, RL differs from supervised learning that there is no presentation of input/output pairs. Instead, after choosing an action the agent is told the immediate reward and the subsequent state, but is *not told which action would have been in its best long-term interests*. The agent therefore faces the problem of **temporal credit assignment**: determining which of the actions in the sequence are to be credited with producing the eventual rewards.
  2. **Exploration**: In RL, the agent influences the distribution of training examples by the action sequence it chooses. This raises the question of which experimentation strategy produces most effective training. Should the agent explore the unknown states and actions, or exploit states and actions that the agent has learned to lead to high cumulative reward.
  3. **Partially observable states**: Most of the time the agent cannot observe/sense its complete state. For instance a robot's forward facing camera would not know what is behind it. In such cases an agent might take actions to improve the observability of its environment - plus the agent might not consider its past observations before taking actions.
  4. **Life-long learning**: Unlike isolated function approximation tasks, robot learning often requires the robot to learn several related tasks within the same environment, using the same sensors (dock to the charger, take out the trash, etc). This raise the possibility of learning from previous experience to reduce sample complexity when learning new tasks.

- At each discrete time step $t$ in a **Markov Decision Process (MDP)** (Mitchell 13.2), the agent senses the current state $s_t$, chooses a current action $a_t$, and performs it. The environment responds by giving the agent

a reward $r_t \triangleq r(s_t, a_t)$ and by producing a the succeeding state $s_{t+1} = \delta(s_t, a_t)$. The $\delta$ function is called **transition model** - where the outcome can also be stochastic $\mathcal{P}(s'|s_t, a_t)$ as we'll see later. We assume the Markovian property i.e. the probability of reaching state $s'$ from $s$ depends only on $s$ and not the history of earlier states. Here the functions $\delta$ and $r$ are part of the environment and are not necessarily known to the agent. In an MDP, the function $\delta(s_t, a_t)$ and $r(s_t, a_t)$ depend only on the current state and action, and not on earlier states or actions. In short, an MDP is defined by the set of states $\mathcal{S}$, the set of actions $\mathcal{A}$, and the two functions $\delta(\cdot, \cdot)$, and $r(\cdot, \cdot)$.

The task of the agent is to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$, for selecting its next action $a_t$ based on the current observed state, i.e. $\pi(s_t) = a_t$. One obvious objective function would require to learn the policy that produces the greatest possible cumulative reward for the robot over time. From state $s_t$ you would like to maximize the cumulative value $V^\pi(s_t)$ (or maximize your utility):

$$V^\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \tag{25.1}$$

$$= \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{25.2}$$

where the sequence of rewards $r_{t+i}$ is generated by beginning at state $s_t$ and by repeatedly using the policy $\pi$ to select actions as described above (i.e. $a_t = \pi(s_t)$, $a_{t+1} = \pi(s_{t+1})$ etc.). Here $0 \leq \gamma < 1$ is a constant that determines the relative value of delayed versus immediate rewards. In particular, rewards received $i$ time steps into the future are discounted exponentially by a factor of $\gamma^i$. Note if $\gamma = 0$, only the immediate reward is considered. In this form, the term is $V^\pi(s)$ is often referred to as **discounted cumulative rewards** (also known as **infinite-horizon discounted model**). Other possible rewards are the **finite horizon reward**, $\sum_{i=0}^{h} r_{t+i}$, and the **average reward**, $\lim_{h\to\infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}$. One problem with the average reward model is that there is no way to distinguish between two policies, one of which gains a large amount of reward in the initial phases and the other of which does not. The finite horizon model is appropriate when the agent's lifetime is known; one important aspect of this model is that as the length of the remaining lifetime decreases, the agent's policy may change. A system with a hard deadline would be appropriately modeled this way.

We are now in a position to state precisely the agent's learning task. We require that the agent learn a policy $\pi$ that maximizes $V^\pi(s)$ for all states $s$. We will call this an **optimal policy** and denote it by $\pi^*$:

$$\pi^* = \arg\max_\pi V^\pi(s), \quad \forall s \tag{25.3}$$

We will denote $V^{\pi^*}(s) \triangleq V^*(s)$, which gives the maximum discounted cumulative reward that the agent can obtain starting from state $s$.

- The **goal state** is the state where the agent should eventually reach. Once the agent reaches that state, it should not take an action to go out of it. That is why it is also as an **absorbing state**. A policy that is guaranteed to reach a terminal state is called a **proper policy**.
- (AI R&N 17.1.2) $\pi$ essentially tells the agent what action to execute at state $s$. The expected utility obtained by executing policy $\pi$:

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right] \tag{25.4}$$

where the $R(s_t)$ is considered as a random variable for the reward received for being in some state at time $t$. Hence the expectation is with respect to the probability distribution over state sequences determined by $s$ and $\pi$. As before, if we start at state $s$, we will choose the policy $\pi_s^*$ with the highest expected utility: $\pi_s^* = \arg\max_\pi V^\pi(s)$. The utility function $V(s)$ allows the agent to select actions by using the principle of maximum expected utility:

$$\pi^*(s) = \arg\max_{a\in\mathcal{A}} \sum_{s'} \mathcal{P}(s'|s, a) V(s') \tag{25.5}$$

Note that $\mathcal{A} \triangleq \mathcal{A}(s)$ i.e. we only consider actions possible at a particular state.

- One way to find an optimal policy is through **Value Iteration** (AI R&N 17.2). The basic idea is to calculate the utility of each state and then use the state utilities to select an optimal action in each state.

  There is a direct relationship between the utility of a state and the utility of its neighbors: *the utility of a state is the immediate reward for that state plus the expected discounted utility of the next state, assuming that the agent chooses the optimal action.* That is, the utility of a state is given by:

$$V(s) = \underbrace{R(s)}_{\text{Reward at } s} + \gamma \max_{a \in \mathcal{A}} \underbrace{\sum_{s'} \mathcal{P}(s' \mid s, a)\, V(s')}_{\text{Expected utility on taking action } a} \tag{25.6}$$

  This is called the **Bellman Equation**. The Bellman equation is the basis of the value iteration algorithm for solving MDPs. The value iteration algorithm is an iterative procedure where we initialize $V(s)$ with arbitrary values - then we calculate Equation 25.6 for each state repeatedly until convergence. The iteration step, called the **Bellman update**, looks like this:

$$V_{n+1}(s) := R(s) + \gamma \max_{a \in \mathcal{A}} \sum_{s'} \mathcal{P}(s' \mid s, a)\, V_n(s') \tag{25.7}$$

  If we apply the Bellman update infinitely often, we are guaranteed to reach an equilibrium, in which case the final utility values must be solutions to the Bellman equations. In fact, they are also the unique solutions, and the corresponding policy (obtained using Equation 25.5) is optimal.

- Another way to find an optimal policy is through **Policy Iteration** (AI R&N 17.3). Policy iteration alternates the following two steps, beginning from some initial policy $\pi_0$: (1) **Policy evaluation**: given a policy $\pi_n$, calculate $V_n \triangleq V^{\pi_n}$, the utility of each state if $\pi_n$ were to be executed; (2) **Policy improvement**: calculate a new MEU policy $\pi_{n+1}$, using one-step look-ahead based on $V_n$ (as in Equation 25.5). The algorithm terminates when the policy improvement step yields no change in the utilities. At this point, we know that the utility function $V_n$ is a fixed point of the Bellman update, so it is a solution to the Bellman equations, and $\pi_n$ must be an optimal policy.

  The policy improvement step is obviously straightforward, but how do we implement the policy-evaluation routine? It turns out that doing so is much simpler than solving the standard Bellman equations (which is what value iteration does), because the action in each state is fixed by the policy. At the $n$th iteration, the policy $\pi_n$ specifies the action $\pi_n(s)$ in state $s$. This means that we have a simplified version of the Bellman equation 25.6 relating the utility of $s$ (under $\pi_n$) to the utilities of its neighbors:

$$V_n(s) = R(s) + \gamma \sum_{s'} \mathcal{P}(s' \mid s, \pi_n(s))\, V_n(s') \tag{25.8}$$

  For policy improvement, you iterate over every state $s$ and see if:

$$\max_{a \in \mathcal{A}} \sum_{s'} \mathcal{P}(s' \mid s, a)\, V_n(s') > \sum_{s'} \mathcal{P}(s' \mid s, \pi_n(s))\, V_n(s') \tag{25.9}$$

  if so, then the policy for that state $s$ is changed $\pi_{n+1}(s) := \arg\max_{a \in \mathcal{A}} \sum_{s'} \mathcal{P}(s' \mid s, a)\, V_n(s')$.

- The algorithms we have described so far require updating the utility or policy for all states at once. It turns out that this is not strictly necessary. In fact, on each iteration, we can pick any subset of states and apply either kind of updating (policy improvement or simplified value iteration) to that subset. This very general algorithm is called **asynchronous policy iteration**.

- Unlike before, where the agent always knew what the state is, the environment could only be partially observable. The agent does not necessarily know which state it is in, so it cannot execute the action $\pi(s)$

recommended for that state. Furthermore, the utility of a state $s$ and the optimal action in $s$ depend not just on $s$, but also on how much the agent knows when it is in $s$. For these reasons, **partially observable MDPs (POMDPs)** (AI R&N 17.4) are usually viewed as much more difficult than ordinary MDPs.

A POMDP has the same elements as an MDP the transition model $\mathcal{P}(s'|s,a)$, actions $A$ (or $A(s)$ if the number of actions are limited by the state you are in), and reward function $R(s)$ but, in addition, it also has a **sensor model** $\mathcal{P}(e \mid s)$. The sensor model specifies the probability of perceiving evidence $e$ in state $s$. This could be extended to being conditioned on the action and the output state: $\mathcal{P}(e \mid s, a, s')$. In addition **belief state** $b(s)$ - which is the distribution of what the agent thinks its actual state is. For instance the agent can start with a uniform belief, meaning that its equally likely that it is anywhere. The agent can calculate its current belief state as the conditional probability distribution over the actual states given the sequence of percepts and actions so far. This is essentially done by filtering, where if the previous belief was $b(s)$, and the agent does an action $a$ and then perceives evidence $e$, then the new belief state is:

$$b'(s') = \alpha \mathcal{P}(e|s') \sum_s \mathcal{P}(s'|s,a)\, b(s) \tag{25.10}$$

$$b' = \textsc{Forward}(b, a, e) \tag{25.11}$$

The fundamental insight required to understand POMDPs is this: *the optimal action depends only on the agent's current belief state.* That is, the optimal policy can be described by a mapping $\pi^*(b)$ from belief states to actions. The decision cycle of a POMDP agent can be broken down into the following three steps: (1) Given the current belief state b, execute the action $a = \pi^*(b)$; (2) Receive percept $e$; (3) Set the current belief state to Equation 25.11. Now we can think of POMDPs as requiring a search in belief-state space. Note that the POMDP belief-state space is continuous, because a POMDP belief state is a probability distribution. An action changes the belief state, not just the physical state. Hence, the action is evaluated at least in part according to the information the agent acquires as a result.

If we calculate the probability that an agent in belief state $b$ reaches belief state $b'$ after executing action $a$. Now, if we knew the action and the subsequent percept, then Equation 25.11 would provide a deterministic update to the belief state: $b' = \textsc{Forward}(b, a, e)$:

$$\mathcal{P}(b' \mid b, a) = \sum_e \mathcal{P}(b' \mid e, a, b) \sum_{s'} \mathcal{P}(e \mid s') \sum_s \mathcal{P}(s' \mid s, a)\, b(s) \tag{25.12}$$

where $\mathcal{P}(b' \mid e, a, b) = 1$ if $b' = \textsc{Forward}(b, a, e)$ and 0 otherwise. Equation 25.12 can be also viewed as the transition model for the belief state. We can also define a reward function for belief states (i.e., the expected reward for the actual states the agent might be in):

$$\rho(b) = \sum_s b(s) R(s) \tag{25.13}$$

Together, $\mathcal{P}(b' \mid b, a)$ and $\rho(b)$ define an *observable* MDP on the space of belief states. Furthermore, it can be shown that an optimal policy for this MDP, $\pi^*(b)$, is also an optimal policy for the original POMDP. In other words, *solving a POMDP on a physical state space can be reduced to solving an MDP on the corresponding belief-state space.* This fact is perhaps less surprising if we remember that the belief state is always observable to the agent, by definition.

- **Value iteration of POMDPs** (AI R&N 17.4.2): Now that we have infinitely many belief states (since its probabilities on beliefs), we can't use the old value iteration algorithm. We make two observations:

  1. Let the utility of executing a fixed conditional plan $p$ starting in physical state $s$ be $\alpha_p(s)$. Then the expected utility of executing $p$ in belief state $b$ is just $\sum_s b(s)\alpha_p(s)$, or $b \cdot \alpha_p$ if we think of them both as vectors. Hence, the expected utility of a fixed conditional plan varies linearly with $b$; that is, it corresponds to a hyperplane in belief space.

2. At any given belief state $b$, the optimal policy will choose to execute the conditional plan with highest expected utility; and the expected utility of $b$ under the optimal policy is just the utility of that conditional plan:

$$V(b) = V^{\pi^*}(b) = \max_p b \cdot \alpha_p \tag{25.14}$$

If the optimal policy $\pi^*$ chooses to execute $p$ starting at $b$, then it is reasonable to expect that it might choose to execute $p$ in belief states that are very close to $b$.

From these two observations, we see that the utility function $V(b)$ on belief states, being the maximum of a collection of hyperplanes, will be piecewise linear and convex. In general, let $p$ be a depth-$d$ conditional plan whose initial action is a and whose depth-$d-1$ subplan for percept $e$ is $p.e$; then

$$\alpha_p(s) = R(s) + \gamma \left( \sum_{s'} \mathcal{P}(s' \mid s, a) \sum_e \mathcal{P}(e \mid s') \alpha_{p.e}(s') \right) \tag{25.15}$$

The recursion gives us a value iteration algorithm. The structure of the algorithm and its error analysis are similar to those of the basic value iteration algorithm; the main difference is that instead of computing one utility number for each state, POMDP-value iteration maintains a collection of undominated plans with their utility hyperplanes.

- Up until now we have concentrated on making decisions in uncertain environments. But what if the uncertainty is due to other agents and the decisions they make? And what if the decisions of those agents are in turn influenced by our decisions? **Game theory** studies exactly this problem, and analyzes games with simultaneous moves and other sources of partial observability. *Game theory describes rational behavior for agents in situations in which multiple agents interact simultaneously. Solutions of games are **Nash equilibria**-strategy profiles in which no agent has an incentive to deviate from the specified strategy.* Game theory can be used in at least two ways:

  1. **Agent design**: Game theory can analyze the agent's decisions and compute the expected utility for each decision (under the assumption that other agents are acting optimally according to game theory).
  2. **Mechanism design**: When an environment is inhabited by many agents, it might be possible to define the rules of the environment (i.e., the game that the agents must play) so that the collective good of all agents is maximized when each agent adopts the game-theoretic solution that maximizes its own utility.

- A **single-move game** (AI R&N 17.5.1) is one where all players take actions simultaneously and result of the game is based on single set of actions. It has three components: (1) players; (2) actions - may are may not be the same for all players; (3) **payoff function** that gives the utility to each player for each combination of actions by all the players. Each player in a game must adopt and then execute a strategy (which is the name used in game theory for a policy). A **pure strategy** is a deterministic policy; for a single-move game, a pure strategy is just a single action. For many games an agent can do better with a **mixed strategy**, which is a randomized policy that selects actions according to a probability distribution. A **solution** to a game is a strategy profile in which each player adopts a rational strategy.

  We say that a strategy $s$ for player $p$ **strongly dominates** strategy $s'$ if the outcome for $s$ is better for $p$ than the outcome for $s'$, for every choice of strategies by the other player(s). Strategy $s$ **weakly dominates** $s'$ if $s$ is better than $s'$ on at least one strategy profile and no worse on any other. It is **rational** to play a dominated strategy. Dominant strategy means that an agent can adopt it without regard for the other strategies.

  We say that an outcome is **Pareto optimal** if there is no other outcome that all players would prefer. An outcome is **Pareto dominated** by another outcome if all players would prefer the other outcome.

  When each player has a dominant strategy, the combination of those strategies is called a **dominant strategy equilibrium**. In general, a strategy profile forms an **equilibrium** if no player can benefit by switching

strategies, given that every other player sticks with the same strategy. An equilibrium is essentially a local optimum in the space of policies. The mathematician John Nash proved that *every game has at least one equilibrium*. The general concept of equilibrium is now called **Nash equilibrium** in his honor. Clearly, a dominant strategy equilibrium is a Nash equilibrium, but some games have Nash equilibria but no dominant strategies.

- (AI R&N 17.5.2) The simplest kind of multiple-move game is the **repeated game**, in which players face the same choice repeatedly, but each time with knowledge of the history of all players previous choices. A strategy profile for a repeated game specifies an action choice for each player at each time step for every possible history of previous choices.

  In repeated games, where the number of games is fixed, it is quite usual that the players don't cooperate. If we are having $N$ games, the players are going to consider the $N$th game as a single-move game, and hence the outcome will be fixed by the dominant strategy equilibrium. The $N-1$ game would now also be treated similarly because the next game is fixed. By induction, both players will choose the dominant strategy equilibrium for all games.

  Players are more likely to cooperate when they don't know which is the last game. For example, one equilibrium strategy is for each player to be considerate unless the other player has ever been inconsiderate. This strategy could be called **perpetual punishment**. At every step, there is no incentive to deviate from being considerate. Perpetual punishment is the *mutually assured destruction* strategy of the prisoner's dilemma: once either player decides to be inconsiderate, it ensures that both players suffer a great deal. But it works as a deterrent only if the other player believes you have adopted this strategyor at least that you might have adopted it. Other strategies are more forgiving. The most famous, called **tit-for-tat**, calls for starting with being considerate and then echoing the other player's previous move on all subsequent moves.

- **Mechanism design** (AI R&N 17.6) can be used to set the rules by which agents will interact, in order to maximize some global utility through the operation of individually rational agents. Sometimes, mechanisms exist that achieve this goal without requiring each agent to consider the choices made by other agents.

- In **Reinforcement Learning (RL)** doesn't have a complete model of the environment and doesn't know what the reward function is. In this case, RL observes rewards and learns an (nearly) optimal policy for the environment.

- $Q$ **Learning** (Mitchell 13.3) What evaluation function should the agent attempt to learn? One obvious choice is $V^*$. The agent should prefer state $s_1$ over state $s_2$ whenever $V*(s_1) > V*(s_2)$, because the cumulative future reward will be greater from $s_1$. Of course, the agent's policy must choose among actions, not among states. The optimal action in state $s$ is the action $a$ that maximizes the sum of the immediate reward $r(s,a)$ plus the value $V^*$ of the immediate successor state, discounted by $\gamma$:

  $$\pi^*(s) \;=\; \arg\max_a \left[r(s,a) \;+\; \gamma V^*(\delta(s,a))\right] \tag{25.16}$$

  Thus, the agent can acquire the optimal policy by learning $V^*$, provided it has perfect knowledge of the immediate reward function $r$ and the state transition function $\delta$. When the agent knows about $r$ and $\delta$ used by the environment to respond to its actions, then it can then use the Equation 25.16 to calculate the optimal action at any state $s$. This scheme is not always feasible because the agent needs to have perfect knowledge of the functions $\delta$ and $r$, which is sometimes not the case. In cases where these functions are unknown, learning $V^*$ is of no use for selecting the optimal policy. In a more general setting, the agent can use the evaluation function described next.

- Let $Q(s,a)$ be the maximum reward when the agent takes actions $a$ at state $s$:

  $$Q(s,a) \;=\; r(s,a) \;+\; \gamma V^*(\delta(s,a)) \tag{25.17}$$

  Note in Equation 25.16, we had $\pi^*(s) \;\triangleq\; \arg\max_a Q(s,a)$. This transformation is important because if the agent learns this $Q$ **Function** (Mitchell 13.3.1) instead of the $V^*$ function, it will be able to select optimal

actions even when it has no knowledge of the functions $r$ and $\delta$. Part of the beauty of $Q$ learning is that the evaluation function is defined to have precisely this property–the value of $Q$ for the current state and action summarizes in a single number all the information needed to determine the discounted cumulative reward that will be gained in the future if action $a$ is selected in state $s$.

- Now, how to learn the $Q$ Function (Mitchell 13.3.2)? The key problem is finding a reliable way to estimate the training values for $Q$, given only a sequence of immediate rewards $r$ spread out over time. This can be achieved through iterative approximation. The first thing to notice is the relationship between $Q$ and $V^*$:

$$V^*(s) = \max_{a'} Q(s, a') \tag{25.18}$$

Using this we can rewrite Equation 25.17 as

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a') \tag{25.19}$$

This equation says that given that the agent was in state $s$ and it took action $a$, it ended up in state $s' \triangleq \delta(s, a)$. Given this information, we can choose an action $a'$ from $s'$ which would maximize the agent's discounted cumulative reward. This indeed points to an iterative algorithm. You can start with $Q$ which is set to all zero, and apply Equation 25.19 at each step to incrementally build the $Q$ function. One important thing to note is that the agent uses its current $\hat{Q}$ values for the new state $s'$ to refine its of $\hat{Q}(s, a)$ for the previous state $s$. Another key point to observe is even though Equation 25.19 is defined in terms of $r$ and $\delta$, the agent does not need to know these general functions to apply the training rule. The Q learning algorithm for deterministic MDP is given in Algorithm 8. In the algorithm we will use the symbol $\hat{Q}$ to refer to the learner's estimate, or hypothesis, of the actual $Q$ function. $\hat{Q}$ can be represented as a table with state-action pairs.

**Algorithm 8:** $Q$ Learning algorithm for deterministic MDP

```
Initialize Q̂(s, a) := 0 for all s and a
Observe the current state s
repeat
    Select an action a and execute it
    Receive immediate reward r(s, a)
    Observe new state s'
    Update the table entry for Q̂(s, a) as follows (given by Equation 25.19):
        Q̂(s, a) := r(s, a) + γ max_a' Q̂(s', a')
    s := s'
until forever
```

Using the algorithm the agent's estimate $\hat{Q}$ converges in the limit to the actual $Q$ function, provided the system can be modeled as a deterministic MDP, the reward function $r$ is bounded, and actions are chosen so that every state-action pair is visited infinitely often.

- Note that if we start with $Q$ being zero everywhere, the agent will make no changes to any $\hat{Q}$ table entry until it happens to reach the goal state and receive a nonzero reward. This will result in refining the $\hat{Q}$ value for the single transition leading to the goal state. Of course, then doing only one experiment would not help in making $\hat{Q}$ converge to $Q$. A remedy is to run multiple **episodes** of experiments for training. In each episode the agent starts at a randomly chosen state and is allowed to execute actions until it reaches the absorbing goal state. *If we run repeated identical episodes in this fashion, the frontier of nonzero $\hat{Q}$ values will creep backward from the goal state at the rate of one new state-action transition per episode.*

- Two general properties of $Q$ learning algorithm that hold for any deterministic MDP in which the rewards are non-negative, assuming we initialize $\hat{Q}$ to all zeros: (1) the values of $\hat{Q}$ will never decrease during training

$(\hat{Q}_{n+1}(s,a) \geq \hat{Q}_n(s,a))$; (2) through-out the training process every $\hat{Q}$ value will remain in the interval $0 \leq \hat{Q}_n(s,a) \leq Q(s,a)$.

**Does $Q$ learning converge** (Mitchell 13.3.4)?:

**Theorem 25.1.** *Consider a Q learning agent in a deterministic MDP with bounded rewards* $(\forall s,a) \quad |r(s,a)| \leq c$. *The Q learning agent uses the training rule of Equation 25.19, initializes its table* $\hat{Q}(s,a)$ *to arbitrary finite values, and uses a discount factor* $\gamma$ *such that* $0 \leq \gamma < 1$. *Let* $\hat{Q}_n(s,a)$ *denote the agent's hypothesis* $\hat{Q}(s,a)$ *following the nth update. If each state-action pair is visited infinitely often, then* $\hat{Q}_n(s,a)$ *converges to* $Q(s,a)$ *as* $n \to \infty$ *for all* $s,a$.

They key idea underlying the proof of convergence is that the table entry $\hat{Q}(s,a)$ with the largest error must have its error reduced by a factor of $\gamma$ whenever it is updated. The reason is that its new value depends only in part on error prone $\hat{Q}$ estimates, with the remainder depending on the error-free observed immediate reward $r$. $\hat{Q}_n$ is the agent's table of estimated $Q$ values after $n$ updates. Let $\Delta_n$ be the maximum error in $\hat{Q}_n$:

$$\Delta_n \triangleq \max_{s,a} |\hat{Q}_n(s,a) - Q(s,a)| \tag{25.20}$$

The updated $\hat{Q}_{n+1}(s,a)$ for any $s,a$ is at most $\gamma$ times the maximum error in the $\hat{Q}_n$ table, $\Delta_n$. The largest error in the initial table, $\Delta_0$, is bounded because values of $\hat{Q}_0(s,a)$ are bounded for all $s,a$. Now after the first interval during which each $s,a$ is visited, the largest error in the table will be at most $\gamma\Delta_0$. After $k$ intervals, the error will be at most $\gamma^k\Delta_0$. Since each state is visited infinitely often, the number of such intervals is infinite, and $\Delta_n \to 0$ as $n \to \infty$. This rule also applies to the Value Iteration using Bellman updates (see Section 25).

- In Algorithm 8, the agent runs the risk that it will over-commit to actions that are found during early training to have high $\hat{Q}$ values. The above theorem required each state-action to occur infinitely many times, which will not occur if we always select actions that maximize the current $\hat{Q}(s,a)$. Hence we take a probabilistic approach (Mitchell 13.3.5), where selecting an action $a_i$ with current state $s$ is given by:

$$\mathcal{P}(a_i|s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}} \tag{25.21}$$

  where $k > 0$ - larger values of $k$ will assign higher probabilities to actions with high $\hat{Q}$ values, causing the agent to exploit what it has learned. In contrast if $k$ is smaller than 1, you will assign higher probabilities to state with low $\hat{Q}$ values. Note, $k = 1$ will result in uniform sampling. You can also make the agent choose a lower $k$ initially to encourage *exploration*, and gradually shift to higher $k$ to encourage *exploitation*.

- (Mitchell 13.3.6) Note that if we start with $\hat{Q}$ set to zero, and we run the first episode - you can run Equation 25.19 in reverse chronological order to update $\hat{Q}$ over the whole state-action path taken in the first episode. Even though this clearly requires more memory (to completely store the state-action transitions) the algorithm will converge in fewer episodes. Another strategy to improve convergence is to store past state-action transitions and after some episodes use them to update $\hat{Q}(s,a)$ values.

- If the agent has the knowledge of the state-transition function $\delta(s,a)$, or the reward function $r(s,a)$, then there are many more efficient methods are possible. For example if performing external actions is expensive the agent may simply ignore the environment and instead simulate it internally, efficiently generating simulated actions and assigning the appropriate simulated rewards. **DYNA** architecture performs a number of simulated actions after each step executed in the external world.

- Now what if the resulting state $s'$ on taking an action $a$ at state $s$ is non-deterministic, or the reward $r(s,a)$ is non-deterministic (Mitchell 13.4). In such cases, the function $\delta(s,a)$ $r(s,a)$ can be viewed as first producing a

probability distribution over outcomes based on $s$ and $a$, and then drawing an outcome at random according to this distribution. When these probability distributions depend solely on $s$ and $a$ (e.g., they do not depend on previous states or actions), then we call the system a **non-deterministic MDP**. We can change the adjust the Q learning algorithm to take this into account. This is done by redefining Equation 25.19 as:

$$Q(s,a) \;=\; \mathbb{E}[r(s,a)] \;+\; \gamma \sum_{s'} \mathcal{P}(s' \mid s,a) \max_{a'} Q(s',a') \tag{25.22}$$

where $\mathcal{P}(s' \mid s,a)$ is the probability of getting to state $s'$ upon taking action $a$ in state $s$. Moreover, rather than getting the deterministic reward $r$, we have extended the model to consider the expectation of the reward given the state $s$ and action $a$.

Now we also need a different training rule for convergence in this non-deterministic setting. This is because the reward would always be different due to its non-deteministic nature. This difficulty can be overcome by modifying the training rule so that it takes a decaying weighted average of the current $\hat{Q}$ value and its revised estimate:

$$\hat{Q}_n(s,a) \;:=\; (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r + \max_{a'} \hat{Q}_{n-1}(s',a')], \qquad \text{where} \quad \alpha_n \;=\; \frac{1}{1+\mathsf{visits}_n(s,a)} \tag{25.23}$$

where $s$ and $a$ here are the state action updated during the $n$th iteration, and where $\mathsf{visits}_n(s,a)$ is the total number of times the state-action pair has been visited up to and including the current iteration. $\alpha_n$ goes down with the number of times you encounter a particular state-action transition.

Deep learning is an unsupervised way to learn good feature representations