CSC 446 Lecture Notes

April 10, 2019

Contents

1 What Is Machine Learning?

Machine learning is about automatically analyzing data; it mostly focuses on the problems of classification and regression. In this class, we will learn multiple methods for solving each of these problems.

- **Classification** is the problem of assigning datapoints to discrete categories; the goal is to pick the best category for each datapoint.
- **Regression** is the problem of learning a function from datapoints to numbers; fitting a line or a curve to the data is an example of a regression problem.

One example of a classification problem would be: given heights and weights, classify people by sex. We are given a number of training datapoints, whose heights, weights, and sexes we know; we can plot these datapoints in a two-dimensional space. Our goal is to learn a rule from these datapoints that will allow us to classify other people whose heights and weights are known but whose sex is unknown. This rule will take the form of a curve in our two-dimensional space: then, when confronted with future datapoints, we will classify those datapoints which fall below the curve as female those which fall above the curve as male. So how should we draw this curve?

One idea would be to draw a winding curve which carefully separates the datapoints, assuring that all males are on one side and all females are on the other. But this is a very complicated rule, and it's likely to match our training data too closely and not generalize well to new data. Another choice would be to draw a straight line; this is a much simpler rule which is likely to do better on new data, but it does not classify all of the training datapoints correctly. This is an example of a fundamental tradeoff in machine learning, that of **overfitting** vs. **generalization**. We will return to this tradeoff many times during this class, as we learn methods of preventing overfitting.

An example of a regression problem would be: given weights of people, predict their heights. We can apply the nearest neighbor model to solve this problem. The nearest neighbor model remembers the weights and corresponding heights of the people in the training data. Then for a new test weight, it looks up the person with the closest weight in the training data, and returns the corresponding height. This results in a piece-wise constant function that may be affected by outliers and may result in overfitting. Another choice would be to fit a straight line.

2 Probability Theory

This section contains a quick review of basic concepts from probability theory.

Let X be a **random variable**, i.e., a variable that can take on various values, each with a certain probability. Let x be one of those values. Then we denote the probability that $X = x$ as $P(X = x)$. (We will often write this less formally, as just $P(x)$, leaving it implicit which random variable we are discussing. We will also use $P(X)$ to refer to the entire probability distribution over possible values of X.)

In order for $P(X)$ to be a valid probability distribution, it must satisfy the following properties:

- For all x, $P(X = x) \geq 0$.
- $\sum_x P(X = x) = 1$ or $\int P(x)\mathrm{d}x = 1$, depending on whether the probability distribution is discrete or continuous.

If we have two random variables, X and Y, we can define the **joint distribution** over X and Y, denoted $P(X = x, Y = y)$. The comma is like a logical "and"; this is the probability that both $X = x$ and $Y = y$. Analogously to the probability distributions for a single random variable, the joint distribution must obey the properties that for all x and for all y, $P(X = x, Y = y) \geq 0$ and either $\sum_{x,y} P(X = x, Y = y) = 1$ or $\int\int P(x, y)\mathrm{d}y\mathrm{d}x = 1$, depending on whether the distribution is discrete or continuous.

From the joint distribution $P(X, Y)$, we can **marginalize** to get the distribution $P(X)$: namely, $P(X = x) = \sum_y P(X = x, Y = y)$. We can also define the **conditional probability** $P(X = x|Y = y)$, the probability that $X = x$ given that we already know $Y = y$. This is $P(X = x|Y = y) = \frac{P(X=x,Y=y)}{P(Y=y)}$, which is known as the **product rule**. Through two applications of the product rule, we can derive **Bayes rule**:

$$P(X = x|Y = y) = \frac{P(Y = y|X = x)P(X = x)}{P(Y = y)}$$

Two random variables X and Y are **independent** if knowing the value of one of the variables does not give us any further clues as to the value of the other variable. Thus, for X and Y independent, $P(X = x|Y = y) = P(X = x)$, or, written another way, $P(X = x, Y = y) = P(X = x)P(Y = y)$.

The **expectation** of a random variable X with respect to the probability distribution $P(X)$ is defined as $E_P[X] = \sum_x P(X = x)x$ or $E_P[X] = \int P(x)x\mathrm{d}x$, depending on whether the random variable is discrete or continuous. The expectation is a weighted average of the values that a random variable can take on. Of course, this only makes sense for random variables which take on numerical values; this would not work in the example from earlier where the two possible values of the "sex" random variable were "male" and "female".

We can also define the **conditional expectation**, the expectation of a random variable with respect to a conditional distribution: $E_{P(X|Y)}[X] = \sum_x P(X = x|Y = y)x$. This is also sometimes written as $E_P[X|Y]$. Lastly, we are not restricted to taking the expectations of random variables only; we can also take the expectation of functions of random variables: $E_P[f(X)] = \sum_x P(X = x)f(x)$. Note that we will often leave the probability distribution implicit and write the expectation simply as $E[X]$.

Expectation is **linear**, which means that the expectation of the sum is the sum of the expectations, i.e., $E[X + Y] = E[X] + E[Y]$, or, more generally, $E[\sum_{i=1}^N X_i] = \sum_{i=1}^N E[X_i]$. For the two-variable case, this can be proven as follows:

$$\begin{aligned}
E[X + Y] &= \sum_{x,y} P(X = x, Y = y)(x + y) \\
&= \sum_{x,y} P(X = x, Y = y)x + \sum_{x,y} P(X = x, Y = y)y \\
&= \sum_x P(X = x)x + \sum_y P(Y = y)y \\
&= E[X] + E[Y]
\end{aligned}$$

The N-variable case follows from this by induction.

For readability, let $\bar{x} = E[X]$. Then we can define the **variance**, $Var[X] = E[(x - \bar{x})^2]$. In words, the variance is the weighted average of the distance from the mean squared. Why is the distance squared? Well, if we take out the square, we get that $Var[X] = E[x - \bar{x}]$, which by linearity of expectation equals $E[x] - E[\bar{x}] = E[X] - \bar{x} = 0$, so we put the square in to keep that from happening. The reason we do not use absolute value instead is that the absolute value function is nondifferentiable. As a result of squaring, the variance penalizes further outliers more.

Unlike expectation, variance is not linear; that means in general $Var[X + Y] \neq Var[X] + Var[Y]$. The covariance of X and Y is defined as: $Cov[X, Y] = E[(X - E[X])(Y - E[Y])]$. We can show that $Var[aX] = a^2 Var[X]$ and $Var[X+Y] = Var[X]+Var[Y]+2Cov[X, Y]$. If X and Y are independent, then $Cov[X, Y] = 0$.

3 Concentration Bounds

Markov's Inequality For a non-negative random variable, $X > 0$, and for any $\delta > 0$,

$$P(X \geq \delta E[X]) \leq \frac{1}{\delta}$$

or equivalently,

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

Proof: Your homework.

Chebyshev's Inequality For any random variable with finite variance σ^2, and for any $k > 0$

$$P(|X - E[X]| \geq k\sigma) \leq \frac{1}{k^2}$$

Proof: Your homework.

4 Maximum Likelihood Estimation

Consider a set of N **independent and identically distributed** random variables $X_1, \ldots, X_N$. "Independent and identically distributed", which is usually abbreviated as i.i.d., means that the random variables are pairwise independent (i.e., for each i, j such that $i \neq j$, X_i and X_j are independent) and that they are all distributed according to the same probability distribution, which we will call P. Much of the data that we will look at in this class is i.i.d. Since our goal is to automatically infer the probability distribution P that is the best description of our data, it is essential to assume that all of our datapoints were actually generated by the same distribution. One of the implications of i.i.d. is that the joint probability distribution over all of the random variables decomposes as follows: $P(X_1, \ldots, X_N) = \prod_{n=1}^{N} P(X_n)$.

Again, our task is to automatically infer the probability distribution P which best describes our data. But how can we quantify which probability distribution gives the best description? Suppose that our random variables are discrete and have K possible outcomes such that each datapoint X takes on a value in $\{1, \ldots, K\}$. We can describe P with a K-dimensional vector that we will call θ, letting $\theta_k = P(X = k)$ for each k; θ is called the **parameters** of the distribution. It's useful to describe the probabilities in our distribution using a vector, because then we can employ the powerful tools of vector calculus to help us solve our problems.

Now the question of finding the best probability distribution becomes a question of finding the optimal setting of θ. A good idea would be to pick the value of θ which assigns the highest probability to the data:

$$\theta^* = \operatorname*{argmax}_{\theta} P(X_1, \ldots, X_N; \theta)$$

This method of estimating θ is called **maximum likelihood estimation**, and we will call the optimal setting of the parameters θ_{MLE}. It is a constrained optimization problem that we can solve using the tools of vector

calculus, though first we will introduce some more convenient notation. For each k, let $c(k) = \sum_{n=1}^{N} I(X_n = k)$ be the number of datapoints with value k. Here, I is an indicator variable which is 1 when the statement in the parentheses is true, and 0 when it is false.

Using these counts, we can rewrite the probability of our data as follows:

$$\begin{aligned} P(X_1, \ldots, X_N|\theta) &= \prod_{n=1}^{N} \theta_{x_n} \\ &= \prod_{k=1}^{K} \theta_k^{c(k)} \end{aligned}$$

This switch in notation is very important, and we will do it quite frequently. Here we have grouped our data according to outcome rather than ordering our datapoints sequentially.

Now we can proceed with the optimization. Our goal is to find $\text{argmax}_\theta \prod_{k=1}^{K} P(X = k)^{c(k)}$ such that $\sum_{k=1}^{K} \theta_k = 1$. (We need to add this constraint to assure that whichever θ we get describes a valid probability distribution.) If this were an unconstrained optimization problem, we would solve it by setting the derivative to 0 and then solving for θ. But since this is a constrained optimization problem, we must use a **Lagrange multiplier**.

In general, we might want to solve a constrained optimization problem of the form $\max_{\vec{x}} f(\vec{x})$ such that $g(\vec{x}) = c$. Here, $f(\vec{x})$ is called the **objective function** and $g(\vec{x})$ is called the **constraint**. We form the **Lagrangian**

$$\nabla f(\vec{x}) + \lambda \nabla g(\vec{x}) = 0$$

and then solve for both $\vec{x}$ and λ.

Now we have all the tools required to solve this problem. First, however, we will transform the objective function a bit to make it easier to work with, using the convenient fact that the logarithm is monotonic increasing, and thus does not affect the solution.

$$\begin{aligned} \max_\theta \prod_{k=1}^{K} P(X = k)^{c(k)} &= \max_\theta \log \left(\prod_{k=1}^{K} \theta_k^{c(k)} \right) \\ &= \max_\theta \sum_{k=1}^{K} \log(\theta_k^{c(k)}) \\ &= \max_\theta \sum_{k=1}^{K} c(k) \log(\theta_k) \end{aligned}$$

We get the gradient of this objective function by, for each θ_k, taking the partial derivative with respect to θ_k:

$$\frac{\partial}{\partial \theta_k} \left[\sum_{j=1}^{K} c(j) \log(\theta_j) \right] = \frac{c(k)}{\theta_k}$$

(To get this derivative, observe that all of the terms in the sum are constant with respect to θ_k except for the one term containing θ_k; taking the derivative of that term gives the result, and the other terms' derivatives are 0.)

Thus, we get that

$$\nabla f = \frac{\partial}{\partial \theta} \left[\sum_{j=1}^{K} c(j) log(\theta_j) \right] = \left(\frac{c(1)}{\theta_1}, \ldots, \frac{c(K)}{\theta_K} \right)$$

In a similar vein,

$$\nabla g = \frac{\partial}{\partial \theta} \left[\sum_{j=1}^{K} \theta_j \right] = (1, \ldots, 1)$$

Now we substitute these results into the Lagrangian $\nabla f + \lambda \nabla g = 0$. Solving this equation, we discover that for each k, $\frac{c(k)}{\theta_k} = -\lambda$, or $\theta_k = -\frac{c(k)}{\lambda}$. To solve for λ, we substitute this back into our constraint, and discover that $\sum_{k=1}^{K} \theta_k = -\frac{1}{\lambda} \sum_{k=1}^{K} c(k)$, and thus $-\lambda = \sum_{k=1}^{K} c(k)$. This is thus our normalization constant.

In retrospect, this formula seems completely obvious. The probability of outcome k is the fraction of times outcome k occurred in our data. The math accords perfectly with our intuitions; why would we ever want to do anything else? The problem is that this formula overfits our data, like the curve separating the male datapoints from the female datapoints at the beginning of class. For instance, suppose we never see outcome k in our data. This formula would have us set $\theta_k = 0$. But we probably don't want to assume that outcome k will never, ever happen. In the next lecture, we will look at how to avoid this issue.

5 Entropy

Entropy is:

$$\begin{aligned} H(X) &= \sum_x P(x) \log \frac{1}{P(x)} \\ &= \int P(x) \log \frac{1}{P(x)} \mathrm{d}x \end{aligned}$$

We can think of this as a measure of information content. An example of this idea of information content is seen in Huffman coding. High frequency letters have short encodings while rarer letters have longer encodings. This forms a binary tree where the letters are at the leaves and edges to the left are 0 bits and edges to the right are 1 bits. If the probablities for the letters are all equal then this tree is balanced.

In the case of entropy we notice that $\log \frac{1}{P(x)}$ is a non-integer, so it is like an expanded Huffman coding.

5.1 Bounds on Entropy for a Discrete Random Variable

If the variable is descrete $H(X)$ is maximized when the distribution is uniform since $P(x) = \frac{1}{K}$, we see:

$$H(X) = \sum_{i=1}^{K} \frac{1}{K} \log K = \log K$$

If K is 2^n then $H(X) = \log 2^n = n$. Part of Homework 2 will be to prove that entropy on a discrete random variable is maximized by a uniform distribution ($\max_\theta H(X)$ where $\sum_n \theta_n = 1$ using the Lagrange equation).

To minimize $H(X)$ we want $P(x_i) = 1$ for some i (with all other $P(x_j)$ being zero[1]) giving $H(X) = \sum_{1 \le j \le K, j \neq i} 0 \log \frac{1}{0} + 1 \log 1 = 0$. We see then that:

$$0 \le H(X) \le \log K$$

If we consider some distribution we can see that if we cut up the "shape" of the distribution and add in gaps that the gaps that are added do not contribute to $P(x) \log \frac{1}{P(x)}$.

[1] What about $0 \cdot \log \frac{1}{0}$? It is standard to define this as equal to zero (justified by the limit being zero).

5.2 Further Entropy Equations

$$
\begin{aligned}
H(X,Y) &= \sum P(x,y) \log \frac{1}{P(x,y)} \\
H(X|Y) &= \sum_{x,y} P(x|y)P(y) \log \frac{1}{P(x|y)} \\
&= \mathrm{E}_{XY}\left[\log \frac{1}{P(x|y)}\right] \\
&= \sum_{x,y} P(x,y) \log \frac{1}{P(x|y)}
\end{aligned}
$$

6 Mutual Information

Mutual information attempts to measure how correlated two variables are with each other:

$$
I(X;Y) = \sum_{x,y} P(x,y) \log \frac{P(x,y)}{P(x)P(y)}
$$

Consider communicating the values of two variables. The mutual information of these two variables is the difference beween the entropy of communicating these varibales individually and the entropy if we can send them together. For example if X and Y are the same then $H(X) + H(Y) = 2H(X)$ while $H(X,Y) = H(X)$ (since we know Y if we are given X). So $I(X;Y) = 2H(X) - H(X) = H(X)$.

6.1 Covariance

A number version of mutual information is covariance:

$$
\begin{aligned}
\mathrm{Covar}[X,Y] &= \sum_{x,y} P(x,y)(x - \bar{X})(y - \bar{Y}) \\
\mathrm{Covar}[X,X] &= \mathrm{Var}[X]
\end{aligned}
$$

Covariance indicates the high level trend, so if both X and Y are generally increasing, or both generally decreasing, then the covariance will be positive. If one is generally increasing, but the other is generally decreasing, then the covariance will be negative. Two variables can have a high amount of mutual information but no general related trend and the covariance will not indicate much (probably be around zero).

6.2 KL divergence

Kullback–Leibler (KL) divergence compares two distributions over some variable:

$$
\begin{aligned}
D(P \parallel Q) &= \sum_{x} P(x) \log \frac{P(x)}{Q(x)} \\
&= \mathrm{E}_P\left[\log \frac{1}{Q(x)} - \log \frac{1}{P(x)}\right] \\
&= \underbrace{H_P(Q)}_{\text{Cross Entropy}} - \underbrace{H(P)}_{\text{Entropy}}
\end{aligned}
$$

If we have the same distribution then the there is non divergence $D(P \parallel P) = 0$. In general the KL divergence is non-symetric $D(P \parallel Q) \neq D(Q \parallel P)$. If neither distribution is "special" the average $\frac{1}{2}[D(P \parallel Q) + D(Q \parallel P)]$ is sometimes used and is symetric. The units of KL divergence are log probability.

The cross entropy has an information interpretation quantifying how many bits are wasted by using the wrong code:

$$H_P(Q) = \sum_x \underbrace{P(x)}_{\text{Sending } P} \overbrace{\log \frac{1}{Q(x)}}^{\text{code for } Q}$$

6.3 Lower Bound for KL divergence

We will show that KL divergence is always greater or equal to zero using Jensen's inequality. First we need a definition of convex. A function f is convex if for all x_1, x_2 and θ where $0 \leq \theta \leq 1$, $f(\theta x_1 + (1-\theta)x_2) \leq \theta f(x_1) + (1-\theta)f(x_2)$. This is saying that any chord on the function is above the function itself on the same interval.

Some examples of convex include a straight line and $f(x) = x^2$. If the Hessian exists for a function then $\nabla^2 f \succeq 0$ (the Hessian is positive semidefinite) indicates that f is convex. This works for a line, but not something like $f(x) = |x|$.

Jensen's inequality states that if f is convex then $\mathrm{E}[f(X)] \geq f(E[X])$.

Proof.

$$\begin{aligned} D(P \parallel Q) &= \mathrm{E}_P \left[\log \frac{P(x)}{Q(x)} \right] \\ &= \mathrm{E}_P \left[-\log \frac{Q(x)}{P(x)} \right] \end{aligned}$$

To apply Jensen's inequality we will let $-\log$ be our function and $\frac{Q(x)}{P(x)}$ be our x (note that this ratio is a number so we can push the E_P inside).

$$\begin{aligned} \mathrm{E}_P \left[-\log \frac{Q(x)}{P(x)} \right] &\geq -\log \mathrm{E}_P \left[\frac{Q(x)}{P(x)} \right] \\ &= -\log \sum_x P(x) \frac{Q(x)}{P(x)} \\ &= -\log 1 = 0 \end{aligned}$$

□

Thinking of our information interpretation, we see that we always pay some cost for using the wrong code. Also note that $\log \frac{P(x)}{Q(x)}$ is sometimes positive and sometimes negative (P and Q both sum to one), yet $D(P \parallel Q) \geq 0$.

6.4 L_1 norm

The L_1 norm is defined as:

$$\|P - Q\|_1 = \sum_x |P(x) - Q(x)|$$

It can be thought of as "how much earth has to be moved" to match the distributions.

Because P and Q sum to one we quickly see that $0 \leq \|P - Q\|_1 \leq 2$. This property can be advantagous when bounds are needed.

7 The Gaussian Distribution

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

$$p(x;\mu,\Sigma) = \frac{1}{2\pi^{d/2}|\Sigma|^{1/2}}\exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

7.1 Maximum Likelihood Estimation

$$\begin{aligned}\mu =& \frac{1}{N}\sum_{n=1}^{N} x^{(n)}\\ \Sigma_{ij} =& \frac{1}{N}\sum_{n=1}^{N}(x_i^{(n)}-\mu_i)(x_j^{(n)}-\mu_j)\end{aligned}$$

7.2 Maximum Entropy for fixed Variance

$$\begin{aligned}\max_{p(x)} \; & -\int p(x)\log p(x)dx\\ \text{s.t.} \; & \int p(x)(x-\mu)^2dx = 1\\ & \int p(x)dx = 1\end{aligned}$$

Solving with langrange multipliers:

$$p(x) = \frac{1}{Z}\exp\left(\lambda(x-\mu)^2\right) \tag{1}$$

7.3 Marginalization

If

$$\begin{aligned}x &= \left[\begin{array}{cc} x_a x_b \end{array}\right]\\ x &\sim N(X;\mu\Sigma)\\ \mu &= \left[\begin{array}{c} \mu_a \\ \mu_b \end{array}\right]\\ \Sigma &= \left[\begin{array}{cc} \Sigma_{aa} & \Sigma_{ab} \\ \Sigma_{ba} & \Sigma_{bb} \end{array}\right]\end{aligned}$$

then the marginal for x_a is itself gaussian:

$$x_a \sim N(x_a;\mu_a,\Sigma_{aa})$$

7.4 Conditioning

With x_a, x_b as above, the conditional distribution $P(X_a \mid x_b)$ is gaussian:

$$\begin{aligned}\mu_{a|b} &= \mu_a + \Sigma_{ab}\Sigma_{bb}^{-1}(x_b-\mu b)\\ \Sigma_{a|b} &= \Sigma_{aa} - \Sigma_{ab}\Sigma_{bb}^{-1}\Sigma_{ba}\end{aligned}$$

8 Linear Regression

Let our prediction $\hat{y} = w^T x$.

$$\min_w \sum_n (\hat{y} - y_n)^2$$
$$\min_w \sum_n (w^T x^{(n)} - y_n)^2$$
$$\min_w \|w^T X - y\|^2$$

$$\begin{aligned} 0 &= \frac{\partial}{\partial w} \|w^T X - y\|^2 \\ &= 2X^T X w - 2X^T y \\ w &= (X^T X)^{-1} X^T y \end{aligned}$$

with regularization

$$w = (X^T X + \lambda I)^{-1} X^T y$$

9 Smoothing

If we use MLE to train a classifier, all of our probabilities are based on counts, any unseen combination of a single feature x and the class label y results in

$$\begin{aligned} P(x|y) &= \frac{c(x,y)}{c(y)} \\ &= 0. \end{aligned}$$

These zeros can ruin the entire classifier. For example, say there's one bill where all the Republicans we know about voted "no". Now, say we are trying to classify an unknown politician who followed the Republican line on every other bill, but voted "yes" on this bill. The classifier will say that there is zero probability of this person being a Republican, since it has never seen the combination (Republican, voted yes) for that bill. It gives that single feature way too much power. To get rid of that, we can use a technique called smoothing, and modify the probabilities a little :

$$\begin{aligned} P(x = k|y) &= \frac{c(x = k, y) + \alpha}{c(y) + K\alpha} \\ k &\in \{1, ..., K\} \end{aligned}$$

Basically we are taking a little bit of the probability mass from things with high probability and giving it to things with otherwise zero probability. (Republicans might veto this technique, since it's like redistribution of wealth!) Note that these probabilities must still sum to 1. This seems great - we've gotten rid of things with zero probability. But doesn't this contradict what we proved earlier? That is, last week we said that we can best infer the probability distribution by solving

$$\begin{aligned} &\operatorname*{argmax}_\theta \prod_{n=1}^{N} P_\theta(x_n) \\ &\text{s.t.} \sum_{k=1}^{K} \theta_k = 1 \end{aligned}$$

which results in the count-based distribution

$$\theta_k^* = \frac{c(k)}{N}.$$

How then can we mathematically justify our smoothed probabilities?

9.1 Prior Distributions

We can treat θ as a random variable itself with some probability distribution $P(\theta)$. Recall that θ is a vector of probabilities for each type of event k, so

$$\theta = [\theta_1, \theta_2, \ldots \theta_K]^T$$
$$\text{and } \sum_{k=1}^{K} \theta_k = 1$$

Suppose that we have a coin with two outcomes, heads or tails (K=2). We can picture the θ_1 and θ_2 which we could pick for the probability distribution of these two outcomes. A fair coin has $\theta_1 = 1/2$ and $\theta_2 = 1/2$. An weighted coin might have $\theta_1 = 2/3$ and $\theta_2 = 1/3$. Since we are treating θ as a random variable, its probability $P(\theta)$ is describing the probability that it takes on these values. $P(\theta)$ is called a prior, since it's what we believe about θ before we even have any observations. For example, we might tend to believe that the coin will be pretty fair, so we could have $P(\theta)$ be a normal curve with the peak where $\theta_1 = 1/2$ and $\theta_2 = 1/2$.

9.2 Dirichlet Prior

One useful prior distribution is the Dirichlet Prior :

$$\begin{aligned} P(\theta) &= \frac{\Gamma(\sum_{k=1}^{K} \alpha_k)}{\prod_{k=1}^{K} \Gamma(\alpha_k)} \prod_{k=1}^{K} \theta_k^{\alpha_k - 1} \\ &= \frac{1}{Z} \prod_{k=1}^{K} \theta_k^{\alpha_k - 1} \end{aligned}$$

This is also written as $P(\theta; \alpha)$, $P_\alpha(\theta)$, or $P(\theta|\alpha)$. α is a vector with the same size as θ, and it is known as a "hyperparameter". The choice of α determines the shape of θ's distribution, which you can see by varying it. If α is simply a vector of ones, we just get a uniform distribution; all θs are equally probable. In the case of two variables, we can have α_1=100, and α_2=50 and we see a sharp peak around $2/3$. The larger α_1, the more shaply peaked it gets around $\frac{\alpha_1}{\alpha_1 + \alpha_2}$.

At this point, we are tactfully ignoring that Γ in the Dirichlet distribution. What is that function, and what does it do?

9.3 Gamma Function

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

This function occurs often in difficult, nasty integrals. However, it has the nice property of being equivalent to the factorial function:

$$\Gamma(n) = (n-1)!$$

We can prove this using integration by parts:

$$\begin{aligned}\Gamma(x) &= \int_0^\infty e^{-t}t^{x-1}dt \\ &= \left[-t^{x-1}e^{-t}\right]_0^\infty + \int_0^\infty e^{-t}(x-1)t^{x-2}dt \\ &= 0 + (x-1)\int_0^\infty e^{-t}t^{x-2}dt \\ &= (x-1)\Gamma(x-1)\end{aligned}$$

Further noting that $\Gamma(1) = 1$, we can conclude that $\Gamma(n) = (n-1)!$. This function is used in the normalization constant of our Dirichlet prior in order to guarantee that:

$$\int_{\sum_k \theta_k = 1} P(\theta)d\theta = 1.$$

9.4 Justifying the Dirichlet Prior

How can we use this prior to compute probabilities?

$$\begin{aligned}P(x = k|\theta) &= \theta_k \\ P(x = k) &= \int_{\sum_k \theta_k = 1} P(x|\theta)P(\theta)d\theta \\ &= \int \theta_k \frac{\Gamma(\sum_{k=1}^K \alpha_k)}{\prod_{k=1}^K \Gamma(\alpha_k)} \prod_{k=1}^K \theta_k^{\alpha_k - 1} d\theta \\ &= \frac{\Gamma(\sum_{k=1}^K \alpha_k)}{\prod_{k=1}^K \Gamma(\alpha_k)} \int \prod_{k'=1}^K \theta_{k'}^{\alpha_{k'} - 1 + I(k'=k)} d\theta \\ &= \frac{\Gamma(\sum_{k=1}^K \alpha_k)}{\prod_{k=1}^K \Gamma(\alpha_k)} \frac{\prod_{k'} \Gamma(\alpha_{k'} + I(k'=k))}{\Gamma(\sum_{k'} \alpha_{k'} + I(k'=k))} \\ &= \frac{\Gamma(\sum_{k=1}^K \alpha_k)}{\Gamma(\sum \alpha_k + 1)} \frac{\Gamma(\alpha_k + 1)}{\Gamma(\alpha_{k'})}\end{aligned}$$

Now we use $\Gamma(x) = (x-1)\Gamma(x-1)$:

$$= \frac{\alpha_k}{\sum_{k'} \alpha_{k'}}$$

Most of the time, all of the α_k's are set to the same number. So, we just showed that

$$P(x) = = \frac{\alpha_k}{\sum_{k'} \alpha_{k'}}$$

But what about

$$\begin{aligned}
P(X_{N+1}|X_1^N) &= \int P(X_{N+1}, \theta|X_1^N)d\theta \\
&= \int P(X_{N+1}|\theta, X_1^N)P(\theta|X_1^N)d\theta \\
&= \int \theta_k \frac{P(X_1^N|\theta)P(\theta)}{P(X_1^N)}d\theta \\
&= \frac{1}{Z}\int \theta_k \prod_n \theta_{X_n} \frac{1}{Z'}\prod_k \theta_k^{\alpha_k - 1}d\theta \\
&\dots \\
&= \frac{c(k) + \alpha_k}{N + \sum_k \alpha_k}
\end{aligned}$$

10 Comparison - Bayesian vs. MLE vs. MAP

10.1 Bayesian

The quantity we just computed is known as the Bayesian:

$$P(X_{N+1}|X_1^N) = \frac{c(k) + \alpha_k}{N + \sum_k \alpha_k}$$

We can compare it to the MLE that we did before:

$$P(x_{N+1}) \text{ follows } \theta^*$$
$$\theta^* = \underset{\theta}{\text{argmax}}\, P_\theta(X_1^N)$$

And a third alternative is the MAP, or Maximum A Posteriori:

$$P(x_{N+1}) \text{ follows } \theta^*$$
$$\theta^* = \underset{\theta}{\text{argmax}}\, P(\theta)P(X_1^N|\theta)$$

This is simpler since it does not require an integral. Using the same Lagrange Multipliers technique as we did before:

$$\text{argmax}\frac{1}{Z}\prod_k \theta_k^{\alpha_k - 1}\prod_k \theta_k^c(k)$$
$$\text{s.t.} \sum_k \theta_k = 1$$

Then we get the result:

$$\theta_k^* = \frac{c(k) + \alpha_k - 1}{N + (\sum_{k'} \alpha_k') - K}$$

11 Perceptrons

A Perceptron is a linear classifier that determines a decision boundary through successive changes to the weight vector $\mathbf{w}^T$ of a linear classifier. A linear classifier computes a linear function of the input data point

$\mathbf{x}$ and then converts it a class label of either -1 or 1 with the sign function:

$$t = \text{sign}(\mathbf{w}^T\mathbf{x} + b)$$

We define our classification function sign as

$$\text{sign}(x) = \begin{cases} -1 & : x < 0 \\ 0 & : x = 0 \\ 1 & : x > 0 \end{cases}$$

We can remove $\mathbf{b}$ from the equation by adding it as an element of $\mathbf{w}$ and adding a 1 to $\mathbf{x}$ in the same spot.

$$\mathbf{w}' = \begin{bmatrix} w_1 \\ \vdots \\ w_N \\ b \end{bmatrix} \quad \mathbf{x}' = \begin{bmatrix} x_1 \\ \vdots \\ x_N \\ 1 \end{bmatrix}$$

$$\mathbf{w}'^T\mathbf{x}' = \mathbf{w}^T\mathbf{x} + b$$

The next question is, how do we pick $\mathbf{w}$? We have $\mathbf{x}$ as the vector of data points, with $\mathbf{x}^n$ as the nth data point. We have t^n as the classifier output (1 or -1) for the nth data point.

$$t^n = \text{sign}(\mathbf{w}^T x)$$

y^n is the true label for the nth data point.

Our general goal is to maximize the number of correctly classified points:

$$\underset{\mathbf{w}}{\text{argmax}} \sum_n I(y^n = t^n)$$

However, in general, maximizing the number of correctly classified points is NP-complete. The perceptron algorithm as well as SVMs and logistic regression can be viewed as ways of approximately solving this problem.

The Perceptron algorithm takes the simple approach of updating the weight vector whenever it misclassifies a data point:

repeat
  for $n = 1...N$ **do**
    if $t^n \neq y^n$ **then**
      $\mathbf{w} \leftarrow \mathbf{w} + y^n\mathbf{x}^n$
until $\forall n\ t^n = y^n$ or maxiters

While this algorithm will completely separate linearly separable data, it may not be the best separation (it may not accurately represent the separating axis of the data).

11.1 Proof of Convergence

As mentioned before, perceptron algorithm will converge eventually if the data is linearly separable, but why? Let's first formally write down the problem,

Definition: $(x^n, y_n), n \in 1, 2, \ldots, N$ is linearly separable, iff $\exists(u, \delta), \|u\| = 1, \delta > 0, s.t.\ \forall n, y_n u^T x^{(n)} \geq \delta$. The vector u is called an *oracle* vector that separates everything correctly.

Theorem: If $x^{(n)}$ is bounded by R, i.e., $\forall n, \|x^{(n)}\| \leq R$, then the perceptron algorithm makes at most $\frac{R^2}{\delta^2}$ updates. (for a vector $\|v\|$ denotes the Euclidean norm of v, i.e., $\|v\| = \sqrt{\sum_i v_i^2}$)

Proof: As we keep updating the weights w in the algorithm, a sequence of $w^{(k)}$ are generated.

Let $w^{(1)} = \mathbf{0}$.

Each time we encountered a misclassification, we use it to update the weights w. Suppose we use data point (x, y) to update $w^{(k)}$, which means Equation (2) holds,

$$w^{(k+1)} = w^{(k)} + yx \tag{2}$$

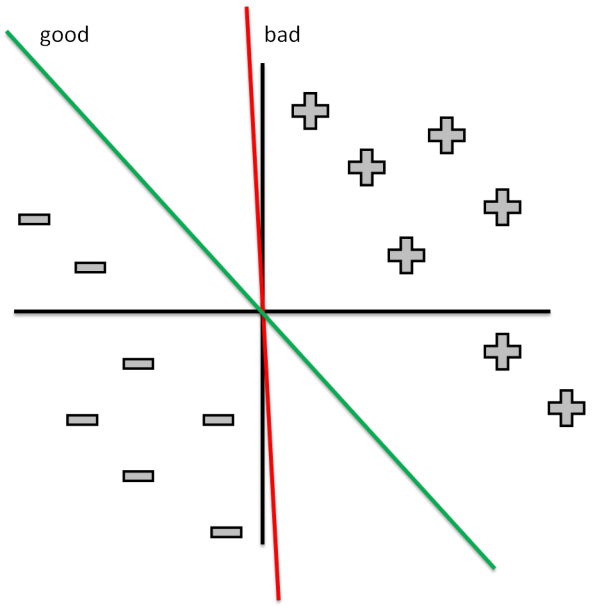


Figure 1: Linear Separability

Bear in mind that since we are using (x, y) to update $w^{(k)}$, $w^{(k)}$ misclassified (x, y), which means Equation (3) holds,

$$y(w^{(k)})^T x < 0 \tag{3}$$

Now we can use Equation (2), (3) and $w^{(1)} = \mathbf{0}$ to prove that $k \leq \frac{R^2}{\delta^2}$. Hence, the convergence holds. From Equation (4), we can get a upper bound of $\|w^{(k+1)}\|$, and from Equation (5), we can get its lower bound.

$$\begin{array}{rrcll}
 & w^{(k+1)} & = & w^{(k)} + yx & \\
\Rightarrow & u^T w^{(k+1)} & = & u^T w^{(k)} + y u^T x & \text{Multiply oracle } u \text{ to both sides} \\
\Rightarrow & u^T w^{(k+1)} & \geq & u^T w^{(k)} + \delta & \text{Definition of } (u, \delta) \\
\Rightarrow & u^T w^{(k+1)} & \geq & k\delta & \text{Induction and } w^{(1)} = \mathbf{0} \\
\Rightarrow & \|u^T w^{(k+1)}\|^2 & \geq & k^2\delta^2 & \text{Both sides positive} \\
\Rightarrow & \|u\|^2 \|w^{(k+1)}\|^2 & \geq & k^2\delta^2 & \text{Cauchy Schwarz: } \|\mathbf{a}\| \times \|\mathbf{b}\| \geq \left|\mathbf{a}^T \mathbf{b}\right| \\
\Rightarrow & \|w^{(k+1)}\|^2 & \geq & k^2\delta^2 & \|u\| = 1
\end{array} \tag{4}$$

$$\begin{array}{rrcll}
 & w^{(k+1)} & = & w^{(k)} + yx & \\
\Rightarrow & \|w^{(k+1)}\|^2 & = & \|w^{(k)} + yx\|^2 & \text{Apply Euclidean norm to both sides} \\
\Rightarrow & \|w^{(k+1)}\|^2 & = & \|w^{(k)}\|^2 + \|y\|^2\|x\|^2 + 2yx^T w^{(k)} & \text{Expansion} \\
\Rightarrow & \|w^{(k+1)}\|^2 & \leq & \|w^{(k)}\|^2 + \|y\|^2\|x\|^2 & \text{Equation (3)} \\
\Rightarrow & \|w^{(k+1)}\|^2 & \leq & \|w^{(k)}\|^2 + R^2 & \|y\| = 1, \|x\| \leq R \\
\Rightarrow & \|w^{(k+1)}\|^2 & \leq & kR^2 & \text{Induction and } w^{(1)} = \mathbf{0}
\end{array} \tag{5}$$

Combining results of Equation (4) and (5), we get $k^2\delta^2 \leq \|w^{(k+1)}\|^2 \leq kR^2$. Thus, $k^2\delta^2 \leq kR^2$ and $k \leq \frac{R^2}{\delta^2}$.

Since the number of updates is bounded by $\frac{R^2}{\delta^2}$, the perceptron algorithm will eventually converge to somewhere no updates are needed. ■

11.2 Perceptron in Stochastic Gradient Descent perspective

The perceptron algorithm can be analyzed in a more general framework, i.e., stochastic gradient descent for a convex optimization problem.

The ultimate goal for perceptron algorithm is to find w, such that $\forall(x^{(k)}, y_k), y_k w^T x^{(k)} \geq 0$. Therefore, a natural penalty for misclassification is $[-y_k w^T x^{(k)}]_+$. For scalar number S, $[S]_+$ is defined in Equation (6), which is sometimes called a *hinge* function.

$$[S]_+ = \begin{cases} S & \text{if } S \geq 0 \\ 0, & \text{if } S < 0 \end{cases} \tag{6}$$

Thus, the optimization problem can be written down as Equation (7).

$$\underset{w}{\operatorname{argmin}} f(w) \triangleq \frac{1}{N} \sum_k f_k(w) \triangleq \frac{1}{N} \sum_k \left[-y_k w^T x^{(k)} \right]_+ \tag{7}$$

One way to optimize the convex function in Equation (7) is called *Gradient Descent*. Essentially, Gradient Descent keeps updating the weights w, $w \leftarrow w - \alpha \nabla_w f(w)$, in which α is called *learning rate*. The gradient $\nabla_w f(w)$ can be carried out by $\nabla_w f(w) = \frac{1}{N} \sum_k \nabla_w f_k(w)$, and $\nabla_w f_k(w)$ is computed by Equation (8).

$$\nabla_w f_k(w) = \begin{cases} -y_k x^{(k)} & \text{if } y_k w^T x^{(k)} < 0 \\ 0, & \text{if } y_k w^T x^{(k)} \geq 0 \end{cases} \tag{8}$$

Because the gradient $\nabla_w f(w)$ is a summation of local gradients $\nabla_w f_k(w)$, we can also do Stochastic Gradient Descent by using one data instance a time.

1. Randomly pick a data instance, $(x^{(k)}, y_k)$
2. Compute local gradient on it, $\nabla_w f_k(w)$ as Equation (8).
3. Update weights using the local gradient, $w \leftarrow w - \alpha \nabla_w f_k(w)$. This is exactly the update in perceptron algorithm.

12 Multi Layer Perceptron

In a two layer neural network, the input, hidden, and output variables are represented by nodes, and the weight parameters are represented by links between the nodes. The arrow of the links indicate the direction of information flow through the network during forward propagation. The overall network function takes the form:

$$\begin{aligned} z_i^{(0)} &= x_i && \text{input layer} \\ a_i^{(\ell)} &= \sum_j w_{ij}^{(\ell)} z_j^{(\ell-1)} && \ell \in 1 \dots L \\ z_i^{(\ell)} &= g(a_i^{(\ell)}) && \ell \in 1 \dots L \\ \hat{y}_i &= z_i^{(L)} && \text{output layer} \end{aligned}$$

where $z_i^{(\ell)}$ is the output of node i in layer ℓ, $w_{ij}^{(\ell)}$ are the weights, x_i are the input variables, $\hat{y}$ is the network's output, and the g is an activation function. Non-linear functions are usually chosen for activation functions such as $\tanh$ and sigmoid functions.

12.1 Training A Network: Error Backpropagation

Given a training set of input vector $\mathbf{x}$ and its target vector $\mathbf{y}$ for $n = 1 \dots N$, we want to minimize the error function $E_W(\mathbf{y}, \hat{\mathbf{y}})$ that quantifies the difference between the network's predication $\hat{\mathbf{y}}$ and the true label $\mathbf{y}$. One common choice for E is squared error:

$$E_W(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2} \|\hat{\mathbf{y}} - \mathbf{y}\|^2 \tag{9}$$

If we treat the ouput values as probabilities, log-likelihood, a.k.a. cross-entropy, is an appropriate choice of E:

$$E_W(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i \hat{y}_i \log y_i \tag{10}$$

We define a table of partial results for dynamic programming:

$$\delta_i^{(\ell)} = \frac{\partial E}{\partial a_i^{(\ell)}} \tag{11}$$

which captures the contribution of node i at layer ℓ to the error.

Using the chain rule of calculus to derive weight updates at the top layer:

$$\delta_i^{(L)} = \frac{\partial E}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial a_i^{(L)}} \tag{12}$$

$$= \frac{\partial E}{\partial \hat{y}_i} g'(a_i^{(L)}) \tag{13}$$

$$\frac{\partial E}{\partial w_{ij}^{(L)}} = \frac{\partial E}{\partial a_i^{(L)}}\frac{\partial a_i^{(L)}}{\partial w_{ij}^{(L)}} \tag{14}$$

$$= \delta_i^{(L)} z_j^{(L-1)} \tag{15}$$

Weight updates are derived recursively for each lower layer:

$$\delta_i^{(\ell)} = \frac{\partial E}{\partial a_i^{(\ell)}} \tag{16}$$

$$= \sum_k \frac{\partial E}{\partial a_k^{(\ell+1)}}\frac{\partial a_k^{(\ell+1)}}{\partial a_i^{(\ell)}} \tag{17}$$

$$= \sum_k \delta_k^{(\ell+1)} \frac{\partial a_k^{(\ell+1)}}{\partial z_i^{(\ell)}}\frac{\partial z_i^{(\ell)}}{\partial a_i^{(\ell)}} \tag{18}$$

$$= \sum_k \delta_k^{(\ell+1)} w_{ki}^{(\ell+1)} g'(a_i^{(\ell)}) \tag{19}$$

$$\frac{\partial E}{\partial w_{ij}^{(\ell)}} = \frac{\partial E}{\partial a_i^{(\ell)}}\frac{\partial a_i^{(\ell)}}{\partial w_{ij}^{(L)}} \tag{20}$$

$$= \delta_i^{(\ell)} z_j^{(l-1)} \tag{21}$$

Putting eq. 13, 15, 19, and 21 into vector notation, we get the backprop algorithm.

procedure BACKPROGPAGATION
  while not converged **do**
    for data point x, y **do**
      $\delta^{(L)} \leftarrow \frac{\partial E}{\partial \hat{y}} \odot g'(a^{(L)})$ ▷ From (13)
      $\frac{\partial E}{\partial W^{(L)}} \leftarrow \delta^{(L)} {z^{(L-1)}}^T$ ▷ From (15)
      for $\ell \leftarrow L-1 \ldots 1$ **do**
        $\delta^{(\ell)} \leftarrow {W^{(\ell+1)}}^T \delta^{(\ell+1)} \odot g'(a^{(\ell)})$ ▷ From (19)
        $\frac{\partial E}{\partial W^{(\ell)}} \leftarrow \delta^{(\ell)} {z^{(l-1)}}^T$ ▷ From (21)
      $W \leftarrow W - \eta \frac{\partial E}{\partial W}$ ▷ update all weights with SGD

13 Support Vector Machines

The Support Vector Machine (SVM) is one of the most widely used classification methods. The SVM is different from other classifiers that we have covered so far. The SVM cares only about the data points near the class boundary and finds a hyperplane that maximizes the margin between the classes.

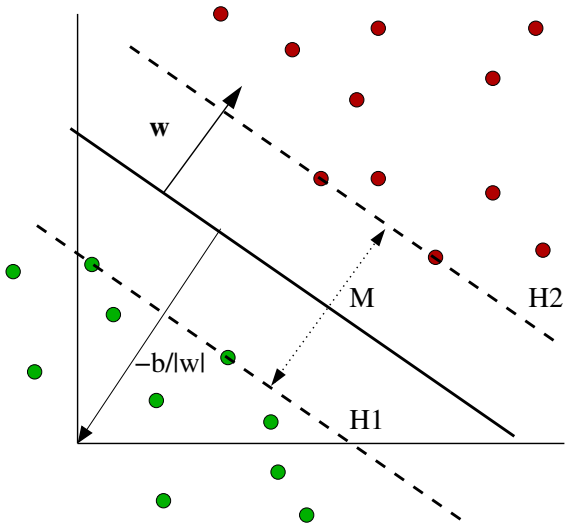


Figure 2: The figure shows a linear SVM classifier for two linearly separable classes. The hyperplane $\mathbf{w}^T x + b$ is the solid line between H_1 and H_2, and the the margin is M.

13.1 Training Linear SVMs

Let the input be a set of N training vectors $\{\mathbf{x}^{(n)}\}_{n=1}^N$ and corresponding class labels $\{y_n\}_{n=1}^N$, where $\mathbf{x}^{(n)} \in \mathbb{R}^D$ and $y_n \in \{-1, 1\}$. Initially we assume that the two classes are linearly separable. The hyperplane separating the two classes can be represented as:

$$\mathbf{w}^T \mathbf{x} + b = 0,$$

such that:

$$\begin{aligned} \mathbf{w}^T \mathbf{x}^{(n)} + b \geq 1 \quad &\text{for} \quad y_n = +1, \\ \mathbf{w}^T \mathbf{x}^{(n)} + b \leq -1 \quad &\text{for} \quad y_n = -1. \end{aligned}$$

Let H_1 and H_2 be the two hyperplanes (Figure 2) separating the classes such that there is no other data point between them. Our goal is to maximize the margin M between the two classes. The objective function:

$$\begin{aligned} \max_{\mathbf{w},b,M} \quad & M \\ \text{s.t.} \quad & y_n(\mathbf{w}^T \mathbf{x}^{(n)} + b) \geq M, \\ & \mathbf{w}^T \mathbf{w} = 1. \end{aligned}$$

The margin M is equal to $\frac{2}{\|\mathbf{w}\|}$. We can rewrite the objective function as:

$$\begin{aligned} \min_{\mathbf{w}} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} \\ \text{s.t.} \quad & y_n(\mathbf{w}^T \mathbf{x}^{(n)} + b) \geq 1 \end{aligned}$$

Now, let's consider the case when the two classes are not linearly separable. We introduce slack variables $\{\xi_n\}_{n=1}^N$ and allow few points to be on the wrong side of the hyperplane at some cost. The modified objective function:

$$\begin{aligned} \min_{\mathbf{w},b,\xi} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{n=1}^{N} \xi_n \\ \text{s.t.} \quad & y_n(\mathbf{w}^T \mathbf{x}^{(n)} + b) + \xi_n \geq 1, \\ & \xi_n \geq 0, \ \ \forall n. \end{aligned} \tag{22}$$

The parameter C can be tuned using development set. This is the primal optimization problem for SVM.

13.2 SGD for SVM

At the solution of problem 22, each ξ_n is determined by one of the two constraints, so

$$\xi_n = \max\left\{0, 1 - y_n(\mathbf{w}^T\mathbf{x}^{(n)} + b)\right\}$$

Folding this equation into the objective function, we have an unconstrained minimization problem:

$$\min_{\mathbf{w},b} \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{n=1}^{N} \max\left\{0, 1 - y_n(\mathbf{w}^T\mathbf{x}^{(n)} + b)\right\} \tag{23}$$

Moving $\mathbf{w}^T\mathbf{w}$ into the sum over data points, we have:

$$\min_{\mathbf{w},b} \sum_{n=1}^{N} \frac{1}{2N}\mathbf{w}^T\mathbf{w} + C\max\left\{0, 1 - y_n(\mathbf{w}^T\mathbf{x}^{(n)} + b)\right\} \tag{24}$$

Let f_n denote the term of the above sum for data point n. The gradient is:

$$\frac{\partial f_n}{\partial \mathbf{w}} = \begin{cases} \frac{1}{N}\mathbf{w} - Cy_n\mathbf{x}^{(n)} & \text{if } 1 - y_n(\mathbf{w}^T\mathbf{x}^{(n)} + b) > 0 \\ \frac{1}{N}\mathbf{w} & \text{otherwise} \end{cases} \tag{25}$$

$$\frac{\partial f_n}{\partial b} = \begin{cases} -Cy_n & \text{if } 1 - y_n(\mathbf{w}^T\mathbf{x}^{(n)} + b) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{26}$$

13.3 Dual Form

The Lagrangian for the primal problem:

$$L(\mathbf{w}, b, \xi, \alpha, \mu) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{n=1}^{N}\xi_n - \sum_n \alpha_n\left[y_n(\mathbf{w}^T\mathbf{x}^{(n)} + b)\right] - \sum_n \alpha_n\xi_n + \sum_n \alpha_n - \sum_n \mu_n\xi_n,$$

where α_n and μ_n, $1 \le n \le N$ are Lagrange multipliers.

Differentiating the Lagrangian with respect to the variables:

$$\frac{\partial}{\partial \mathbf{w}} L(\mathbf{w}, b, \xi, \alpha, \mu) = \mathbf{w} - \sum_n \alpha_n y_n \mathbf{x}^{(n)} = 0$$

$$\frac{\partial}{\partial b} L(\mathbf{w}, b, \xi, \alpha, \mu) = -\sum_n \alpha_n y_n = 0$$

$$\frac{\partial}{\partial \xi_n} L(\mathbf{w}, b, \xi, \alpha, \mu) = C - \alpha_n - \mu_n = 0$$

Solving these equations, we get:

$$\mathbf{w} = \sum_n \alpha_n y_n \mathbf{x}^{(n)} \tag{27}$$

$$\sum_n \alpha_n y_n = 0$$

$$\alpha_n = C - \mu_n \tag{28}$$

We now plug-in these values to get the dual function and cancelling out some terms:

$$\begin{aligned} g(\alpha,\mu) &= \frac{1}{2}\sum_n\sum_m \alpha_n\alpha_m y_n y_m {\mathbf{x}^{(n)}}^T\mathbf{x}^{(m)} - \sum_n\sum_n \alpha_n\alpha_m y_n y_m {\mathbf{x}^{(n)}}^T\mathbf{x}^{(m)} + \sum_n \alpha_n \\ &= \sum_n \alpha_n - \frac{1}{2}\sum_n\sum_m \alpha_n\alpha_m y_n y_m {\mathbf{x}^{(n)}}^T\mathbf{x}^{(m)} \end{aligned} \tag{29}$$

Using the equation (28) and (29) and the KKT conditions, we obtain the dual optimization problem:

$$\begin{aligned} \max_{\alpha}\ & \sum_n \alpha_n - \frac{1}{2}\sum_n\sum_m \alpha_n\alpha_m y_n y_m {\mathbf{x}^{(n)}}^T\mathbf{x}^{(m)} \\ \text{s.t. }\ & 0 \le \alpha_n \le C. \end{aligned}$$

The dual optimization problem is concave and easy to solve. The dual variables (α_n) lie within a box with side C. We usually vary two values α_i and α_j at a time and numerically optimize the dual function. Finally, we plug in the values of the α_n^*'s to the equations (27) to obtain the primal solution $\mathbf{w}^*$.

13.4 Convex Optimization Review

Suppose we are given an optimization problem:

$$\begin{aligned} \min_{x}\ & f_0(x) \\ \text{s.t. }\ & f_i(x) \le 0, \ \text{ for } i \in 1,2,\ldots,K, \end{aligned}$$

where f_0 and f_i ($i \in \{1,2,\ldots,K\}$) are convex functions. We call this optimization problem the 'primal' problem.

The Lagrangian is:

$$L(x,\lambda) = f_0(x) + \sum_{i=1}^{K}\lambda_i f_i(x)$$

The Lagrange dual function:

$$g(\lambda) = \min_x L(x,\lambda)$$

The dual function $g(\lambda)$ is concave and hence easy to solve. We can obtain the minima of a convex primal optimization problem by maximizing the dual function $g(\lambda)$. The dual optimization problem is:

$$\begin{aligned} \max_{\lambda}\ & g(\lambda) \\ \text{s.t. }\ & \lambda_i \ge 0, \ \text{ for } i \in 1,2,\ldots,K. \end{aligned}$$

13.4.1 Karush-Kuhn-Tucker (KKT) Conditions

The Karush-Kuhn-Tucker (KKT) conditions are the conditions for optimality in primal and dual functions. If f_0 and f_i's are convex, differentiable, and the feasible set has some interior points (satisfies Slater condition), the x^* and λ_i^*'s are the optimal solutions of the primal and dual problems if and only if they satisfy the following conditions:

$$
\begin{aligned}
f_i(x^*) &\leq 0 \\
\lambda_i^* &\geq 0, \ \forall i \in 1, \ldots, K \\
\frac{\partial}{\partial x} L(x^*, \lambda_1^*, \ldots, \lambda_K^*) &= 0 \\
\lambda_i^* f_i(x^*) &= 0
\end{aligned}
$$

13.4.2 Constrained Optimization

We now proceed to give some intuition about why the relationship between the primal and dual holds, as well as the why KKT conditions hold at the solution. Consider a constrained optimization problem:

$$
\begin{aligned}
&\min_x f(x) \\
&\text{s.t. } g(x) = 0
\end{aligned}
$$

At the solution, the gradient of the objective function f must be perpendicular to the constraint surface (feasible set) defined by $g(x) = 0$, so there exists a scalar Lagrange multiplier λ such that

$$
\frac{\partial f}{\partial x} + \lambda \frac{\partial g}{\partial x} = 0
$$

at the solution.

13.4.3 Inequalities

Consider an optimization problem with constraints specified as inequalities:

$$
\begin{aligned}
&\min_x f(x) \\
&\text{s.t. } g(x) \leq 0
\end{aligned}
\tag{30}
$$

If, at the solution, $g(x) = 0$, then as before there exists a λ such that

$$
\frac{\partial f}{\partial x} + \lambda \frac{\partial g}{\partial x} = 0 \tag{31}
$$

and furthermore $\lambda > 0$, otherwise we would be able to decrease $f(x)$ by moving in the direction $-\frac{\partial f}{\partial x}$ without leaving the feasible set defined by $g(x) \leq 0$.

If, on the other hand, at the solution $g(x) < 0$, then we must be at a maximum of $f(x)$, so $\frac{\partial f}{\partial x} = 0$ and eq. 31 holds with $\lambda = 0$. In either case, the following system of equations (known as the KKT conditions) holds:

$$
\begin{aligned}
\lambda g(x) &= 0 \\
\lambda &\geq 0 \\
g(x) &\leq 0 \\
\frac{\partial f}{\partial x} + \lambda \frac{\partial g}{\partial x} &= 0
\end{aligned}
$$

13.4.4 Convex Optimization

Suppose now that $f(x)$ is convex, $g(x)$ is also convex, and both are continuously differentiable. Define

$$
L(x, \lambda) = f(x) + \lambda g(x)
$$

and Equation 31 is equivalent to

$$\frac{\partial L}{\partial x} = 0$$

For any fixed $\lambda \geq 0$, L is convex in x, and has a unique minimum. For any fixed x, L is linear in λ.

Define the **dual function**

$$h(\lambda) = \min_x L(x, \lambda)$$

The minimum of a set of linear functions is concave, and has a maximum corresponding to the linear function with derivative of 0. Thus $h(\lambda)$ also has a unique maximum over $\lambda \geq 0$. Either the maximum of h occurs at $\lambda = 0$, in which case

$$h(0) = \min_x L(x, 0) = \min_x f(x)$$

and we are at the global minimum of f, or the maximum of h occurs at

$$\frac{\partial L}{\partial \lambda} = g(x) = 0$$

and we are on the boundary of the feasible set. Because $\lambda > 0$, we cannot decrease f by moving into the interior of the feasible set, and therefore this is the solution to the original problem (30). In either case, the solution to **dual problem**

$$\begin{aligned} &\max_\lambda h(\lambda) \\ &\text{s.t. } \lambda \geq 0 \end{aligned}$$

corresponds to the solution of the original **primal** problem. Generalizing to primal problmes with more than one constraint, the dual problem has one dual variable λ_i for each constraint in the primal problem, and has simple non-negativity constraints on each dual variable. The dual problem is often easier to solve numerically due to its simple constraints. Thus one approach to solving convex optimization problems us to find the dual problem by using calculus to solve $\frac{\partial L}{\partial x} = 0$, and then solving the dual problem numerically.

Substituting the definition of the dual function into the dual problem yields the **primal-dual problem**:

$$\begin{aligned} &\max_\lambda \min_x L(x, \lambda) \\ &\text{s.t. } \lambda \geq 0 \end{aligned}$$

Another approach to solving convex optimization problems is to maintain both primal and dual variables, and to solve the primal-dual problem numerically.

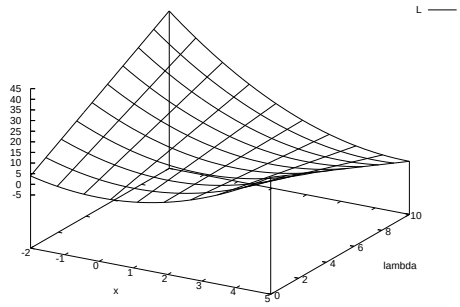
We have already seen that the KKT conditions must hold at the solution to the primal-dual problem. If the objective function and constraint functions are convex and differentiable, and if the feasible set has an interior (Slater's condition), than any solution to the KKT conditions is also a solution to the primal-dual problem.

13.4.5 An Example

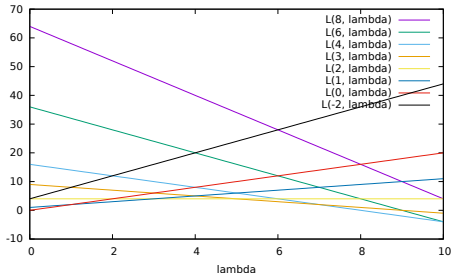
Minimize x^2 subject to $x \geq 2$.

$$\begin{aligned} L(x, \lambda) &= f(x) + \lambda g(x) \\ &= x^2 + \lambda(2 - x) \end{aligned}$$

The Lagrangian function L has a saddle shape:



Projecting onto the λ dimension, we see the concave function h formed from the minimum of linear functions $L(c, \lambda)$



To find h, set

$$\frac{\partial L}{\partial x} = 0$$
$$2x - \lambda = 0$$

and solve for x: $x = \lambda/2$. Substituting $x = \lambda/2$ into L gives $h(\lambda) = -\frac{1}{4}\lambda^2 - 2\lambda$. Setting $\frac{\partial h}{\partial \lambda} = 0$ yields $\lambda = 4$, which we see is the maximum of the concave shape in the figure. Substituting back into the original problem yields $x = 2$, a solution on the boundary of the constraint surface.

14 Kernel Functions

14.1 Review Support Vector Machines

Goal: To solve equation:

$$\min_w \left(\frac{1}{2}\|w\|^2 + C\sum_n \xi_n \right)$$
$$\text{s.t. } y^n(w^T x^n + b) + \xi_n \geq 1$$
$$\xi_n \geq 0$$

where

$$x^n = [x_1, x_2, ..., x_K]^T, n \in 1, ..., N$$

This is a K-dimensional problem, which means the more features the data has, the more complicated to solve this problem.

In the meantime, this equation is equal to

$$\begin{aligned} \max_{\alpha} \quad & \left(-\frac{1}{2} \sum_{n} \sum_{m} \alpha_n \alpha_m y^n y^m x^{nT} x^m + \sum_{n} \alpha_n \right) \\ \text{s.t. } & \alpha_n \geq 0 \\ & \alpha_n \leq C \end{aligned}$$

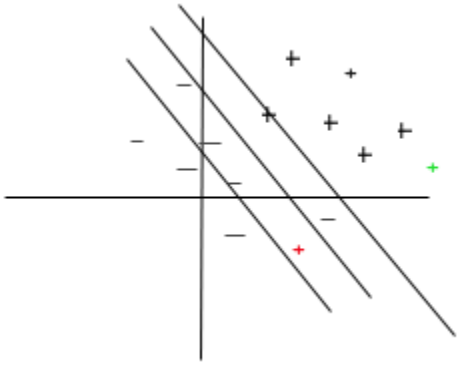
This is a N-dimensional problem, which means the more data points we include in the training set, the more time it takes to find the optimized classifier.

To train the classifier is actually to solve this problem inside the box of alpha.

According to KKT,

$$\begin{aligned} \lambda_i f_i(x) &= 0 \\ \lambda_i &\geq 0 \\ f_i(x) &\leq 0 \end{aligned}$$

As shown in the figure below,



$$w = \sum_{n} \alpha_n y^n x_n$$

Points on the right side but not on the margin contribute nothing because alpha equals to 0. (The green point)

For points on the wrong side (the red point), alpha equals to C, and

$$\xi_n > 0$$

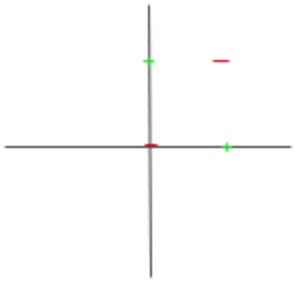
so they along with points on the margin contribute to the vector, but no point is allowed to contribute more than C.

SVM can train classifier better than naive bayes in the most of time, but since its still binary classification it is not able to deal with situation like this one below:

14.2 Kernel Function

Now when we look back, the classification formula is

$$Sign\left(w^T x\right) = Sign\left(\left(\sum_{n} \alpha_n y^n x_n\right)^T x\right) = Sign\left(\sum_{n} \alpha_n y^n \left(x^{nT} x\right)\right)$$



We can introduce Kernel Function K now, the simplest one is:

$$x^{nT}x = K\left(x^n, x\right)$$

Now the problem is transformed into:

$$\max_{\alpha}\left(-\frac{1}{2}\sum_{n}\sum_{m}\alpha_n\alpha_m y^n y^m K\left(x^n, x^m\right) + \sum_{n}\alpha_n\right)$$

where

$$K\left(x, y\right) = \phi\left(x\right)^T \phi\left(y\right)$$

for some ϕ.

The most commonly seen Kernel Functions are:

$$K\left(x, y\right) = x^T y$$

$$K\left(x, y\right) = \left(x^T y\right)^m$$

$$K\left(x, y\right) = e^{-c\|x-y\|^2}$$

Generally, Kernel function is a measure of how x and y are similar, then they are the same, it has the peak output.

14.3 Proof that ϕ exists

For a two dimensional

$$x = \left[x_1, x_2\right]^T y = \left[y_1, y_2\right]^T$$
$$K\left(x, y\right) = \left(x_1 y_1 + x_2 y_2\right)^m$$

Let m = 2, then

$$K\left(x, y\right) = \left(x_1 y_1\right)^2 + \left(x_2 y_2\right)^2 + 2\left(x_1 y_1 x_2 y_2\right) = \phi\left(x\right)^T \phi\left(y\right)$$

Thus, we can conclude that

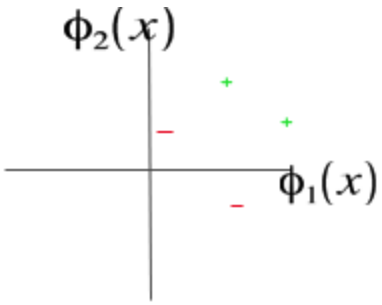
$$\phi\left(x\right) = \left[\sqrt{2}x_1 x_2, x_1^2, x_2^2\right]^T$$

Basically, ϕ transforms x from a linear space to a multi nominal space like shown below:

so that the points can be classified.

For

$$K\left(x, y\right) = e^{-\|x-y\|^2}$$



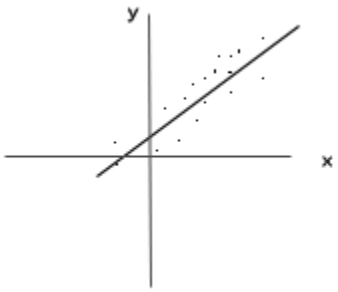
because we have

$$e^x = 1 + x + \frac{x^2}{2} + ...$$

it transforms feature into a infinite dimensional space. Generally Kernel Functions lead to more dimension of w which is K-dimensional so solve dual is more practical.

14.4 Regression

When we are predicting output we actually have a space like this: The line is the prediction line, the points



around it are the data set we have. We predict y with formula:

$$\hat{y} = w^T x$$
$$w = \left(X^T X\right)^{-1} X^T \vec{y}$$

its known as linear regression. The goal is to

$$\min_w \sum_n \frac{1}{2} \|\hat{y}^n - y^n\|^2$$

which leads us to Support Vector Regression:

$$\begin{aligned} \min_w \ & \frac{1}{2}\|w\|^2 + C\sum_n \left(\xi_n + \hat{\xi}_n\right) \\ \text{s.t. } & y^n - w^T x - \xi_n \leq \epsilon \\ & -(y^n - w^T x) - \hat{\xi}_n \leq \epsilon \\ & \xi_n \geq 0 \\ & \hat{\xi}_n \geq 0 \end{aligned}$$

15 Graphical Models

A directed graphical model, also known as a Bayes net or a belief net, is a joint distribution over several variables specificied in terms of a conditional distribution for each variable:

$$P(X_1, X_2, \ldots, X_N) = \prod_i P(X_i|\text{Parents}(X_i))$$

We draw a Bayes net as a graph with a node for each variable, and edges to each node from its parents. This graph expresses the independence relations implicit in the choice of parents for each node. The parent-child edges must form a acyclic graph.

An undirected graphical model is also a distribution specified in terms of a set of functions of sepcific variables:

$$P(X_1, X_2, \ldots, X_N) = \frac{1}{Z} \prod_m f_m(\mathbf{X}_m))$$

where is each $\mathbf{X}_m$ is a subset of $\{X_1, X_2, \ldots, X_N\}$. The are no normalization constraints on the individual factors f_m, and the normaliztion constant Z ensures that the entire joint distribution sums to one:

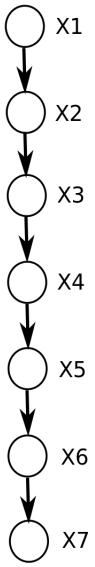
$$Z = \sum_{X_1,\ldots,X_N} \prod_m f_m(\mathbf{X}_m)$$

Suppose that we wish to find the mariginal probability of a variable X_i in a directed graphical model:

$$P(X_i) = \sum_{X_1,\ldots,X_{i-1},X_{i+1},\ldots,X_N} \prod_i P(X_i|\text{Parents}(X_i))$$

For binary variables, there are 2^{N-1} terms in this sum. Our goal in this section is to compute this probability more efficiently by using the structure of network, thus taking advantage of the independence assumptions of the network. The techniques apply to both directed and undirected graphical models. They also apply to the problem of computing conditional probabilities where some variables are known, and we must marginalize over the others.

15.1 Example



To compute $P(X_7|X_2)$, we have

$$P(x_7|x_2) = \frac{1}{Z}\sum_{X_3}\sum_{X_4}\sum_{X_5}\sum_{X_6} P(X_3|x_2)P(X_4|X_3)P(X_5|X_4)P(X_6|X_5)P(x_7|X_6)$$

Suppose every variable X_i is binary, then the summation has $2^4 = 16$ terms. On the other hand, we can use the same trick in dynamic programming by recording every probabilities we have computed for reuse. For example, in above example, if we define

$$f_5(x_5) = \sum_{X_6} P(X_6|X_5 = x_5)P(x_7|X_6) \tag{32}$$

$$f_4(x_4) = \sum_{X_5} P(X_5|X_4 = x_4)f_5(X_5) \tag{33}$$

$$f_3(x_3) = \sum_{X_4} P(X_4|X_3 = x_3)f_4(X_4) \tag{34}$$

$$f_2(x_2) = \sum_{X_3} P(X_3|X_2 = x_2)f_3(X_3) \tag{35}$$

Then the probability above can be computed as

$$P(X_7 = x_7|X_2 = x_2) = \frac{1}{Z}\sum_{X_3}\sum_{X_4}\sum_{X_5}\sum_{X_6} P(X_3|X_2 = x_2)P(X_4|X_3)P(X_5|X_4)P(X_6|X_5)P(X_7 = x_7|X_6) \tag{36}$$

$$= \frac{1}{Z}\sum_{X_3}\sum_{X_4}\sum_{X_5} P(X_3|x_2)P(X_4|X_3)P(X_5|X_4)f_5(X_5) \tag{37}$$

$$= \frac{1}{Z}\sum_{X_3}\sum_{X_4} P(X_3|x_2)P(X_4|X_3)f_4(X_4) \tag{38}$$

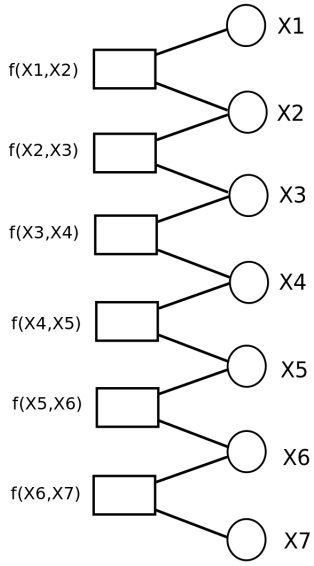
$$= \frac{1}{Z}\sum_{X_3} P(X_3|x_2)f_3(X_3) \tag{39}$$

$$= \frac{1}{Z}f_2(x_2) \tag{40}$$

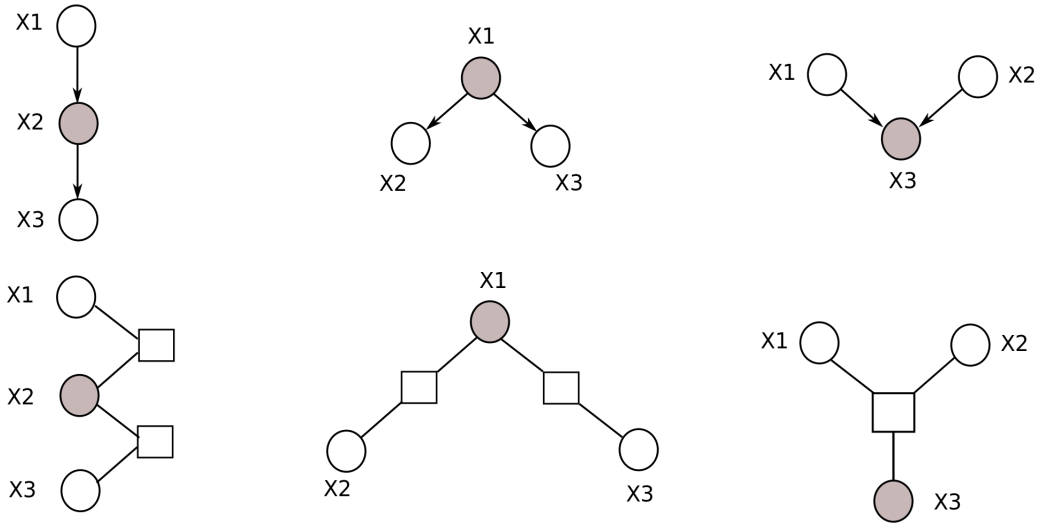
There are 4 sums and each sum needs to compute 2x2 probabilities, so a total of 16 steps.

15.2 Factor Graph

Factor graph is an undirected bipartite graph. There are two types of vertex in a factor graph, factor vertices and variable vertices. Factor vertices correspond to the function f_m in the above example, and each distinct variable vertex corresponds to a distinct variable. If factor function f_m is a function of X_i, its factor vertex is connected to X_i. So the factor graph for above example is,



For more examples,



Note that in the figures above, factor graphs illustrate that the shadowed variable nodes block the information flow from one variable node to another except the last one. In the last example, the two parent nodes are independent, although this cannot be seen from the graph structure. However, the blockage can be read from the table of the factor node in the center. Also note that the last two graphs have same undirected shape, but their factor graphs are different.

15.3 Message Passing (Belief Propagation)

We assume that the factor graph is a tree here. For each variable vertex n and its neighboring factor vertex f_m, the information propagated from n to f_m is,

$$q_{n\to m}(X_n) = \prod_{m'\in M(n)\setminus\{m\}} r_{m'\to n}(X_n)$$

where $M(n)$ is the set of factors touching X_n. This table contains the information propagated from variable n to its neighboring factor vertex f_m. For each factor vertex f_m and its neighboring variable n, the

information propagated from f_m to n is,

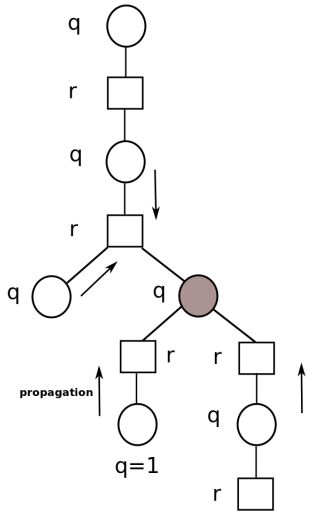
$$r_{m\to n}(X_n) = \sum_{\overrightarrow{X_m}\setminus X_n} f_m(\overrightarrow{X_m}) \prod_{n'\in N(m)\setminus\{n\}} q_{n'\to m}(X_{n'})$$

where $N(m)$ is the set of variables touching f_m. $\sum_{\overrightarrow{X_m}\setminus X_n}$ is the sum is over all variables connected to f_m except X_n. This table contains the information propagated from factor f_m to its neighbor variable n. Note that if variable vertex n is a leaf, $q_{n\to m} = 1$, and if factor vertex m is a leaf, $r_{m\to n} = f_m(X_n)$.

The procedure of message passing or belief propagation is first to propagate the information from leaf vertices to the center (i.e., from leaves to internal nodes) by filling in the tables for each message. Once all the messages variable x_n have been computed, the marginal probability of x_n is computed by combining the incoming messages:

$$P(X_n) = \frac{1}{Z} \prod_{m\in M(n)} r_{m\to n}(X_n)$$

To compute marginal probabilities for all variables, the information is propagated from center back to leaves.



For continuous variables the equation:

$$r_{m\to n} = \sum_{\vec{X}_m\setminus X_n} f(\vec{X_m}) \prod_{n\in N(m)\setminus n} q_{n\to m}$$

changes to

$$r_{m\to n} = \int f(\vec{X}_m) \prod_{n\in N(m)\setminus n} q_{n\to m}\, d(\vec{X}_m\setminus X_n)$$

15.4 Running Time

Suppose in a factor graph, there are N variable vertices and M factor vertices. For every variable vertex n, $|M(n)| < k$ and for every factor vertex f_m, $|N(m)| < l$, the running time is,

$$\mathcal{O}((N+M)(k+l)2^{l-1})$$

16 Junction Tree

From last class, we know that

$$q_{n\to m}(x_n) = \prod_{m'\in M(n)\setminus m} r_{m'\to n}(x_n)$$
$$r_{m\to n}(x_n) = \sum_{\overrightarrow{x_m}\setminus x_n} f_m(\overrightarrow{x_m}) \prod_{n'\in N(m)\setminus\{n\}} q_{n'\to m}(x_{n'})$$

$q_{n\to m}(x_n)$ means the information propagated from variable node n to factor node f_m; $r_{m\to n}(x_n)$ is the information propagated from factor node f_m to variable node n. And our goal is to compute the marginal probability for each variable x_n:

$$P(x_n) = \frac{1}{Z} \prod_{m\in M(n)} r_{m\to n}(x_n).$$

The joint distribution of two variables can be found by, for each joint assignment to both variables, performing message passing to marginalize out all other variables, and then renormalizing the result:

$$P(x_i, x_j) = \frac{1}{Z_{\{i,j\}}} \left(\prod_{m\in M(i)} r_{m\to i}(x_i)\right)\left(\prod_{m\in M(j)} r_{m\to j}(x_j)\right)$$

In the original problem, the marginal probability of variable x_n is obtained by summing the joint distribution over all the variables except x_n:

$$P(x_n) = \frac{1}{Z}\sum_{x_1}\cdots\sum_{x_{n-1}}\sum_{x_{n+1}}\cdots\sum_{x_N}\prod_m f_m(\overrightarrow{x_m}).$$

And by pushing summations inside the products, we obtain the efficient algorithm above.

16.1 Max-Sum

In practice, sometimes we wish to find the set of variables that maximizes the joint distribution $P(x,^N) = \frac{1}{Z}\prod_m f_m(\overrightarrow{x_m})$. Removing the constant factor, it can be expressed as

$$\max_{x_1,\ldots,x_N} \prod_m f_m(\overrightarrow{x_m})$$
$$= \max_{x_1} \ldots \max_{x_N} \prod_m f_m(\overrightarrow{x_m})$$

Figure 3 shows an example, in which the shadowed variables x_j, x_k, and x_l block the outside information flow. So to compute $P(x_i|x_j, x_k, x_l)$, we can forget everything outside them, and just find assignments for inside variables:

$$\max_{\text{inside var}} \prod_m f_m(\overrightarrow{x_m}).$$

Like the *sum-product* algorithm, we can also make use of the distributive law for multiplication and push maxs inside the products to obtain an efficient algorithm. We can put max whenever we see $\sum$ in the *sum-product* algorithm to get the *max-sum* algorithm, which now actually is *max-product* (*Viterbi*) algorithm. For example,

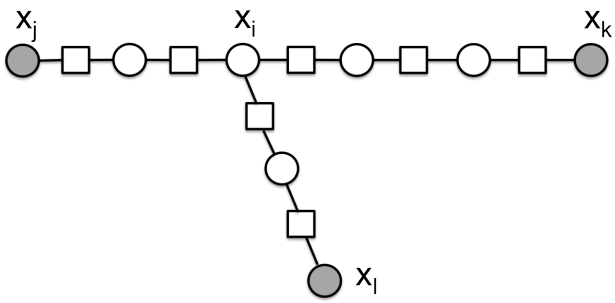


Figure 3: An example of *max-product*

$$r_{m\to n}(x_n) = \max_{\overrightarrow{x_m}\setminus x_n} f_m(\overrightarrow{x_m}) \prod_{n'\in N(m)\setminus\{n\}} q_{n'\to m}(x_{n'})$$

Since products of many small probabilities may lead to numerical underflow, we take the logarithm of the joint distribution, replacing the products in the *max-product* algorithm with sums, so we obtain the *max-sum* algorithm.

$$\max \prod f_m \longrightarrow \max \log \prod f_m \longrightarrow \max \sum \log f_m$$

16.2 Tree Decomposition

If we consider a decision problem instead of a numerical version, the original *max-product* algorithm will be:

$$\text{find} x_1, ..., x_N \text{s.t.} \bigwedge_m f_m(\overrightarrow{x_m}).$$

We need to find some assignments to make it 1, which can be seen as a reduction from the *3-SAT* problem (*constraint satisfaction*). So the problem is $\mathcal{NP}$-complete in general.

To solve the problem, we force the graph to look like a tree, which is *tree decomposition*. Figure 4 shows an example.

Given a *Factor Graph*, we first need to make a new graph (*Dependency Graph*) by replacing each factor with a clique, shown in Figure 5. Then we apply the *tree decomposition*.

Tree decomposition can be explained as: given graph $G = (V, E)$, we want to find $(\{X_i\}, T)$, $X_i \subseteq V$, $T =$ tree over$\{X_i\}$. It should satisfy 3 conditions:

1. $\bigcup_i X_i = V$, which means the new graph should cover all the vertex;
2. For $(u, v) \in E$, $\exists X_i$ such that $u, v \in X_i$;
3. If j is on the path from i to k in T, then $(X_i \cap X_k) \subseteq X_j$ (*running intersection property*).

Using this method, we can get the new graph in Figure 5 with $X_1 = \{A, B\}$, $X_2 = \{B, C, D\}$, and $X_3 = \{D, E\}$. The complexity of original problem is $O((N + M)(k + l)2^{l-1})$, with $l = \max_m |N(m)|$. By *tree decomposition*, we can obtain $l = \max_i |X_i|$. Figure 6 shows the procedure to do *tree decomposition* on a directed graphical model.

A new concept is the *treewidth* of a graph:

$$treewidth(G) = \min_{(\{X_i\},T)} \max_i |X_i| - 1$$

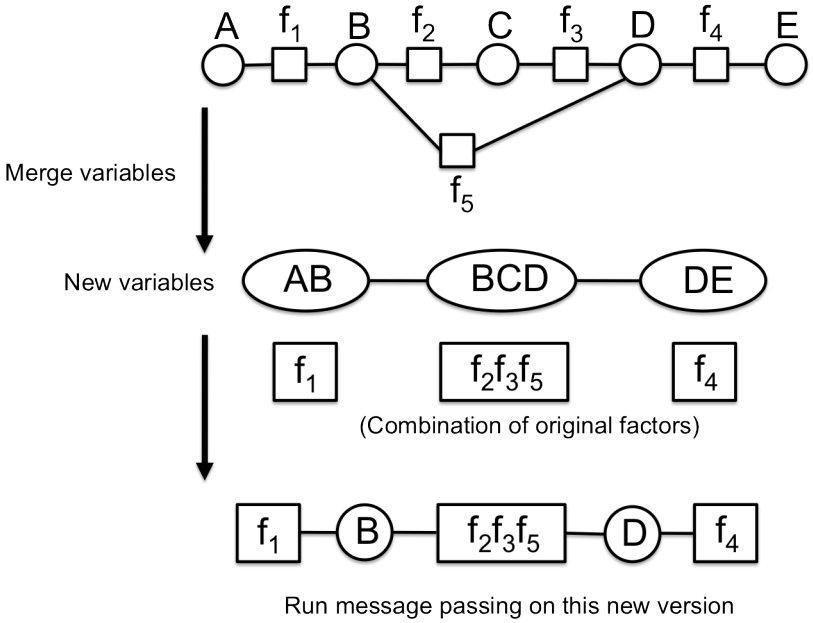


Figure 4: An example of *tree decomposition*

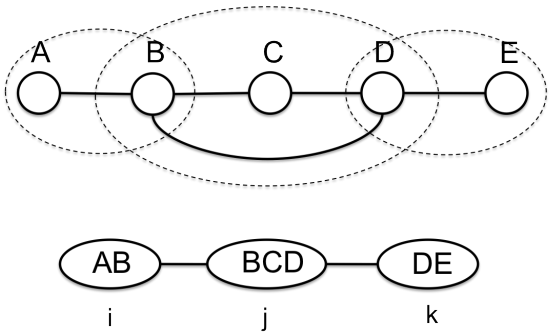


Figure 5: Dependency graph for *tree decomposition* (vertex for each variables)

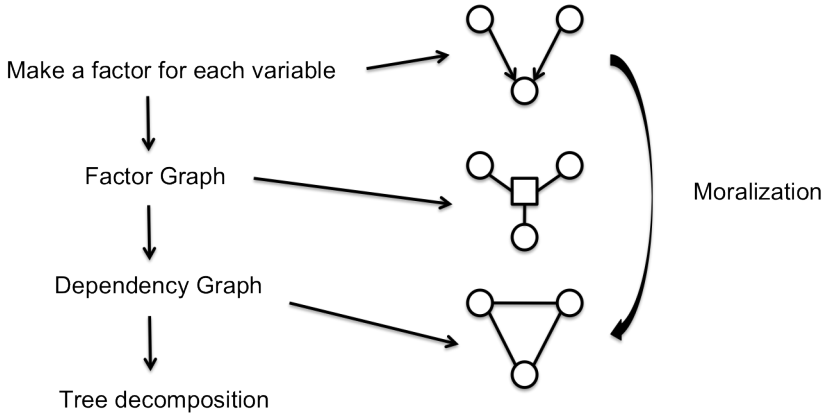


Figure 6: The procedure of *tree decomposition* on a directed graphical model (we can directly get *Dependency Graph* by *moralization*)

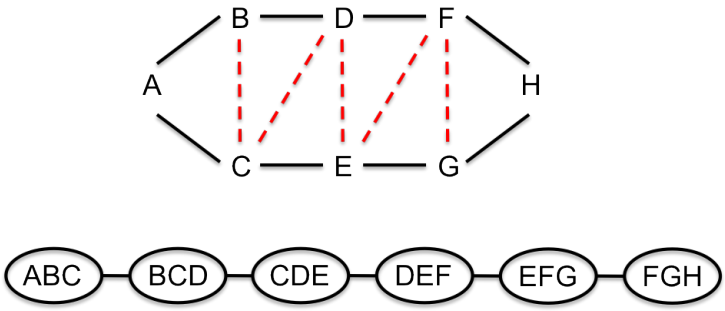


Figure 7: An example of *Vertex Elimination* on a single cycle

For example, $treewidth(tree) = 1$, $treewidth(cycle) = 2$, and the worst case, $treewidth(K_n) = n - 1$ (K_n is a complete graph with n vertices). If the treewidth of the original graph is high, the *tree decomposition* becomes impractical.

Actually, finding the best *tree decomposition* is $\mathcal{NP}$-complete. One practical way is *Vertex Elimination*:

1. choose vertex v (heuristicly, choose v with fewest neighbors);
2. create X_i for v and its neighbors;
3. remove v;
4. connect v's neighbors;
5. repeat the first four steps until no new vertex.

Vertex Elimination cannot ensure to find the optimum solution. Figure 7 shows an example of this method on a single cycle.

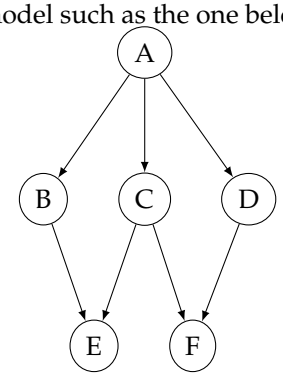
Another way to do *tree decomposition* is *Triangulation*:

1. find cycle without chord (shortcut);
2. add chord;
3. repeat the first two steps until triangulated (no cycles without chords).

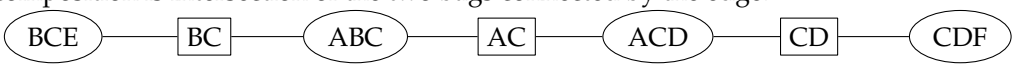
The cliques in the new graph are X_i in the *tree decomposition*.

16.3 Inference on the Tree Decomposition

The tree decomposition can be used to create a new tree-structured factor graph, to which the message passing algorithm can be applied to compute probabilities. For example, given a non-tree-structured graphical model such as the one below:

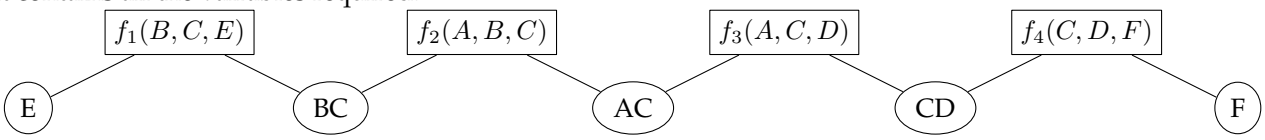


We can compute the tree decomposition shown below. In addition to the bags X_i of the tree decomposition shown in circles, we also show the **separators** in rectangles. The separator associated with an edge in a tree decomposition is intersection of the two bags connected by the edge.



We now create a factor graph with a factor for each bag in the tree decomposition. The value of the factor functions are derived from the factors of the original factor graph: each original factor is assigned to a bag

that contains all the vairables required.



$$
\begin{aligned}
f_1(B,C,E) &= P(E|B,C) \\
f_2(A,B,C) &= P(A)P(B|A)P(C|A) \\
f_3(A,C,D) &= P(D|A) \\
f_4(C,D,F) &= P(F|C,D)
\end{aligned}
$$

This guarantees that the product of all factors in the new factor graph is the same as the product of all factors in the old factor graph. Variables in the new factor graph consist of separators from the tree decomposition, and single variables for any original variable that appear in only one bag, such as E and F above.

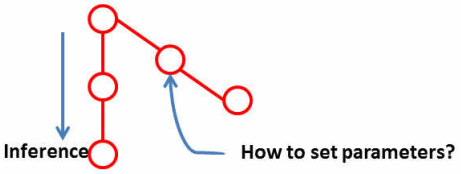
We now generalize the message passing equations to handle the "supervariables" of the new factor graph:

$$
\begin{aligned}
q_{n\to m}(\overrightarrow{x_n}) &= \prod_{m'\in M(n)\setminus m} r_{m'\to n}(\overrightarrow{x_n}) \\
r_{m\to n}(\overrightarrow{x_n}) &= \sum_{\overrightarrow{x_m}\setminus\overrightarrow{x_n}} f_m(\overrightarrow{x_m}) \prod_{n'\in N(m)\setminus\{n\}} q_{n'\to m}(\overrightarrow{x_{n'}})
\end{aligned}
$$

Here $\overrightarrow{x_n}$ represents the set of variables from the original factor graph that are present at node n of the new graph. Messages are tables indexed by combinations of values of these variables. Note that, in the second equation, some varables within each $\overrightarrow{x_{n'}}$ may also be contained in $\overrightarrow{x_n}$. The values of these variables in the incoming q message are bound by the values of $\overrightarrow{x_n}$ on the lefthand side of the equation. This ensures that each original variable has a consistent value at all factors in which it appears in the new factor graph.

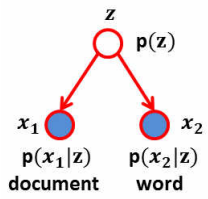
17 Expectation Maximization

In last lecture, we introduced Tree Decomposition. Till now, we have covered a lot as regards how to do inference in a graphical model. In this lecture, we will move back to the learning part. We will consider how to set parameters for the variables.



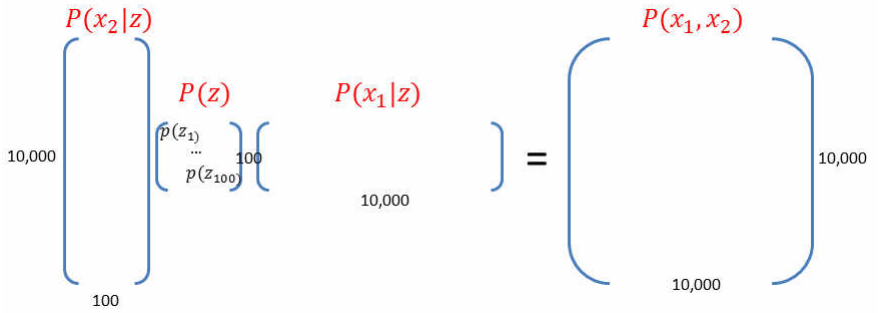
17.1 Parameter Setting: An Example

In the discrete case, we set the parameters just by counting how often each variable occurs. However, we may not know the value of some variables. Thus, in the following, we will discuss learning with hidden (latent) variables. The simplest model is shown below. This model has been used in the Aspect Model for probabilistic Latent Semantic Analysis (pLSA). The variable's value is called an aspect. pLSA has been widely used in information retrieval. In this example, let x_1 and x_2 respectively denote the document ID and word ID. Then, there is a sequence of pair (x_1, x_2), e.g., (1, "the"), (1, "green"), ..., (1000, "the"). In this context, we may have various tasks, e.g., to find the words which co-occur, to find the words on the same topic, or to find the documents containing the same words.



Now, we will introduce the term *cluster*. The reasons why we need the cluster representation are as followed: (1) There are a large amount (e.g., 10,000) of documents, each of which is formed of a large amount (e.g., 10,000) of words. Without a cluster representation, we have to handle a huge query table with too many (say, $10,000^2$) entries. That makes any query difficult. (2) If we still set the parameters just by counting how often each variable occurs, then there is a risk of over-fitting the individuals (i.e., the pair of (document, word)). Because of them, we need to do something smart: clustering. Recall the graphical model displayed above, the hidden (latent) variable z is just the cluster ID.

Note that $x_1 \perp x_2 \| z$. Therefore, the joint distribution $p(x_1, x_2) = \sum_z p(z)p(x_1|z)p(x_2|z)$. As shown in the figure below, now we will not directly compute each entry to obtain the $10,000 \times 10,000$ query table $P(x_1, x_2)$. Instead, we maintain low-rank matrices $P(z)$, $P(x_1|z)$ and $P(x_2|z)$.



Now, we have a set of observed variables $\mathbf{X} = \{(x_1^1, x_2^1), (x_1^2, x_2^2), ...\}$, a set of hidden variables $\mathbf{Z} = \{z_1, z_2, ...\}$ and a set of parameters $\theta = \{\theta_z, \theta_{x_1|z}, \theta_{x_2|z}\}$. Note that x_1^i are i.i.d. variables, and the same for x_2^i. To choose θ, we maximize the likelihoods (MLE): $\max_\theta P(\mathbf{X}; \theta)$.

$$\theta = \underset{\theta}{\operatorname{argmax}} \prod_n P_\theta(x_1^n, x_2^n) = \underset{\theta}{\operatorname{argmax}} \prod_n \sum_z p(z)p(x_1^n|z)p(x_2^n|z) = \underset{\theta}{\operatorname{argmax}} \sum_n \log \sum_z p(z)p(x_1^n|z)p(x_2^n|z) \tag{41}$$

If there is no hidden variable z, we will just count, instead of summing over z. However now, we need to sum over z and find the maximum of the above objective function, which is not a closed-form expression. Thus, it is not feasible to directly set the derivative to zero. To solve this tough optimization problem, we will introduce the Expectation-Maximization (EM) algorithm.

17.2 Expectation-Maximization Algorithm

The EM algorithm is an elegant and powerful method for finding maximum likelihood solutions for models with hidden (latent) variables. It breaks down the potentially tough problem of MLE into two stages. The basic idea is shown below.

E-step:
Guess $\mathbf{z}$
M-step:
MLE to fit θ to $\mathbf{X}, \mathbf{Z}$

Following the above example, we present the EM algorithm in detail.

REPEAT

E-step:

for $n = 1 \dots N$

for $z = 1 \dots K$

$p(z, n) = \theta_z(z) \cdot \theta_{x_1|z}(x_1^n|z) \cdot \theta_{x_2|z}(x_2^n|z)$

$sum \mathrel{+}= p(z, n)$

for z

$p(z, n) \leftarrow \frac{p(z,n)}{sum}$

(An alternative:

$ec(z) \mathrel{+}= p(z, n)$

$ec(z, x_1^n) \mathrel{+}= p(z, n)$

$ec(z, x_2^n) \mathrel{+}= p(z, n)$)

M-step:

for z

$ec(z) = \sum_{n=1}^{N} p(z, n)$

$\theta_z \leftarrow \frac{ec(z)}{N}$

for z, x_1

$ec(z, x_1) = \sum_{n=1}^{N} I(x_1^n = x_1) p(z, n)$

$\theta_{x_1|z} = \frac{ec(z,x_1)}{ec(z)}$

for z, x_2

$ec(z, x_2) = \sum_{n=1}^{N} I(x_2^n = x_2) p(z, n)$

$\theta_{x_2|z} = \frac{ec(z,x_2)}{ec(z)}$

UNTIL convergence

where sum is for normalization and $ec(\cdot)$ denotes the expected count, which is not a real count but an average on what we think z is. Namely, this count is probabilistic. The intuition is to assign some credit to each possible value. Also note that $I(\cdot)$ is an indicator function (return $1/0$ if the condition is true/fasle). In the following, we will derive how to approximate the maximum of the likelihood by maximizing the joint probability's log likelihood iteratively through E-M steps. For the example present in Sec.2, now let us go

further using the same formulation with Eqn. (41).

$$
\begin{aligned}
\theta &= \underset{\theta}{\operatorname{argmax}}\, Q(\theta;\theta^{old}) \\
&= \underset{\theta}{\operatorname{argmax}}\, E_{p(z|x,\theta^{old})} \log p(\mathbf{X},\mathbf{Z}) \\
&= \underset{\theta}{\operatorname{argmax}}\, E_{p(z|x,\theta^{old})} \big[\log \prod_n p(x_1^n, x_2^n, z^n)\big] \\
&= \underset{\theta}{\operatorname{argmax}}\, E\big[\log \prod_n p(z^n)\cdot p(x_1^n|z^n)\cdot p(x_2^n|z^n)\big] \\
&= \underset{\theta}{\operatorname{argmax}}\, E\big[\sum_n \log p(z^n) + \sum_n \log p(x_1^n|z^n) + \sum_n \log p(x_2^n|z^n)\big] \\
\Big(&= \underset{\theta}{\operatorname{argmax}}\, E\big[\sum_n \log \theta_{z^n} + \sum_n \log \theta_{x_1^n|z^n} + \sum_n \log \theta_{x_2^n|z^n}\big] \Big) \\
&= \underset{\theta}{\operatorname{argmax}}\, E\big[\sum_k \sum_n I(z^n=k)\log p(z=k) + \sum_k \sum_n I(x_1^n=x_1|z^n=k)\log p(x_1|z=k) \\
&= \underset{\theta}{\operatorname{argmax}}\, E\big[\sum_k c(z=k)\log p(z=k) + \sum_k c(x_1,z=k)\log p(x_1|z=k) + \sum_k c(x_2,z=k)\log p(x_2|z=k)\big] \\
\Big(&= \underset{\theta}{\operatorname{argmax}} \sum_k E\big[c(z=k)\big]\log\theta_{z=k} + \sum_k E\big[c(x_1,z=k)\big]\log\theta_{x_1|z=k} + \sum_k E\big[c(x_2,z=k)\big]\log\theta_{x_2|z=k} \\
&= \underset{\theta}{\operatorname{argmax}} \sum_k ec(z)\log\theta_{z=k} + \sum_k ec(x_1,z)\log\theta_{x_1|z=k} + \sum_k ec(x_2,z)\log\theta_{x_2|z=k} \Big)
\end{aligned}
$$

Therefore, $\theta = \frac{1}{sum_0} ec(z)$, $\theta_{x_1|z} = \frac{1}{sum_1} ec(x_1,z)$, and $\theta_{x_2|z} = \frac{1}{sum_2} ec(x_2,z)$, but make sure that normalization is done (sum to 1). Notably, $c(\cdot)$ denotes the count and $ec(\cdot)$ denotes the expected count. Also note that $E\big[c(z=k)\big] = ec(z)$ which we have mentioned in the EM algorithm flow, and similar for $ec(x_1,z)$ and $ec(x_2,z)$.

17.3 EM Algorithm in General

Now, we will give a general derivation for the EM algorithm. The denotation will be the same with the above. Similarly, we have $\theta = \operatorname{argmax}_\theta Q(\theta;\theta^{old}) = \operatorname{argmax}_\theta E_{p(z|x,\theta^{old})} \log p(\mathbf{X},\mathbf{Z})$. Let us focus on the objective function Q.

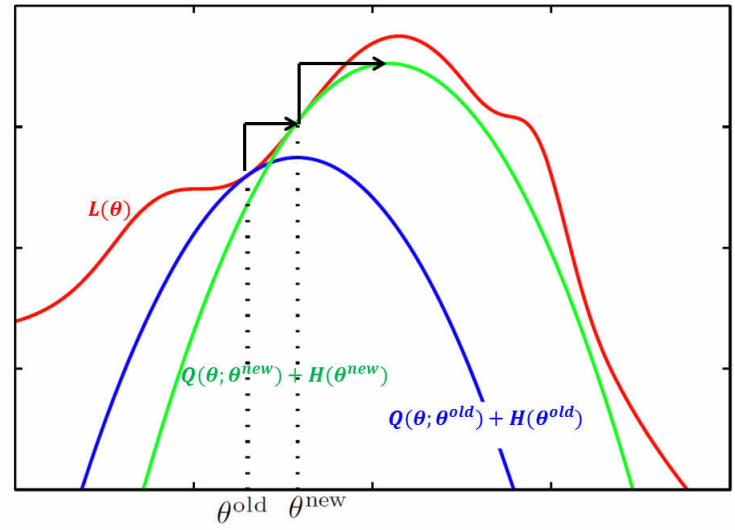
$$
\begin{aligned}
Q(\theta;\theta^{old}) &= E_{p(z|x,\theta^{old})} \log p(\mathbf{X},\mathbf{Z}) \\
&= E_{p(z|x,\theta^{old})} \log \big[p(\mathbf{Z}|\mathbf{X})\cdot p(\mathbf{X})\big] \\
&= E_{p(z|x,\theta^{old})} \big[\log p(\mathbf{Z}|\mathbf{X}) + \log p(\mathbf{X})\big] \\
&= E\Big[\log \frac{p(\mathbf{Z}|\mathbf{X})}{p_{\theta^{old}}(\mathbf{Z}|\mathbf{X})}\cdot p_{\theta^{old}}(\mathbf{Z}|\mathbf{X})\Big] + \log p(\mathbf{X})
\end{aligned}
$$

make it look like K-L divergence

$$
\begin{aligned}
&= E\Big[-\log \frac{p_{\theta^{old}}(\mathbf{Z}|\mathbf{X})}{p(\mathbf{Z}|\mathbf{X})}\Big] - E\big[-\log p_{\theta^{old}}(\mathbf{Z}|\mathbf{X})\big] + \log p(\mathbf{X}) \\
&= -D\Big(\mathbf{Z}|\mathbf{X},\theta^{old}\Big\|\mathbf{Z}|\mathbf{X},\theta\Big) - H\Big(\mathbf{Z}|\mathbf{X},\theta^{old}\Big) + L(\theta)
\end{aligned}
$$

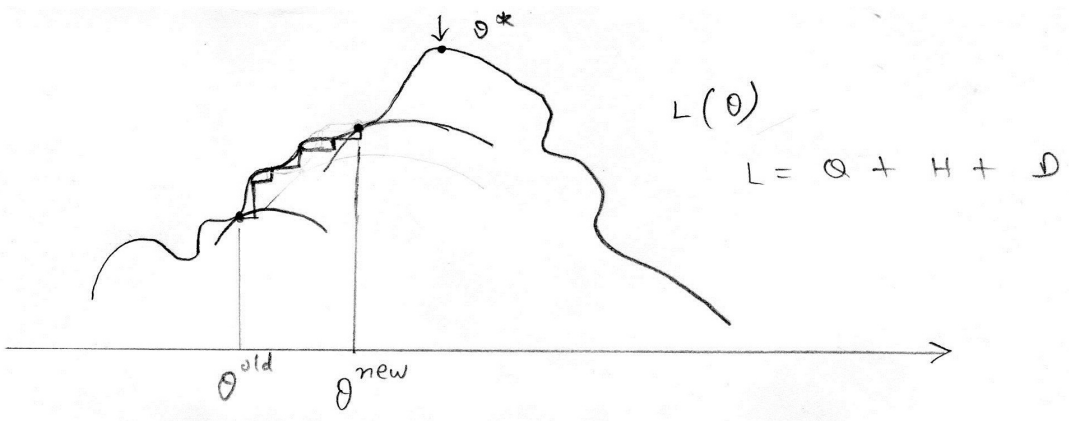
where D, H, L are respectively the K-L divergence, the entropy and the likelihood. Note that our objective is to maximize the likelihood $\log p(\mathbf{X})$. It does not have Z inside, so it can be put out of $E(\cdot)$. Now, we write down $L(\theta)$ with simplified notations:

$$
L(\theta) = Q(\theta;\theta^{old}) + H(\theta^{old}) + D(\theta^{old}\|\theta) \tag{42}
$$



where Q, H and D are all dynamic functions. The approximation can be illustrated in the space of parameter θ, as shown schematically in the figure below. Here the red curve depicts $L(\theta)$ (incomplete data) which we wish to maximize. We start with some initial parameter value θ^{old}, and in the first E step we evaluate the distribution of $Q(\theta;\theta^{old}) + H(\theta^{old})$, as shown by the blue curve. Since the K-L divergence $D(\theta^{old}\|\theta)$ is always positive, the blue curve gives a lower bound to the red curve $L(\theta)$. And $D(\theta^{old}\|\theta)$ just gives the gap between the two curves. Note that the bound makes a tangent contact with $L(\theta)$ at θ^{old}, so that both curves have the same gradient and $D(\theta^{old}\|\theta^{old}) = 0$. Thus, $L(\theta^{old}) = Q(\theta^{old};\theta^{old}) + H(\theta^{old})$. Besides, the bound is a convex function having a unique maximum at $\theta^{new} = \text{argmax}_\theta \left[Q(\theta;\theta^{old})\right]$. In the M step, the bound is maximized giving the value θ^{new}, which also gives a larger value of $L(\theta)$ than θ^{old}: $L(\theta^{new}) = Q(\theta^{new};\theta^{old}) + H(\theta^{old}) + D(\theta^{old}\|\theta^{new})$. In practice, during the beginning iterations, this point is usually still far away from the maximum of $L(\theta)$. However, if we run one more iteration, the result will get better. The subsequent E step then constructs a bound that is tangential at θ^{new} as shown by the green curve. Iteratively, the maximum will be accessed in the end, in a manner kind of similar with gradient ascent. In short, there are two properties for the EM algorithm: (1) the performance gets better step by step. (2) it will converge. At last, it should also be emphasized that EM is not guaranteed to find the global maximum, for there will generally be multiple local maxima. Being sensitive to the initialization, EM may not find the largest of these maxima.

In this lecture, we quickly go through the details of the EM algorithm, which maximizes L through maximizing Q at each iteration step. Then, the remaining problems are how to compute Q, and exactly how to compute $p(\mathbf{Z}|\mathbf{X},\theta^{old})$.



17.4 Gradient Ascent ($\frac{\partial L}{\partial \theta}$)

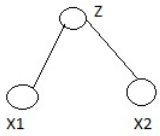
Gradient Ascent $\frac{\partial L}{\partial \theta}$ means the gradient of likelihood function w.r.t. parameters.

Using the gradient ascent, our new parameter θ^{new} is given by :

$$\theta^{new} = \theta^{old} + \eta \frac{\partial L}{\partial \theta}$$

EM (expectation maximization) generally finds a local optimum of θ more quickly, because in this method we optimize the q-step before making a jump. It is similar to saying that we make a big jump first and then take smaller steps accordingly. But for Gradient Ascent method, the step size varies with the likelihood gradient, and so it requires more steps when the gradient is not that steep.

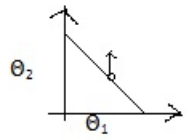
We are required to find $E_{P(Z|X,\theta)}[Z|X]$



$$\text{Our } \theta = \begin{cases} P(Z) \\ P(X_1|Z) \\ P(X_2|Z) \end{cases}$$

which is a long list of all the possible probabilities, after unfolding each of them.

We should make sure that the parameters ($\theta = [\theta_1, \theta_2]$, for only 2 parameters) always stays within the straight line shown below. It should never go out of the line.



$$P(Z = k) = \theta_k = \frac{e^{\lambda}_k}{\sum_{k'} e^{\lambda}_{k'}}$$

In this case, the gradient ascent is used to find the new value of λ

$$\lambda_{new} = \lambda_{old} + \eta \frac{\partial L}{\partial \lambda}$$

17.5 Newton's Method

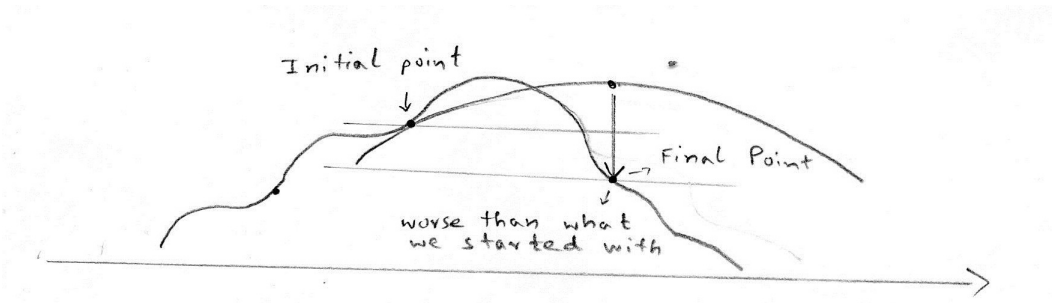
$$\theta^{new} = \theta^{old} + (\nabla^2_\theta L)^{-1}(\nabla_\theta L)$$

In Newton' Method, we approximate the curve with a quadratic function, and we jump to the maximum of the approximation in each step.

Differences between Newton's Method and EM :

1. Newton's Method takes a lot of time, because we need to calculate the Hessian Matrix, which is the 2nd derivative.

2. Since there is no KL divergence in Newton's Method, there is always a chance that we take a big jump, and arrive at a point far away from the global optimum and in fact worse than where we started.



17.6 Variational Method

In this method, at first we trace the Likelihood function by a new parameter, then fix and optimize that parameter, and then once again start the iteration, until we get the optimum value of the Likelihood function.

17.7 Mixture of Gaussians

$$\begin{aligned} P(X) &= \sum_k P(Z=k) N(X|\mu_k, \Sigma_k) \\ &= \sum_k \lambda_k N(X|\mu_k, \Sigma_k), \end{aligned}$$

where μ and Σ are the mean vector and co-variance matrix respectively.

$$N(X|\mu, \Sigma) = \frac{e^{-0.5(X-\mu)^T \Sigma^{-1}(X-\mu)}}{(2\pi)^{D/2}|\Sigma|^{0.5}} \tag{43}$$

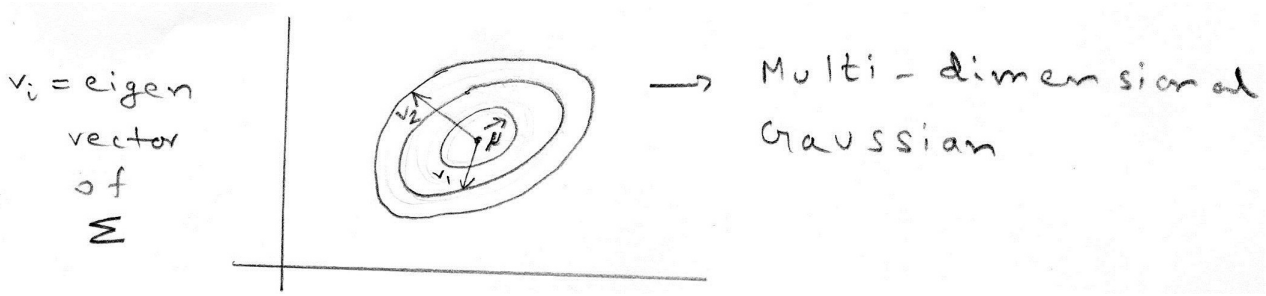
Here X and μ are vectors, Σ is a 2-D matrix, and D is the dimensionality of data X. The lefthand side of equation 43 refers to :

$$N\left(\left[\begin{array}{c} X_1 \\ X_2 \\ \vdots \\ X_N \end{array}\right] \middle| \mu, \Sigma\right)$$

For 1-D data,

$$N(X|\mu, \sigma) = \frac{e^{-0.5(X-\mu)^2/\sigma^2}}{(2\pi)^{0.5}\sigma}$$

Equation (1) is similar to writing as $f(X) = X^T AX$; wherein we are stretching the vector space.



The MLE estimates of the parameters of a Gaussian are as follows:

$$\mu^* = \frac{\sum_n X_n}{N}$$
$$\sigma^* = \sqrt{\frac{\sum_n (X_n - \mu^*)^2}{N}}$$
$$\Sigma^* = \frac{\sum_n (X_n - \mu^*)(X_n - \mu^*)^T}{N}$$, where Σ is the covariance matrix
$$\Sigma_{ij} = \frac{\sum_n (X_n^i - \mu_i)(X_n^j - \mu_j)}{N} = \text{covariance}(\mathrm{X_i}, \mathrm{X_j})$$

Θ (parameters of the model) = λ, μ_k, Σ , where λ and μ_k are vectors

E-step

for $n = 1, 2, \ldots, N$

$$P(Z|X^n) = \frac{P(Z)P(X^n|Z)}{P(X^n)}$$
$$= \frac{\lambda_k \exp[-\frac{1}{2}(X^n - \mu_k)^T \Sigma^{-1}(X^n - \mu_k)]}{\sum_{k'} \lambda_{k'} \exp[-\frac{1}{2}(X^n - \mu_{k'})^T \Sigma^{-1}(X^n - \mu_{k'})]}$$

M-step

for $k = 1, 2, \ldots, K$ (total number of hidden variables)

$$\lambda_k = \frac{\sum_{n=1}^N P(Z = k|X^n)}{N}$$
$$\mu_k = \frac{\sum_n P(Z = k|X^n) X^n}{\sum_n P(Z = k|X^n)}$$
$$\Sigma_k = \frac{\sum_n P(Z = k|X^n)(X^n - \mu_k)(X^n - \mu_k)^T}{\sum_n P(Z = k|X^n)}$$

18 Sampling

We have already studied how to calculate the probability of a variable or variables using the message passing method. However, there are some times when the structure of the graph is too complicated to be calculated. The relation between the diseases and symptoms is a good example, where the variables are all mixed together and brings the graph a high tree width. Another case is that of continuous variable, where during the message passing,

$$r_{m \to n} = \int f(\vec{x}_m) \prod_{n'} q_{n' \to m}(x_{n'}) \mathrm{d}(\vec{x}_m \backslash x_n).$$

If this integration can not be calculated, what can we do to evaluate the probability of variables? This is what sampling is used for.

18.1 Importance Sampling

Suppose we can compute $P(x)$ but not sample from it. Define an auxiliary ditribution $Q(x)$ that is easy to sample from, and which, ideally, approximates $P(x)$. We wish to estimate $E_P[f(x)]$:

$$x^{(n)} \sim Q(x) \qquad n \in 1 \ldots N$$

$$\hat{f} = \frac{1}{N} \sum_n \frac{P(x^{(n)})}{Q(x^{(n)})} f(x^{(n)})$$

In this case, $\hat{f}$ is an unbiased estimator of $E_P[f(x)]$:

$$E_Q[\hat{f}] = \frac{1}{N} \sum_n E_Q \left[\frac{P(x)}{Q(x)} f(x) \right] = \sum_x P(x) f(x) = E_P[f(x)]$$

More often, we cannot compute $P(x)$, but can compute $P^*(x) = ZP(x)$. We can estimate $E_P[f(x)]$ as follows:

$$x^{(n)} \sim Q(x) \qquad n \in 1 \dots N$$

$$\hat{f} = \frac{1}{\sum_n \frac{P(x^{(n)})}{Q(x^{(n)})}} \sum_n \frac{P(x^{(n)})}{Q(x^{(n)})} f(x^{(n)})$$

In this case, $\hat{f}$ is a biased, but consistent, estimator of $E_P[f(x)]$. Consistent means it will converge to the true value as the number of samples increases:

$$\lim_{N \to \infty} \hat{f} = E_P[f(x)]$$

18.2 How to Sample a Continuous Variable: Basics

Now let us forget the above for a moment, say if we want to sample for a continuous variable, how can we ensure that the points we pick up satisfy the distribution of that variable? This question is easy for variables with uniform distribution, since we can generate random numbers directly using a computer. For some complicated distributions, we could use the inverse of cumulative distribution function (CDF) to map the uniform distribution onto the required distribution to generate samples, where the CDF for a distribution with probability distribution function (PDF) of P is

$$\mathrm{CDF}(x) = \int_{-\infty}^{x} P(t) \mathrm{d}t.$$

For example, if we want to sample from a variable with standard normal distribution, the points we pick up are calculated from

$$X = \mathrm{erf}^{-1}(x),$$

where x is drawn from a uniform distribution, and

$$\mathrm{erf}(x) = \int_0^x \mathcal{N}(t, 0, 1) \mathrm{d}t,$$

We could play the same trick for many other distributions. However, there are some distributions which do not have a closed-form integral to calculate their CDF, which makes the above method fail. Under such conditions, we could turn to a framework called *Markov chain Monte Carlo* (MCMC).

18.3 The Metropolis-Hastings Algorithm

Before discussing this method in more detail, let us review some basic properties of Markov chains. A first-order Markov chain is a series of random variables such that each variable depends only on its previous state, that is,

$$x^t \sim P(x^t | x^{t-1}).$$

Our goal is to find a Markov chain which has a distribution similar to a given distribution which we want to sample from, so that by running the Markov chain, we get results as if we were sampling from the original distribution. In other words, we want to have the Markov chain that eventually be able to 1) explore over the entire space of the original distribution, 2) reflect the original PDF.

The general algorithm for generating the samples is called *the Metropolis-Hastings algorithm*. Such an algorithm draws a candidate

$$x' \sim Q(x'; x^t),$$

and then accepts it with probability

$$\min\left\{1, \frac{P(x')Q(x^t; x')}{P(x^t)Q(x'; x^t)}\right\}.$$

The key here is the function Q, called *proposed distribution* which is used to reduce the complexity of the original distribution. Therefore, we have to select a Q that is easy to sample from, for instance, a Gaussian function. Note that there is a trade-off on choosing the variance of the Gaussian, which determines the step size of the Markov chain. If it is too small, it will take a long time, or even make it impossible for the states of the variable to go over the entire space. However, if the variance is too large, the probability of accepting the new candidate will become small, and thus it is possible that the variable will stay on the same state for ever. All these extremes will make the chain fail to simulate the original PDF.

If we sample from P directly, that is $Q(x'; x^t) = P(x')$, we have

$$\frac{P(x')Q(x^t; x')}{P(x^t)Q(x'; x^t)} = 1,$$

which means that the candidate we draw will always be accepted. This tells us that Q should approximate P.

By the way, how do we calculate $P(x)$? There are two cases.

- Although we cannot get the integration of $P(x)$, $P(x)$ itself is easy to compute.
- $P(x) = f(x)/Z$, where $Z = \int f(x)\mathrm{d}x$ is what we do not know. But since we know $f(x) = ZP(x)$, we could just substitute $f(x)$ instead of $P(x)$ in calculating the probability of acceptance of a candidate.

18.4 Proof of the method

In this section, our goal is to prove that the Markov chain generated by the Metropolis-Hastings algorithm has a unique stationary distribution. We will first introduce some basics about the definition of the stationary distribution, and the method to prove this "stationary". Then we will apply those knowledge to accomplish our goal.

1. Stationary distribution

   A distribution with respect to a Markov chain is said to be *stationary* if the distribution remains the same before and after taking one step in the chain, which could be denoted as

   $$\Pi^t = T \times \Pi^{t-1} = \Pi^{t-1},$$

   or

   $$\Pi_i = \sum_j T_{ij}\Pi_j, \quad \forall i,$$

   where Π is a vector which contains the stationary distribution of the state of the variable in each step with its element $\Pi_i = P(x = i)$, and T is the transition probability matrix where its element $T_{ij} = P(x^t = i|x^{t-1} = j)$ denotes the probability that the variable transits from state j to i. For example, the two Markov chains in 8a and 8b all have a stationary distribution $\Pi = \begin{bmatrix}0.5\\0.5\end{bmatrix}$.

   The stationary distribution of a Markov chain could be calculated by solving the equation

   $$\begin{cases} T\Pi = \Pi \\ \sum_i \Pi_i = 1 \end{cases}.$$

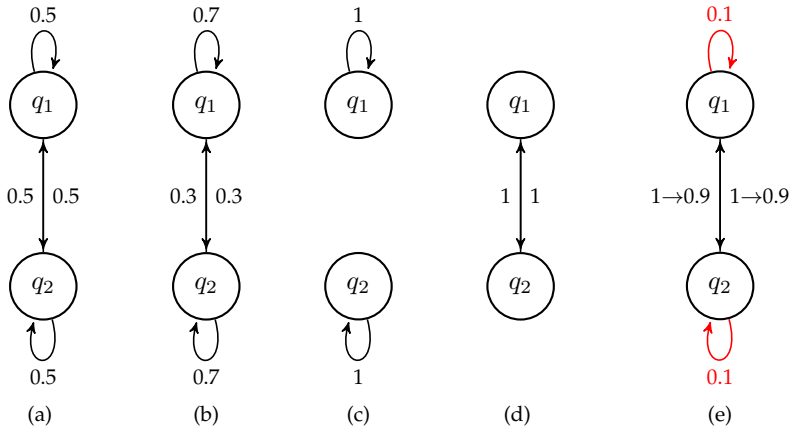


Figure 8: Example Markov Chains

Note that there might be more than one stationary distribution for a Markov chain. A rather simple example would be a chain with a identity transition matrix shown in 8c.

If a Markov chain has a stationary distribution and the stationary distribution is unique, it is ensured that it will converge eventually to that distribution no matter what the original state the chain is.

2. Detailed Balance: Property to ensure the stationary distribution

   Once we know a Markov chain is uniquely stationary, then we can use it to sample from a given distribution. Now, we will see a sufficient (but not necessary) condition for ensuring a Π is stationary, which is a property of the transition matrix called *Detailed Balance*. The definition of such a property is

$$\forall ij, \quad T_{ij}\Pi_j = T_{ji}\Pi_i,$$

   which means $P_{i\to j} = P_{j\to i}$, and is also called reversibility due to the symmetry of the structure.

   Starting from such definition, we have

$$\forall i, \quad \sum_j T_{ij}\Pi_j = \sum_j T_{ji}\Pi_i = \Pi_i \sum_j T_{ji}.$$

   Note that $\sum_j T_{ji} = 1$, we come up with

$$\forall i, \quad \sum_j T_{ij}\Pi_j = \Pi_i \cdot 1 = \Pi_i,$$

   which is exactly the second definition of stationary distribution we have just discussed. Therefore, if a distribution makes the transition matrix of a Markov chain satisfy detailed balance, that distribution is the stationary distribution of that chain. Note that although a periodic Markov chain like that shown in 8d satisfies detailed balance, we do not call it stationary. This because it will not truly converge and thus is not guaranteed to approximate the original PDF. What is more, it is often the case that we add a probability like shown in 8e to avoid such a periodic circumstance.

   Note that the Detailed Balance does not ensure the uniqueness of the stationary distribution of a Markov chain. However, such uniqueness is necessary, or the Markov chain would not go to the PDF we want. What we could do is that, when we construct the chain at the very beginning, we make the chain such that 1) any state is reachable for any other and 2) the chain is aperiodic. Under that condition, we could ensure the uniqueness of the stationary distribution.

3. Final proof

   Now, let us be back to the Metropolis-Hastings algorithm and prove that the transition matrix of its Markov chain has the detailed balance property. If we can prove that, it is obvious that such a Markov chain has a unique stationary distribution.

   According to the Metropolis-Hastings algorithm, the transition probability of the Markov chain of the algorithm is

$$T(x';x) = Q(x';x) \cdot \min\left\{1, \frac{P(x')Q(x;x')}{P(x)Q(x';x)}\right\}$$

   If $x' = x$, then it is automatically detailed balancing due to the symmetry of the definition of detailed balance. To be specific, the condition of detailed balance, which is

$$\forall ij, T_{ij}\Pi_j = T_{ji}\Pi_i,$$

   will always be valid if $i = j$, which is just the case of $x' = x$.

   For the circumstances that $x' \neq x$, by using the distributive property of multiplication, the transition probability is derived as,

$$T(x';x) = \min\left\{Q(x';x), \frac{P(x')Q(x;x')}{P(x)}\right\}.$$

   Multiply both sides by $P(x)$, it turns out that

$$\begin{aligned} T(x';x)P(x) &= \underbrace{\min\left\{Q(x';x)P(x), P(x')Q(x;x')\right\}}_{\text{symmetric for } x \text{ \& } x'} \\ &= T(x;x')P(x') \end{aligned}$$

   Therefore, we proved the detailed balance of the transition matrix, and thus the Markov chain of the Metropolis-Hastings algorithm does have a stationary distribution, which means that we could use such a Markov chain to simulate the original PDF.

18.5 Gibbs Sampling

Now, back to the very first problem of this class, we want to get the result of

$$P(x_k) = \sum_{\vec{x}\setminus x_k} \frac{1}{Z} \prod_m f(\vec{x}_m)$$

without knowing Z. We could use the Gibbs Sampling, shown in Algorithm 1, where $x_{\neg k}$ means all the

Algorithm 1 Gibbs Sampling

```
procedure GIBBS SAMPLING
    repeat
        for k = 1...K do
            x_k ~ P(x_k|x_¬k) = (1/Z') ∏_{m∈M(k)} f(x⃗_m)
    until convergence
```

variables x except x_k. Note that the Gibbs Sampling is actually a particular instance of the Metropolis-Hastings algorithm where the new candidate is always accepted. This is proved as follows.

Substitute

$$\begin{cases} P(x) = P(x_{\neg k})P(x_k|x_{\neg k}) \\ P(x') = P(x'_{\neg k})P(x'_k|x'_{\neg k}) \\ Q(x';x) = P(x'_k|x_{\neg k}) \\ Q(x;x') = P(x_k|x'_{\neg k}) \\ x'_{\neg k} = x_{\neg k} \end{cases}$$

into the probability of the accepting the new candidate, we have

$$\begin{aligned} \frac{P(x')Q(x;x')}{P(x)Q(x';x)} &= \frac{P(x'_{\neg k})P(x'_k|x'_{\neg k})P(x_k|x'_{\neg k})}{P(x_{\neg k})P(x_k|x_{\neg k})P(x'_k|x_{\neg k})} \\ &= 1. \end{aligned}$$

Therefore, the Gibbs Sampling algorithm will always accept the candidate.

18.6 Gibbs sampling with Continuous Variables.

What about the continuous case particularly where sampling is hard? Here we can have a second sampling step:

repeat
  for $k \leftarrow 1 \dots K$

$$\vec{x}_k \sim P(\vec{x}_k|\vec{x}_{\neg k}) = \begin{cases} \frac{1}{Z} \prod_{m \in M(k)} f_m(\vec{x}_m) \\ \text{Metropolis–Hastings} \end{cases}$$

18.7 EM with Gibbs Sampling.

E–step Sample each variable.

M–step Use hard (fixed value of the sample) assignments from sampling in:

$$\lambda_k = \frac{\sum_n I(z^n = k)}{N} = \frac{N_k}{N} \tag{44}$$

In this I is counting how many points are assigned k and we denote this sum as N_k.

$$\mu_k = \frac{\sum_n I(z^n = k)\vec{x}^n}{\sum_n I(z^n = k)} = \frac{\sum_n I(z^n = k)\vec{x}^n}{N_k} \tag{45}$$

The computation of Σ is similar.

18.7.1 Some problems.

What we have above is an approximation to what EM is doing. If we put the computations in expectation it is the same as EM. One problem with EM in general is that if a probability of a cluster hits zero it never comes back. In sampling we can get unlucky and get all zeros and never get that cluster back.

18.7.2 Some advantages.

With sampling we can apply EM to complicated models. For example, a factor graph with cycles or highly connected components. Sampling can be improved by sampling L times in the E–step.

In practice a short cut is taken and we combine the E and the M steps and take advantage of samples immediately:

for $n \leftarrow 1 \dots N$
sample z^n

$\lambda_k \leftarrow \lambda_k + \frac{1}{N} I(z^n = k) - \frac{1}{N} I(z^n_{\text{old}} = k)$
or
$\lambda_k \leftarrow \frac{N_k}{N}$

$\hat{\mu} \leftarrow \hat{\mu} + I(z^n = k)\vec{x}^n - I(z^n_{\text{old}} = k)\vec{x}^n$
$\mu_k \leftarrow \frac{\hat{\mu}}{\lambda_k N}$

We are keeping a running count and updating it as we sample. New observations move data points from their current cluster to a new cluster so we subtract the old observation and add the new one. This technique is widely used and easy to implement.

If we run this long enough it should correspond to the real distribution of hidden variables $P(z^1 \dots z^N | \vec{x}^1 \dots \vec{x}^N)$ — but what does that mean here? Although we have defined $P(z^n | \vec{x}^n; \lambda, \mu, \Sigma)$, the parameters λ, μ, and Σ are changing as sampling progresses. If we run long enough we know that, because the problem is symmetric, $P(z^n = k) = \frac{1}{K}$.

To make sense of this we will make λ, Σ, and μ variables in our model with some distribution $P(\lambda)$. These variables have no parents so we can pick this distribution. We have seen this before and choose to use the Dirichlet distribution so we let $P(\lambda) = \text{Dir}(\alpha)$. Any point is just as likely to be μ so we let $P(\mu) = 1$ and similarly $P(\Sigma) = 1$.

We can take $P(\lambda)$ into account when we sample:

$$z^n \sim \frac{\lambda_k \mathcal{N}(\vec{x}; \mu_k \Sigma_k)}{Z}$$

When we have seen data the probability of the next is:

$$P(z^{N+1} | z^1 \dots z^N) = \frac{c(k) + \alpha}{N + K\alpha}$$

Applying this we have:

$$z^n \sim \frac{\frac{N_k + \alpha}{N + K\alpha} \mathcal{N}(\vec{x}; \mu_k \Sigma_k)}{Z}$$
$$\lambda_k = \frac{N_k + \alpha}{N + K\alpha}$$

Now λ_k can never go all the way to zero. Now if we run long enough we will converge to the real distribution. We have a legitimate Gibbs sampler (states with just relabeling have equal probability). We are sampling with

$$\lambda_k = \frac{1}{Z} \int P(z^1 \dots z^N | \lambda) P(\lambda) \mathrm{d}\lambda$$

The λ is integrated out, giving what is called a collapsed Gibbs sampler.

A remaining question is: when should we stop? If we plot $P(x)$ as we iterate we should see a general trend upward with some small dips and then the curve levels off. But, it could be that in just a few steps a better state could continue the upward trend.

The real reason for doing this Gibbs sampling is to handle a complicated model. One example of that could be a factor graph of diseases and symptoms due to its high tree width. If we try to use EM directly

we have exponential computation for the expected values of the hidden variables. With Gibbs we avoid that problem.

19 PAC learning

See reading from Kearns & Vazirani.

20 Logistic Regression a.k.a. Maximum Entropy

$$P(y|x) = \frac{1}{Z_x} e^{\sum_i \lambda_i f_i(x,y)} \tag{46}$$

$$Z_x = \sum_y e^{\sum_i \lambda_i f_i(x,y)} \tag{47}$$

For example, in NLP, we may use

$$f_{100}(x,y) = \begin{cases} 1 & \text{if x=word that ends with 'tion' and y=Noun}\,, \\ 0 & \text{otherwise}\,. \end{cases}$$

The above is more general binary classification where we are only deciding whether an example belongs to a class or NOT. Here, we can have features contributing to multiple classes according to their weights. For binary classification, the decision boundary is linear, as with perceptron or SVM. A major difference from SVMs is that, during training, every example contributes to the objective function, whereas in SVMs only the examples close to the decision boundary matter.

If we plot this function, we get a sigmoid-like graph. We can draw analogy between maximum entropy and neural network, and consider features as the input nodes in the neural network.

If we take the log of equation (46), we get a linear equation

$$\log P = \sum_i \lambda_i f_i + c$$

What should λ be?

$$\begin{aligned} \max_\lambda \log \left(\prod_{n=1}^{N} P(y_n|x_n) \right)^{\frac{1}{N}} &= \max_\lambda \sum_n \frac{1}{N} \log \left(\frac{1}{Z_x} e^{\sum_i \lambda_i f_i} \right) \\ &= \max_\lambda \frac{1}{N} \sum_n \left(\sum_i \lambda_i f_i - \log Z_x \right) \end{aligned}$$

In the above equation, the first term is a simple function of λ is easy, but the second term is more complex. To maximize w.r.t. λ, we turn it into a concave form and find the point where the derivative w.r.t.

λ is zero (hill climbing)

$$L = \frac{1}{N}\sum_n \sum_i \lambda_i f_i - \log \sum_y e^{\sum_i \lambda_i f_i(x_n, y)} \tag{48}$$

$$\frac{\partial L}{\partial \lambda_j} = \frac{1}{N}\sum_n f_j - \frac{1}{Z_x}\frac{\partial}{\partial \lambda_j}\left(\sum_y e^{\sum_i \lambda_i f_i}\right) \tag{49}$$

$$= \frac{1}{N}\sum_n \left[f_j - \frac{1}{Z_x}\sum_y f_j e^{\sum_i \lambda_i f_i(y, x_n)}\right] \tag{50}$$

$$= \frac{1}{N}\sum_n f_j - \sum_y f_j P\left(y|x_n\right) \tag{51}$$

$$= \frac{1}{N}\sum_n f_j(x_n, y_n) - \sum_y f_j P\left(y|x_n\right) \tag{52}$$

We can morph eq. 52 into expectation form by defining joint probability as follows:

$$P(y, x) = P\left(y|x\right)\tilde{P}(x) \tag{53}$$

$$\tilde{P}(x) = \frac{1}{N}\sum_n I(x_n = x) \tag{54}$$

$$\tilde{P}(y|x) = \frac{c(x_n = x, y_n = y)}{c(x_n = x)} \tag{55}$$

Rewriting eq. 52 in expectation form, we get:

$$\frac{\partial L}{\partial \lambda_j} = E_{\tilde{P}}[f_j] - E_P[f_j] \tag{56}$$

where the first term (before the minus) is a constant, and the complexity of calculating the second term depends on the number of classes in the problem. Now we have:

$$\lambda \leftarrow \lambda + \eta \frac{\partial L}{\partial \lambda} \tag{57}$$

We will justify why we chose log linear form instead of something else. Assume we want to find the maximum entropy subject to constraints on the feature expectations:

$$\max_{P(y|x)} \; H(y|x) \tag{58}$$

$$\min_{P(y|x)} \; -H(y|x) \tag{59}$$

$$\text{s.t. } E_{\tilde{P}}[f_i] = E_P[f_i] \qquad \forall i \tag{60}$$

$$\sum_y P(y|x) = 1 \qquad \forall x \tag{61}$$

To put this into words, we want to build a model such that for each feature, our model should match the training data. We have

$$H(y|x) = \sum_{x,y} \tilde{P}(x)P(y|x)\log\frac{1}{P(y|x)} \tag{62}$$

Find the maximum entropy of the above equation as follows

$$L(P,\lambda,\mu) = f_0 + \sum_j \lambda_j f_j \tag{63}$$

$$= \sum_{x,y} \tilde{P}(x)P(y|x)\log P(y|x) \tag{64}$$

$$+ \sum_i \lambda_i \left(\sum_{x,y} \tilde{P}(x)\tilde{P}(y|x)f_i - \tilde{P}(x)P(y|x)f_i \right) \tag{65}$$

$$+ \sum_x \tilde{P}(x)\mu_x \left(\sum_y P(y|x) - 1 \right) \tag{66}$$

$$\frac{\partial L}{\partial P(y|x)} = \tilde{P}(x)\left(\log P(y|x) + 1\right) - \sum_i \lambda_i \tilde{P}(x) f_i + \tilde{P}(x)\mu_x = 0 \tag{67}$$

$$\log P(y|x) = -1 + \sum_i \lambda_i f_i - \mu_x \tag{68}$$

$$P(y|x) = e^{-1-\mu_x} e^{\sum_i \lambda_i f_i} \tag{69}$$

$$= \frac{1}{Z_x} e^{\sum_i \lambda_i f_i} \tag{70}$$

The above result shows that maximum entropy has log-linear form. If we solve the dual of the problem

$$g(\lambda,\mu) = E_P\left[\sum_i \lambda_i f_i - 1 - \mu_x\right] + E_{\tilde{P}}\left[\sum_i \lambda_i f_i\right] - E_P\left[\sum_i \lambda_i f_i\right] + E_P\left[\mu_x\right] - \sum_x \tilde{P}(x)\mu_x \tag{71}$$

$$= E_P\left[-1 - \mu_x\right] + E_{\tilde{P}}\left[\sum_i \lambda_i f_i\right] + E_P\left[\mu_x\right] - \sum_x \tilde{P}(x)\mu_x \tag{72}$$

$$= E_P\left[-1\right] + E_{\tilde{P}}\left[\sum_i \lambda_i f_i\right] - \sum_x \tilde{P}(x)\mu_x \tag{73}$$

$$= -\sum_{x,y} \tilde{P}(x) e^{-1-\mu_x} e^{\sum_i \lambda_i f_i} + E_{\tilde{P}}\left[\sum_i \lambda_i f_i\right] - \sum_x \tilde{P}(x)\mu_x \tag{74}$$

$$\tag{75}$$

Solving analytically for μ_x that maximzes g:

$$0 = \frac{\partial g}{\partial \mu_x} = \tilde{P}(x)\left(\sum_y e^{-1-\mu_x+\sum_i \lambda_i f_i} - 1\right) \tag{76}$$

$$1 = \sum_y e^{-1-\mu_x+\sum_i \lambda_i f_i} \tag{77}$$

$$e^{\mu_x} = \sum_y e^{-1+\sum_i \lambda_i f_i} \tag{78}$$

$$\mu_x = \log \sum_y e^{\sum_i \lambda_i f_i} - 1 \tag{79}$$

Substituting μ_x into g:

$$g(\lambda, \mu) = -\sum_{x,y} \tilde{P}(x) \frac{1}{\sum_y e^{\sum_i \lambda_i f_i}} e^{\sum_i \lambda_i f_i} + E_{\tilde{P}} \left[\sum_i \lambda_i f_i \right] - \sum_x \tilde{P}(x) \left(\log \sum_y e^{\sum_i \lambda_i f_i} - 1 \right) \tag{80}$$

$$= -1 + E_{\tilde{P}} \left[\sum_i \lambda_i f_i \right] - \sum_x \tilde{P}(x) \log \left(\sum_y e^{\sum_i \lambda_i f_i} \right) + 1 \tag{81}$$

$$= E_{\tilde{P}} \left[\sum_i \lambda_i f_i \right] - \sum_x \tilde{P}(x) \log \left(\sum_y e^{\sum_i \lambda_i f_i} \right) \tag{82}$$

$$= E_{\tilde{P}} \left[\sum_i \lambda_i f_i - \log \left(\sum_y e^{\sum_i \lambda_i f_i} \right) \right] \tag{83}$$

$$= E_{\tilde{P}} \left[\sum_i \lambda_i f_i - \log Z_x \right] \tag{84}$$

$$= E_{\tilde{P}} \left[\log P(y|x) \right] \tag{85}$$

$$= L \tag{86}$$

Thus solving the dual of the entropy maximization problem consists of maximizing the likelihood of the training data with a log-linear functional form for $P(y|x)$.

21 Hidden Markov Models

A Hidden Markov Model (HMM) is a Markov Chain (a series of states with probabilities of transitioning from one state to another) where the states are hidden (latent) and each state has an emission as a random variable. The model is described as follows:

- Ω : the set of states, with $y_i \in \Omega$ denoting a particular state
- Σ : the set of possible emissions with $x_i \in \Sigma$ denoting a particular emission
- $P \in \mathbb{R}_{[0,1]}^{\Omega \times \Omega}$: the matrix with each element giving the probability of a transition
- $Q \in \mathbb{R}_{[0,1]}^{\Omega \times \Sigma}$: the matrix with each element giving the probability of an emission
- Π : the matrix with each element giving the probability of starting in each state

The probability distribution of an HMM can be decomposed as follows:

$$P(x_1, \ldots x_n, y_1, \ldots, y_n) = \Pi(y_1) \prod_{i=1}^{n-1} P(y_i, y_{i+1}) \prod_{i=1}^{n} Q(y_i, x_i)$$

An example HMM is given:

$$\Omega = \{1, 2\}$$

$$\Sigma = \{a, b, c\}$$

$$P = \begin{pmatrix} \frac{1}{3} & \frac{2}{3} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix}$$

$$Q = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{3} & \frac{1}{2} & \frac{1}{6} \end{pmatrix}$$

One possible sequence of observations would be:

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

$$\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow\downarrow$$
$$a\ b\ c\ a\ a\ a\ a\ a\ b$$

We can consider multiple problems relating to HMMs.

1. Decoding I: Given $x_1, \ldots, x_n, P, Q, \Pi$, determine the sequence $y_1 \ldots y_n$ that maximizes $P(Y_1, \ldots Y_n | X_1, \ldots X_n)$.
2. Decoding II: Given $x_1, \ldots, x_n$ and t, determine the distribution of y_K, that is, for all values a of y_t, $P(y_t = a | X_1, \ldots, X_n)$.
3. Evaluation: Given $x_1, \ldots x_n$, determine $P(X_1, \ldots X_n)$.
4. Learning: Given a sequence of observations, $x_1^{(1)}, \ldots x_n^{(1)}, \ldots x_1^{(k)}, \ldots x_n^{(k)}$, learn P, Q, Π that maximize the likelihood of the observed data.

We define two functions, α and β.

$$\alpha^t(a) := P(X_1 = x_1, \ldots, X_t = x_t, Y_t = a)$$

$$\beta^t(a) := P(X_{t+1} = x_{t+1}, \ldots, X_n = x_n \mid Y_t = a)$$

which are also recursively defined as follows:

$$\alpha^{t+1}(a) = \sum_{c \in \Omega} \alpha^t(c) P(c, a) Q(a, x_{t+1})$$

$$\beta^{t-1}(a) = \sum_{c \in \Omega} Q(c, x_t) \beta^t(c) P(a, c)$$

We return to the Decoding II problem. Given $x_1, \ldots, x_n$ and t, determine the distribution of Y_K, that is, for all values a of Y_t, $P(Y_t = a | X_1, \ldots, X_n)$. To do this, we rewrite the equation as follows:

$$P(y_t = a | X_1, \ldots, X_n) = \frac{P(X_1, \ldots, X_n, Y_t = a)}{P(X_1, \ldots, X_n)}.$$

However, we need to calculate $P(X_1, \ldots, X_n)$. We can do this using either α or β.

$$P(X_1, \ldots, X_n) = \sum_{a \in \Omega} \alpha^n(a)$$

$$= \sum_{a \in \Omega} \beta^1(a) \Pi(a) Q(a, x_1)$$

The Decoding I problem can be solved with Dynamic Programming. (Given $x_1, \ldots, x_n, P, Q, \Pi$, determine the sequence $y_1 \ldots y_n$ that maximizes $P(Y_1, \ldots Y_n | X_1, \ldots X_n)$.) We can fill in a table with the following values:

$$T[t, a] = \max_{y_1 \ldots y_t, y_t = a} P(y_1, \ldots y_t | X_1, \ldots X_t)$$

which means that each value is the probability of the most likely sequence at time t with the last emission being a. This can be computed using earlier values with the following formula:

$$T[t+1, a] = \max_{c \in \Omega} T[t, c] P(c, a) Q(a, x_{t+1})$$

To compute the most likely sequence, we simply solve

$$\max_{a \in \Omega} T[n, a]$$

The learning problem can be solved using EM. Given the number of internal states, and $x_1, \ldots x_n$, we want to figure out P, Q, and Π. In the E step, we want to compute an expectation over hidden variables:

$$L(\theta, q) = \sum_y q(Y|X) \log \frac{P(X, Y|\theta)}{q(Y|X)}$$

For HMM's, the number of possible hidden state sequences is exponential, so we use dynamic programming to compute expected counts of individual transitions and emissions:

$$P(a, b) \propto \sum_{i=1}^{n-1} q(Y_i = a, Y_{i-1} = b|X_1 \ldots X_n) \tag{87}$$

$$Q(a, w) \propto \sum_{i=1}^{n} q(Y_i = a|X_1 \ldots X_n) I(X = w) \tag{88}$$

The new P is defined as:

$$P^{\text{new}}(a, b) \propto \sum_{i=1}^{n-1} \alpha^i(a) P^{\text{old}}(a, b) Q^{\text{old}}(b, x_{i+1}) \beta^{i+1}(b)$$

The new Q is defined as:

$$Q^{\text{new}}(a, w) \propto \sum_{i: x_i = w} \alpha^i(a) \beta^{i+1}(a)$$

22 LBFGS

To train maximum entropy (logistic regression) models, we maximized the probability of the training data over possible feature weights λ:

$$\max_\lambda \prod_n P(Y_n|X_n)$$

It is to maximize

$$L = \log \prod_n \frac{1}{Z_{X_n}} e^{\sum_i \lambda_i f_i}$$

Of course we can solve it by using gradient ascend, but we today will talk about using an approximation of Newton's method.

22.1 Preliminary

For quadratic objective function

$$f(x) = \frac{1}{2} x^\top A x + b^\top x + c$$

Newton's iteration is given by

$$x_{k+1} = x_k + (\nabla^2 f(x_k))^{-1} \nabla f(x_k)$$

However, the exact version of Newton's method involves a few problems as follows:

- We need to compute Hessian $\nabla^2 f$, which is expensive.
- We also need to invert Hessian, which is also a costly operation.
- Furthermore, we need to store Hessian, which is expensive in terms of space.

So in order to compute Newton's iteration in a fast but relatively accurate way, an approximation should be developed. L-BFGS is one of them.

22.2 The BFGS Algorithm

Let B_k denote our approximation of the Hessian $\nabla^2 f(x_k)$ and then we can write Newton's iteration as

$$x_{k+1} = x_k - \alpha B_k^{-1} \nabla f(x_k)$$

Because the Hessian can be seen as the second order derivative of f, we wish to choosse B_k such that:

$$B_k(x_{k+1} - x_k) = \nabla f(x_{k+1}) - \nabla f(x_k)$$

Let

$$s_k = x_{k+1} - x_k$$
$$y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$$

then we have

$$B_k s_k = y_k$$

This is to say that, our approximation is a solution of above equation. Consider

$$B_k = \frac{y_k y_k^\top}{s_k^\top y_k}$$

Because

$$B_k s_k = \frac{y_k y_k^\top s_k}{s_k^\top y_k} = \frac{y_k (y_k^\top s_k)}{s_k^\top y_k} = y_k$$

Further, let H_k be our approximation of $(\nabla^2 f(x_k))^{-1}$, the inverse of Hessian. We will have:

$$s_k = H_k y_k$$

H_k is a solution of above equation. One such H_k is given by

$$H_k = \frac{s_k s_k^\top}{s_k^\top y_k}$$

So, we want a direct formula of computing a symmetric H_{k+1} from H_k. That is, we want to fill in the ? term in following equation

$$H_{k+1} = H_k + \frac{s_k s_k^\top}{s_k^\top y_k} + ?$$
$$\text{such that } s_k = H_{k+1} y_k$$

With careful proofs and calculation, we get

$$H_{k+1} = (I - \rho_k s_k y_k^\top) H_k (I - \rho_k y_k s_k^\top) + \rho_k s_k s_k^\top \tag{89}$$

where $\rho_k = \frac{1}{s_k^\top y_k}$.

22.3 Proof of the method

The problem can be formalized as minimizing $\|H_{k+1} - H_k\|$ such that $H_{k+1}y_k = s_k$ and $H_{k+1}^\top = H_{k+1}$. Here $\|\cdot\|$ is defined as follows:

$$\|A\|^2 = \|W^{\frac{1}{2}} A W^{\frac{1}{2}}\|_f^2$$

where $\|A\|_f$ is defined as the square sum of all entries, $\sum_{i,j} a_{ij}^2$ and W is any matrix such that $Ws_k = y_k$.

It is easy to verify that $H_{k+1}y_k = s_k$ and H_{k+1} is symmetric, as follows:

$$\begin{aligned} H_{k+1}y_k &= (I - \rho_k s_k y_k^\top) H_k (I - \rho_k y_k s_k^\top) y_k + \rho_k s_k s_k^\top y_k \\ &= ...H_k(y_k - \rho_k s_k^\top y_k) + \rho_k s_k s_k^\top y_k \\ &= ...H_k(y_k - y_k) + s_k \\ &= s_k \end{aligned}$$

22.4 L-BFGS Algorithm

L-BFGS algorithm tries to approximate $H_{k+1}\nabla f(x_{k+1})$ together. From 89, we can unroll the last m H_k's. Then we will compute H_{k+1} directly from H_{k-m}.

$$\begin{aligned} H_{k+1} =& V_k^\top ..V_{k-m}^\top H_{k-m} V_{k-m}..V_k \\ &+ \rho_{k-m} V_k^\top ..V_{k-m}^\top s_{k-m} s_{k-m}^\top V_{k-m}..V_k \\ &+ .. \\ &+ \rho_k s_k s_k^\top \end{aligned}$$

In the equation above, $V_k = I - \rho_k s_k y_k^\top$. The algorithm is

Algorithm 2 L-BFGS

procedure LBFGS(H_{k-m}, s_i, y_i)
  $q \leftarrow \nabla f_k$
  for $i = k-1..k-m$ **do**
    $\alpha_i \leftarrow \rho_i s_i^\top q$
    $q \leftarrow q - \alpha_i y_i$
  $r \leftarrow H_{k-m} q$
  for $i = k-m..k-1$ **do**
    $\beta \leftarrow \rho_i y_i^\top r$
    $r \leftarrow r + s_i(\alpha_i - \beta)$
  return r

In the algorithm, $V_k = I - \rho_k y_k s_k^\top$. This algorithm needs to keep track of s_k and y_k in the last m steps and each step requires $2n$ space(n for s_k and n for y_k). So a total of $O(2mn)$ space is needed.

There are many user libraries that have already implemented this algorithm, so we can just use them for our computing.

23 Gradient Descent

Gradient descent is guaranteed to converge in the limit under very general assumptions:

$$\lim_{t\to\infty} \Big(f(x^{(t)}) - f(x^*) \Big) = 0 \quad \begin{cases} \sum_{t=1}^{\infty} \eta_t = +\infty \\ \lim_{t\to\infty} \eta_t = 0 \end{cases}$$

We now analyze the speed of convergence with a fixed learning rate under specific assumptions about $f(x)$.

Theorem 1. *Assume $f(x)$ is differentiable and convex, $\nabla f(x)$ is L-Lipschitzian, i.e. $\|\nabla f(x') - \nabla f(x'')\| \leq L\|x' - x''\|$, then $f(x^{(t)}) - f(x^*) = O(\frac{1}{t})$ when $\eta < \frac{1}{2L}$.*

The assumption that $f(x)$ is L-Lipschitzian gives a quadratic upper bound:

$$f(x) \leq f(x') + (x - x')^T \nabla f(x') + \frac{L}{2}\|x - x'\|^2 \tag{90}$$

The assumption that $f(x)$ is convex gives a linear lower bound:

$$f(x) \geq f(x') + (x - x')^T \nabla f(x') \tag{91}$$

The update rule for gradient decent is:

$$x^{(t+1)} = x^{(t)} - \eta \nabla f(x^{(t)})$$

The learning rate is chosen to guarantee that we make progress at each step. When we are up against the quadratic upper bound, $f(x) = \frac{L}{2}\|x\|^2$ and $\eta = \frac{1}{2L}$, and $f(x^{(t+1)}) = f(x^{(t)})$. Any smaller step size will guarantee that $f(x^{(t+1)}) < f(x^{(t)})$.

Proof.

$$\begin{aligned}
\|x^{(t+1)} - x^*\|^2 &= \|x^{(t)} - x^* - \eta \nabla f(x^{(t)})\|^2 \\
&= \left(x^{(t)} - x^* - \eta \nabla f(x^{(t)})\right)^T \left(x^{(t)} - x^* - \eta \nabla f(x^{(t)})\right) \\
&= \|x^{(t)} - x^*\|^2 + \eta^2 \|\nabla f(x^{(t)})\|^2 + 2\eta \nabla f(x^{(t)})^T (x^* - x^{(t)})
\end{aligned}$$

From Equation 91, $f(x) \geq f(x') + (x - x')^T \nabla f(x')$:

$$\|x^{(t+1)} - x^*\|^2 \leq \|x^{(t)} - x^*\|^2 + \eta^2 \|\nabla f(x^{(t)})\|^2 + 2\eta(f(x^*) - f(x^{(t)})) \tag{92}$$

The quadratic upper bound of equation 90 implies that:

$$\begin{aligned}
f(x^*) \leq f(x') &\leq f(x) + \nabla f(x)^T (x' - x) + \frac{L}{2}\|x' - x\|^2 \\
f(x^*) &\leq f(x) - \frac{1}{2L}\|\nabla f(x)\|^2 \qquad \text{choosing } x' = x - \frac{1}{L}\nabla f(x) \\
\|\nabla f(x)\|^2 &\leq 2L(f(x) - f(x^*))
\end{aligned}$$

Substituting into equation 92:

$$\begin{aligned}
\|x^{(t+1)} - x^*\|^2 &\leq \|x^{(t)} - x^*\|^2 + 2L\eta^2 (f(x^{(t)}) - f(x^*)) + 2\eta(f(x^*) - f(x^{(t)})) \\
&= \|x^{(t)} - x^*\|^2 - 2(\eta - \eta^2 L)(f(x^{(t)}) - f(x^*)) \\
&\leq \|x^{(t)} - x^*\|^2 - \eta(f(x^{(t)}) - f(x^*))
\end{aligned}$$

when $L\eta \leq \frac{1}{2}$.
Therefore:

$$\begin{aligned}
\frac{1}{T}\sum_{t=0}^{T} \eta(f(x^{(t)}) - f(x^*))) &\leq \frac{1}{T}\sum_{t=0}^{T} \left(\|x^{(t)} - x^*\|^2 - \|x^{(t+1)} - x^*\|^2\right) \\
&= \frac{1}{T}\|x^{(0)} - x^*\|^2 - \frac{1}{T}\|x^{(T+1)} - x^*\|^2 \\
&\leq \frac{1}{T}\|x^{(0)} - x^*\|^2
\end{aligned}$$

$$f(x^{(t)}) - f(x^*) \leq \frac{1}{\eta t}\|x^{(0)} - x^*\|^2 = O\left(\frac{1}{t}\right)$$

□

Theorem 2. *When $f(x)$ is strongly convex, gradient descent has a faster convergence rate, i.e. $f(x^{(t)}) - f(x^*) = O\left(C^{-t}\right)$*

A function is strongly convex with parameter μ if:

$$f(x) \geq f(x') - (x - x')^T \nabla f(x) + \frac{\mu}{2}\|x - x'\|^2$$

23.1 Stochastic Gradient Descent

$$x^{(t+1)} = x^{(t)} - \eta_t g_t \quad \begin{cases} E[g_t] = \nabla f(x) \\ \lim_{t\to\infty} \eta_t = 0 \end{cases}$$

Theorem 3. *When $\eta_t = \frac{1}{\sqrt{t}}$, then $E\left(\|x^{(t)} - x^*\|^2\right) = O\left(\frac{1}{\sqrt{t}}\right)$.*

24 Principal Component Analysis

Principal Component Analysis is type of dimensionality reduction, and represents each point $x^{(n)} \in \mathbb{R}^d$ with a lower-dimensional representation $z^{(n)} \in \mathbb{R}^d$, where $m < d$. The two spaces are related through a set of basis vectors u_i, which are orthonormal:

$$u_i^T u_j = \delta_{ij}$$

$$x^{(n)} = \sum_i {z_i^{(n)}}^T u_i$$

$$z_i^{(n)} = u_i^T x^{(n)}$$

Our reconstructed points $\tilde{x}^{(n)}$ are obtained by fixing the components in $d - m$ of the basis directions:

$$\tilde{x}^{(n)} = \sum_{i=1}^{m} z_i^{(n)} u_i + \sum_{i=m+1}^{d} b_i u_i$$

We wish to minimize the squared error E_m between the reconstructed points and the original points:

$$E_m = \frac{1}{2}\sum_{n=1}^{N} \|x^{(n)} - \tilde{x}^{(n)}\|^2 \tag{93}$$

$$x^{(n)} - \tilde{x}^{(n)} = \sum_{m+1}^{d} (z_i^{(n)} - b_i) u_i \tag{94}$$

$$E_m = \frac{1}{2}\sum_{n=1}^{N}\sum_{i=m+1}^{d} (z_i^{(n)} - b_i)^2 \tag{95}$$

Setting $\frac{\partial E_m}{\partial b_i} = 0$:

$$b_i = \frac{1}{N}\sum_{n=1}^{N} z_i^{(n)} = u_i^T \bar{x} \tag{96}$$

$$E_m = \frac{1}{2}\sum_{n=1}^{N}\sum_{i=m+1}^{d} (u_i^T (x^{(n)} - \bar{x}))^2 \tag{97}$$

$$= \frac{N}{2}\sum_{i=m+1}^{s} u_i^T \Sigma u_i \tag{98}$$

Thus, the error is a function of the basis vectors of the discarded dimensions and the covariance matrix Σ of the data. This can be expressed as a constrained optimization problem:

$$\min_U \sum_{i=m+1}^{d} u_i^T \Sigma u_i$$
$$\text{s.t. } u_i^T u_j = \delta_{ij}$$

Taking the Lagrangian:

$$L(U, M) = \sum_{i=m+1}^{d} u_i^T \Sigma u_i - \frac{1}{2} \sum_i \sum_j \mu_{ij}(u_i^T u_j - \delta_{ij}) \tag{99}$$
$$= \frac{1}{2} \operatorname{Tr}(U^T \Sigma U) - \frac{1}{2} \operatorname{Tr}(M(U^T U - I)) \quad \text{using matrix notation} \tag{100}$$

Using $\frac{\partial}{\partial X} \operatorname{Tr}(AXBX^TC) = BX^TCA + B^TX^TA^TC^T$:

$$\frac{\partial L}{\partial U} = 0 = (\Sigma + \Sigma^T)U - U(M + M^T) \tag{101}$$
$$\Sigma U = UM \quad M \text{ and } \Sigma \text{ symmetric} \tag{102}$$

Define Ψ and Λ to be an eigenvalue decomposition of M:

$$M\Psi = \Psi\Lambda \tag{103}$$
$$\Lambda = \Psi^T M \Psi \quad \text{multiply on left by } \Psi^T \tag{104}$$
$$= \Psi^T U^T \Sigma U \Psi \quad \text{substitute from eq. 102} \tag{105}$$
$$= \tilde{U}^T \Sigma \tilde{U} \quad \text{define } \tilde{U} = U\Psi \tag{106}$$

This result means that $\tilde{U}$ and Λ form an eigenvalue decompositon of Σ, and can be used to rewrite the error E_m:

$$E_m = \frac{1}{2} \operatorname{Tr}(U^T \Sigma U) \tag{107}$$
$$= \frac{1}{2} \operatorname{Tr}(\Psi \tilde{U}^T \Sigma \tilde{U} \Psi^T) \quad \text{using def. of } \tilde{U} \tag{108}$$
$$= \frac{1}{2} \operatorname{Tr}(\tilde{U}^T \Sigma \tilde{U}) \quad \text{rotate matrices inside trace, } \Psi^T\Psi = I \tag{109}$$
$$= \frac{1}{2} \operatorname{Tr}(\Lambda) \quad \text{using eq. 106} \tag{110}$$

Thus the error is a sum of eigenvalues of Σ, and can be minimized by choosing the smallest eigenvalues.

25 Reinforcement Learning

25.1 Markov Decision Processes

A Markov Decision Process is an extension of the standard (unhidden) Markov model. Each state has a collection of actions that can be performed in that particular state. These actions serve to move the system into a new state. More formally, the MDP's state transitions can be described by the transition function $T(s, a, s')$, where a is an action moving performable during the current state s, and s' is some new state.

As the name implies, all MDPs obey the *Markov property*, which holds that the probability of finding the system in a given state is dependent only on the previous state. Thus, the system's state at any given time is determined solely by the transition function and the action taken during the previous timestep:

$$P(S_t = s' | S_{t-1} = s,\ a_t = a) = T(s, a, s')$$

Each MDP also has a reward function $R : S \mapsto \mathbb{R}$. This reward function assigns some value $R(s)$ to being in the state $s \in S$. One common way of trading off present reward against future reward is by introducing a *discount rate* γ. The discount rate is between 0 and 1, and we can use it to construct a discounted sum of future rewards:

$$\sum_{t=0}^{\infty} \gamma^t R(s_t)$$

Here, we assume that $t = 0$ is the current time. Since $0 < \gamma < 1$, greater values of t (indicating rewards farther in the future) are given smaller weight than rewards in the nearer future.

Given a Markov Decision Process we wish to find a *policy* – a mapping from states to actions. The policy function $\Pi : S \mapsto A$ selects the appropriate action $a \in A$ given the current state $s \in S$.

The consequences of actions (i.e., rewards) and the effects of policies are not always known immediately. As such, we need some mechanisms to control and adjust policy when the rewards of the current state space are uncertain. These mechanisms are collectively referred to as *reinforcement learning*.

25.2 Value Iteration

Let $V^{\Pi}(s)$ be the value function for the policy Π. This function $V^{\Pi} : S \mapsto \mathbb{R}$ maps the application of Π to some state $s \in S$ to some reward value. Assuming the system starts in state s_0, we would expect the system to have the value

$$V^{\Pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid s_0 = s, \Pi\right]$$

Since the probability of the system being in a given state $s' \in S$ is determined by the transition function $T(s, a, s')$, we can rewrite the formula above for some arbitrary state $s \in S$ as

$$V^{\Pi}(s) = R(s) + \sum_{s'} T(s, a, s') \gamma V^{\Pi}(s')$$

where $a = \Pi(s)$ is the action selected by the policy for the given state s.

Our goal here is to determine the optimal policy Π*(s). Examining the formula above, we see that $R(s)$ is unaffected by choice of policy. This makes sense because at any given state s, local reward term $R(s)$ is determined simply by virtue of the fact that the system is in state s. Thus, if we wish to find the maximum policy value function (and therefore find the optimum policy) we must find the action a that maximizes the summation term above:

$$V^{\Pi^*}(s) = R(s) + \max_a \sum_{s'} T(s, a, s') \gamma V^{\Pi^*}(s')$$

Note that this formulation assumes that the number of states is finite.

The formula above forms the basis of the *value iteration* algorithm. This operation starts with some initial policy value function guess and iteratively refines $V(s)$ until some acceptable convergence is reached:

Each pass of the value iteration maximizes $V(s)$ and assigns to $\Pi^*(s)$ the action a that maximizes $V(s)$.

25.3 Q-Learning

The example of value iteration above presumes that the transition function $T(s, a, s')$ is known. If the transition function is not known, then the function $Q(s, a)$ can be obtained through a similar process of iterative learning, the aptly-named *Q-learning*. The function $Q(s, a)$ represents the potential value for $V(s)$ produced by the action $a \in A$.

procedure VALUE ITERATION
  Initialize $V(s)$.
  repeat
    for all $s \in S$ **do**
      $R(s) \leftarrow R(s) + \max_a \sum_{s'} T(s,a,s')\gamma V(s')$
      $\Pi(s) \leftarrow \operatorname{argmax}_a \sum_{s'} T(s,a,s')\gamma V(s')$
  until converged

The correct values of $Q(s,a)$ should satisfy the following system of equations:

$$Q(s,a) = R(s) + \gamma \left[\sum_{s'} T(s,a,s') \max_{a'} Q(s',a') \right]$$

Q-learning using on online update to converge to the solution above by treating what happens at each timestep as a stochastic estimate of the sum over s':

$$Q(s,a) \leftarrow (1-\eta)Q(s,a) + \eta\gamma \left[R(s) \max_{a'} Q(s',a') \right]$$

where η is a user-selected learning parameter.

This formula is applied online at each timestep t with $s = s_t$ and $s' = s_{t+1}$. The action is selected by a user-defined function $f(s)$, which returns the appropriate policy action $\Pi(s)$ most of the time, but occasionally selects a random action to blunt the effects of sampling bias.

procedure ONE-STEP Q-LEARNING
  Initialize $\Pi(s)$ to $\operatorname{argmax}_a Q(s,a)$.
  repeat
    At timestep t
    Select an action $a_t = f(s_t)$.
    $Q(s_t,a_t) \leftarrow (1-\eta)Q(s_t,a_t) + \eta \left[R(s_t) + \gamma \max_{a'} Q(s_{t+1},a') \right]$
    $\Pi(s_t) \leftarrow \operatorname{argmax}_a Q(s_t,a)$.
  until Π converges

25.4 Temporal Difference Learning

One disadvantage of value iteration is that it can take a long time for updates to later states to propagate back to earlier states. For instance, an MDP attempting to navigate a maze would see its reward function jump once it reaches the final stage N, but it would take N iterations for the effects of that jump propagate back to stage 1.

Temporal difference learning remedies this by modifying the value for each state according to how recently it has been seen:

$$\Delta V(s) = \eta \left(R(s_t) + \gamma V(s_{t+1}) - V(s_t) \right) \sum_{k=1}^{t} (\gamma\lambda)^{t-k} I(s_k = s)$$

where $0 < \lambda \leq 1$ controls the degree to which updates are pushed back in time. This computation can be made more efficient by defining an eligibility trace:

$$e_{s,t} = \sum_{k=1}^{t} (\gamma\lambda)^{t-k} I(s_k = s)$$

such that the update is:

$$\Delta V(s) = \eta \left(R(s_t) + \gamma V(s_{t+1}) - V(s_t) \right) e_{s,t}$$

The eligibility trace is easily updated online:

$$e_{s,t} = \gamma \lambda e_{s,t-1} + I(s_t = s)$$

---

procedure TEMPORAL DIFFERENCE LEARNING
  for time t **do**
    Use $\Pi(s_t)$ to obtain a new state s_{t+1} and calculate $V(s_{t+1})$.
    $\delta_t \leftarrow R(s_t) + \gamma V(s_{t+1}) - V(s_t)$
    for s **do**
      $e_s \leftarrow \gamma \lambda e_s + I(s_t = s)$
      $V(s) \mathrel{+}= \eta \delta_t e_s$.

25.5 Function Approximation

For large state space, we estimate a function $V_w(s)$ from features of the state s to the value of s, with parameters w. Typically w are the weights of a neural network. If we define our objective function as squared error,

$$E(s_t) = \frac{1}{2}(V_w(s_t) - V(s_t))^2$$
$$\frac{\partial E(s_t)}{\partial w} = \frac{\partial E(s_t)}{\partial V_w(s_t)} \frac{\partial V_w(s_t)}{\partial w} = -\delta_t \frac{\partial V_w(s_t)}{\partial w}$$

where δ_t is the difference of our noisy estimate of the value $V(s_t)$ from the observation at the current timestep and the current output of the network $V_w(s_t)$:

$$\delta_t = -\frac{\partial E(s_t)}{\partial V_w(s_t)} = (R(s_t) + \gamma V_w(s_{t+1}) - V_w(s_t))$$

This gives us an update rule for the network weights:

$$\Delta w = \eta \delta_t \frac{\partial V_w(s)}{\partial w}$$

The gradient $\frac{\partial V_w(s)}{\partial w}$ is computed with backpropagation.

25.6 Function Approximation with Eligibility Trace

The eligibility trace of TD-learning can be applied to the weight vector of a neural network. We accumulate updates across all states that have been visited:

$$e_t = \gamma \lambda e_{t-1} + \frac{\partial V_w(s)}{\partial w}$$
$$\Delta w = \eta \delta_t e_t$$

Here e_t is a vector with the dimensionality of the weight vector, which tracks responsibility of each weight for the currect state. This derives from adding together TD updates for each state:

$$\Delta w = \eta \, \delta_t \sum_{k=1}^{t} (\gamma \lambda)^{t-k} \frac{\partial V_w(s_k)}{\partial w}$$

A The Greek alphabet

A	α	alpha
B	β	beta
Γ	γ	gamma
Δ	δ	delta
E	ϵ	epsilon
Z	ζ	zeta
H	η	eta
Θ	θ	theta
I	ι	iota
K	κ	kappa
Λ	λ	lambda
M	μ	mu
N	ν	nu
Ξ	ξ	xi
O	o	omicron
Π	π	pi
P	ρ	rho
Σ	σ	sigma
T	τ	tau
Υ	υ	upsilon
Φ	ϕ	phi
X	χ	chi
Ψ	ψ	psi
Ω	ω	omega