HMM 学习最佳范例

作者：52nlp(52nlpcn@gmail.com)
我爱自然语言处理：www.52nlp.cn

一、介绍（Introduction）

我们通常都习惯寻找一个事物在一段时间里的变化模式（规律）。这些模式发生在很多领域，比如计算机中的指令序列，句子中的词语顺序和口语单词中的音素序列等等，事实上任何领域中的一系列事件都有可能产生有用的模式。

考虑一个简单的例子，有人试图通过一片海藻推断天气——民间传说告诉我们‘湿透的’海藻意味着潮湿阴雨，而‘干燥的’海藻则意味着阳光灿烂。如果它处于一个中间状态（‘有湿气’），我们就无法确定天气如何。然而，天气的状态并没有受限于海藻的状态，所以我们可以在观察的基础上预测天气是雨天或晴天的可 能性。另一个有用的线索是前一天的天气状态（或者，至少是它的可能状态）——通过综合昨天的天气及相应观察到的海藻状态，我们有可能更好的预测今天的天 气。

这是本教程中我们将考虑的一个典型的系统类型。

首先，我们将介绍产生概率模式的系统，如晴天及雨天间的天气波动。

然后，我们将会看到这样一个系统，我们希望预测的状态并不是观察到的——其底层系统是隐藏的。在上面的例子中，观察到的序列将是海藻而隐藏的系统将是实际的天气。

最后，我们会利用已经建立的模型解决一些实际的问题。对于上述例子，我们想知道：

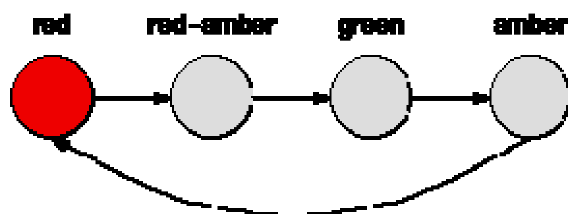
1. 给出一个星期每天的海藻观察状态，之后的天气将会是什么？

2. 给定一个海藻的观察状态序列，预测一下此时是冬季还是夏季？直观地，如果一段时间内海藻都是干燥的，那么这段时间很可能是夏季，反之，如果一段时间内海藻都是潮湿的，那么这段时间可能是冬季。

二、生成模式（Generating Patterns）

1、确定性模式（Deterministic Patterns）

考虑一套交通信号灯，灯的颜色变化序列依次是红色-红色/黄色-绿色-黄色-红色。这个序列可以作为一个状态机器，交通信号灯的不同状态都紧跟着上一个状态。



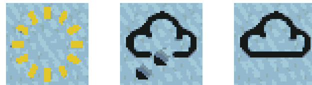
注意每一个状态都是唯一的依赖于前一个状态，所以，如果交通灯为绿色，那么下一个颜色状态将始终是黄色——也就是说，该系统是确定性的。确定性系统相对比较容易理解和分析，因为状态间的转移是完全已知的。

2、非确定性模式（Non-deterministic patterns）

为了使天气那个例子更符合实际，加入第三个状态——多云。与交通信号灯例子不同，我们并不期望这三个天气状态之间的变化是确定性的，但是我们依然希望对这个系统建模以便生成一个天气变化模式（规律）。

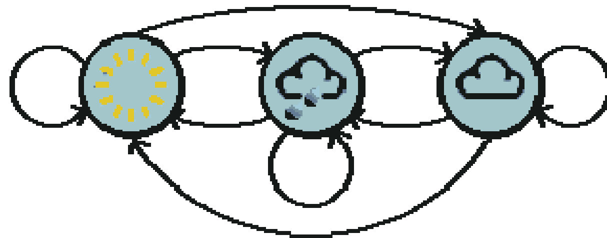
一种做法是假设模型的当前状态仅仅依赖于前面的几个状态，这被称为马尔科夫假设，它极大地简化了问题。显然，这可能是一种粗糙的假设，并且因此可能将一些非常重要的信息丢失。

当考虑天气问题时，马尔科夫假设假定今天的天气只能通过过去几天已知的天气情况进行预测——而对于其他因素，譬如风力、气压等则没有考虑。在这个例子 以及其他相似的例子中，这样的假设显然是不现实的。然而，由于这样经过简化的系统可以用来分析，我们常常接受这样的知识假设，虽然它产生的某些信息不完全 准确。



一个马尔科夫过程是状态间的转移仅依赖于前 n 个状态的过程。这个过程被称之为 n 阶马尔科夫模型，其中 n 是影响下一个状态选择的（前）n 个状态。最简单 的马尔科夫过程是一阶模型，它的状态选择仅与前一个状态有关。这里要注意它与确定性系统并不相同，因为下一个状态的选择由相应的概率决定，并不是确定性 的。

下图是天气例子中状态间所有可能的一阶状态转移情况：



对于有 M 个状态的一阶马尔科夫模型，共有 M^2 个状态转移，因为任何一个状态都有可能是所有状态的下一个转移状态。每一个状态转移都有一个概率值，称为状态转移概率——这是从一个状态转移到另一个状态的概率。所有的 M^2 个概率可以用一个状态转移矩阵表示。注意这些概率并不随时间变化而不同——这是一个非常重要（但常常不符合实际）的假设。

下面的状态转移矩阵显示的是天气例子中可能的状态转移概率：

$$
\begin{array}{ccc}
 & & \quad\quad Today \\
Yesterday & \begin{array}{c} \\ sun \\ cloud \\ rain \end{array} & \left[\begin{array}{ccc} sun & cloud & rain \\ 0.50 & 0.375 & 0.125 \\ 0.25 & 0.125 & 0.625 \\ 0.25 & 0.375 & 0.375 \end{array}\right]
\end{array}
$$

-也就是说，如果昨天是晴天，那么今天是晴天的概率为 0.5，是多云的概

率为 0.375。注意，每一行的概率之和为 1。

要初始化这样一个系统，我们需要确定起始日天气的（或可能的）情况，定义其为一个初始概率向量，称为π向量。

$$\begin{pmatrix} \text{Sun} & \text{Cloud} & \text{Rain} \\ 1.0 & 0.0 & 0.0 \end{pmatrix}$$

-也就是说，第一天为晴天的概率为 1。

现在我们定义一个一阶马尔科夫过程如下：

状态：三个状态——晴天，多云，雨天。

π向量：定义系统初始化时每一个状态的概率。

状态转移矩阵：给定前一天天气情况下的当前天气概率。

任何一个可以用这种方式描述的系统都是一个马尔科夫过程。

3、总结

我们尝试识别时间变化中的模式，并且为了达到这个目我们试图对这个过程建模以便产生这样的模式。我们使用了离散时间点、离散状态以及做了马尔科夫假设。在采用了这些假设之后，系统产生了这个被描述为马尔科夫过程的模式，它包含了一个π向量（初始概率）和一个状态转移矩阵。关于假设，重要的一点是状态转移矩阵并不随时间的改变而改变——这个矩阵在整个系统的生命周期中是固定不变的。

三、隐藏模式（Hidden Patterns）

1、马尔科夫过程的局限性

在某些情况下，我们希望找到的模式用马尔科夫过程描述还显得不充分。回顾一下天气那个例子，一个隐士也许不能够直接获取到天气的观察情况，但是他有一些水藻。民间传说告诉我们水藻的状态与天气状态有一定的概率关系——天气和水藻的状态是紧密相关的。在这个例子中我们有两组状态，观察的状态（水藻的状态）和隐藏的状态（天气的状态）。我们希望为隐士设计一种算法，在不能够直接观察天气的情况下，通过水藻和马尔科夫假设来预测天气。

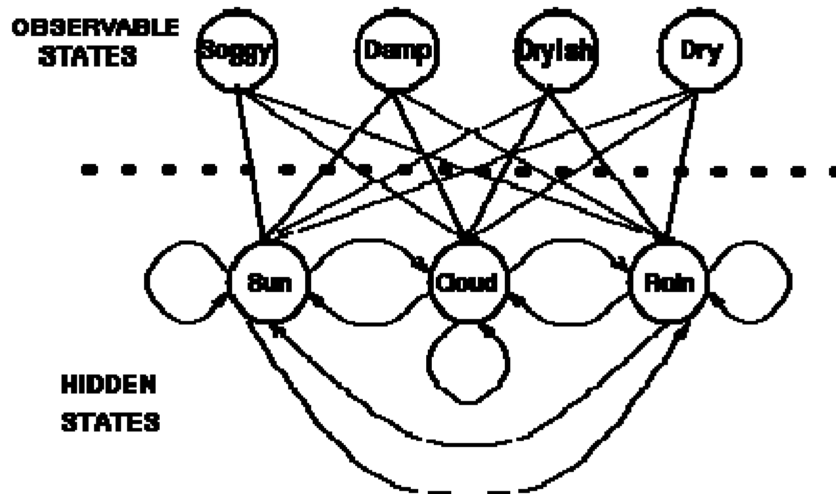
一个更实际的问题是语音识别，我们听到的声音是来自于声带、喉咙大小、舌头位置以及其他一些东西的组合结果。所有这些因素相互作用产生一个单词的声音，一套语音识别系统检测的声音就是来自于个人发音时身体内部物理变化所引起的不断改变的声音。

一些语音识别装置工作的原理是将内部的语音产出看作是隐藏的状态，而将声音结果作为一系列观察的状态，这些由语音过程生成并且最好的近似了实际（隐 藏）的状态。在这两个例子中，需要着重指出的是，隐藏状态的数目与观察状态的数目可以是不同的。一个包含三个状态的天气系统（晴天、多云、雨天）中，可以 观察到 4 个等级的海藻湿润情况（干、稍干、潮湿、湿润）；纯粹的语音可以由 80 个音素描述，而身体的发音系统会产生出不同数目的声音，或者比 80 多，或者 比 80 少。

在这种情况下，观察到的状态序列与隐藏过程有一定的概率关系。我们使用隐马尔科夫模型对这样的过程建模，这个模型包含了一个底层隐藏的随时间改变的马尔科夫过程，以及一个与隐藏状态某种程度相关的可观察到的状态集合。

2、隐马尔科夫模型（Hidden Markov Models）

下图显示的是天气例子中的隐藏状态和观察状态。假设隐藏状态（实际的天气）由一个简单的一阶马尔科夫过程描述，那么它们之间都相互连接。



隐藏状态和观察状态之间的连接表示：在给定的马尔科夫过程中，一个特定的隐藏状态生成特定的观察状态的概率。这很清晰的表示了‘进入’一个观察状态的 所有概率之和为 1，在上面这个例子中就是 Pr(Obs|Sun), Pr(Obs|Cloud) 及 Pr(Obs|Rain)之和。（对这句话我有点疑惑？）

除了定义了马尔科夫过程的概率关系，我们还有另一个矩阵，定义为混淆矩阵（confusion matrix），它包含了给定一个隐藏状态后得到的观察状态的概率。对于天气例子，混淆矩阵是：

Seaweed

weather	Dry	Dryish	Damp	Soggy
Sun	0.60	0.20	0.15	0.05
Cloud	0.25	0.25	0.25	0.25
Rain	0.05	0.10	0.35	0.50

注意矩阵的每一行之和是 1。

3、总结（Summary）

我们已经看到在一些过程中一个观察序列与一个底层马尔科夫过程是概率相关的。在这些例子中，观察状态的数目可以和隐藏状态的数码不同。

我们使用一个隐马尔科夫模型（HMM）对这些例子建模。这个模型包含两组状态集合和三组概率集合：

* 隐藏状态：一个系统的（真实）状态，可以由一个马尔科夫过程进行描述（例如，天气）。

* 观察状态：在这个过程中‘可视’的状态（例如，海藻的湿度）。

* π向量：包含了（隐）模型在时间 t=1 时一个特殊的隐藏状态的概率（初始概率）。

* 状态转移矩阵：包含了一个隐藏状态到另一个隐藏状态的概率

* 混淆矩阵：包含了给定隐马尔科夫模型的某一个特殊的隐藏状态，观察到的某个观察状态的概率。

因此一个隐马尔科夫模型是在一个标准的马尔科夫过程中引入一组观察状态，以及其与隐藏状态间的一些概率关系。

四、隐马尔科夫模型（Hidden Markov Models）

1、定义（Definition of a hidden Markov model）

一个隐马尔科夫模型是一个三元组（π, A, B）。

$\Pi = (\pi_i)$ ：初始化概率向量；

$A = (a_{ij})$：状态转移矩阵；$Pr(x_{i_t} | x_{j_{t-1}})$

$B = (b_{ij})$：混淆矩阵；$Pr(y_i | x_j)$

在状态转移矩阵及混淆矩阵中的每一个概率都是时间无关的——也就是说，当系统演化时这些矩阵并不随时间改变。实际上，这是马尔科夫模型关于真实世界最不现实的一个假设。

2、应用（Uses associated with HMMs）

一旦一个系统可以作为 HMM 被描述，就可以用来解决三个基本问题。其中前两个是模式识别的问题：给定 HMM 求一个观察序列的概率（评估）；搜索最有可能生成一个观察序列的隐藏状态训练（解码）。第三个问题是给定观察序列生成一个 HMM（学习）。

a) 评估（Evaluation）

考虑这样的问题，我们有一些描述不同系统的隐马尔科夫模型（也就是一些（ π,A,B)三元组的集合）及一个观察序列。我们想知道哪一个 HMM 最有可能产生了这个给定的观察序列。例如，对于海藻来说，我们也许会有一个“夏季”模型和一个“冬季”模型，因为不同季节之间的情况是不同的——我们也许想根据海藻湿度的观察序列来确定当前的季节。

我们使用前向算法（forward algorithm）来计算给定隐马尔科夫模型（HMM）后的一个观察序列的概率，并因此选择最合适的隐马尔科夫模型(HMM)。

在语音识别中这种类型的问题发生在当一大堆数目的马尔科夫模型被使用，并且每一个模型都对一个特殊的单词进行建模时。一个观察序列从一个发音单词中形成，并且通过寻找对于此观察序列最有可能的隐马尔科夫模型（HMM）识别这个单词。

b) 解码（ Decoding）

给定观察序列搜索最可能的隐藏状态序列。

另一个相关问题，也是最感兴趣的一个，就是搜索生成输出序列的隐藏状态

序列。在许多情况下我们对于模型中的隐藏状态更感兴趣，因为它们代表了一些更有价值的东西，而这些东西通常不能直接观察到。

考虑海藻和天气这个例子，一个盲人隐士只能感觉到海藻的状态，但是他更想知道天气的情况，天气状态在这里就是隐藏状态。

我们使用 Viterbi 算法（Viterbi algorithm）确定（搜索）已知观察序列及 HMM 下最可能的隐藏状态序列。

Viterbi 算法（Viterbi algorithm）的另一广泛应用是自然语言处理中的词性标注。在词性标注中，句子中的单词是观察状态，词性（语法类别）是隐藏状态（注意对于许多单 词，如 wind，fish 拥有不止一个词性）。对于每句话中的单词，通过搜索其最可能的隐藏状态，我们就可以在给定的上下文中找到每个单词最可能的词性标 注。

C）学习（Learning）

根据观察序列生成隐马尔科夫模型。

第三个问题，也是与 HMM 相关的问题中最难的，根据一个观察序列（来自于已知的集合），以及与其有关的一个隐藏状态集，估计一个最合适的隐马尔科夫模型（HMM），也就是确定对已知序列描述的最合适的（π,A,B）三元组。

当矩阵 A 和 B 不能够直接被（估计）测量时，前向-后向算法（forward-backward algorithm）被用来进行学习（参数估计），这也是实际应用中常见的情况。

3、总结（Summary）

由一个向量和两个矩阵(π,A,B)描述的隐马尔科夫模型对于实际系统有着巨大的价值，虽然经常只是一种近似，但它们却是经得起分析的。隐马尔科夫模型通常解决的问题包括：

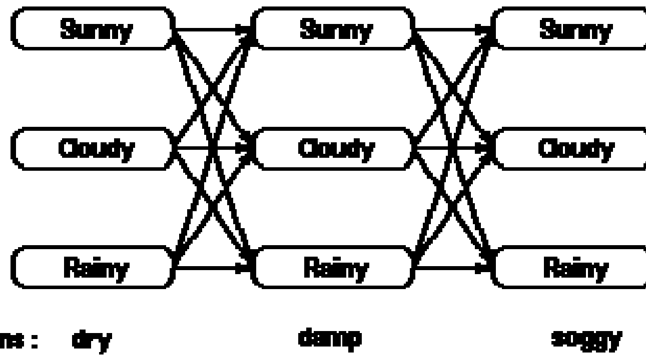
1. 对于一个观察序列匹配最可能的系统——评估，使用前向算法（forward algorithm）解决；
2. 对于已生成的一个观察序列，确定最可能的隐藏状态序列——解码，使用 Viterbi 算法（Viterbi algorithm）解决；
3. 对于已生成的观察序列，决定最可能的模型参数——学习，使用前向-后向算法（forward-backward algorithm）解决。

五、前向算法（Forward Algorithm）

计算观察序列的概率（Finding the probability of an observed sequence）

1.穷举搜索（ Exhaustive search for solution）

给定隐马尔科夫模型，也就是在模型参数（π, A, B)已知的情况下，我们想找到观察序列的概率。还是考虑天气这个例子，我们有一个用来描述天气及与它密切相关的海藻湿度状态的隐马尔科夫模型(HMM)， 另外我们还有一个海藻的湿度状态观察序列。假设连续 3 天海藻湿度的观察结果是（干燥、湿润、湿透）——而这三天每一天都可能是晴天、多云或下雨，对于观察 序列以及隐藏的状态，可以将其视为网格：



网格中的每一列都显示了可能的的天气状态，并且每一列中的每个状态都与相邻列中的每一个状态相连。而其状态间的转移都由状态转移矩阵提供一个概率。在每一列下面都是某个时间点上的观察状态，给定任一个隐藏状态所得到的观察状态的概率由混淆矩阵提供。

可以看出，一种计算观察序列概率的方法是找到每一个可能的隐藏状态，并且将这些隐藏状态下的观察序列概率相加。对于上面那个（天气）例子，将有 3^3 = 27 种不同的天气序列可能性，因此，观察序列的概率是：

Pr(dry,damp,soggy | HMM) = Pr(dry,damp,soggy | sunny,sunny,sunny) + Pr(dry,damp,soggy | sunny,sunny ,cloudy) + Pr(dry,damp,soggy | sunny,sunny ,rainy) + ... Pr(dry,damp,soggy | rainy,rainy ,rainy)

用这种方式计算观察序列概率极为昂贵，特别对于大的模型或较长的序列，因此我们可以利用这些概率的时间不变性来减少问题的复杂度。

2.使用递归降低问题复杂度

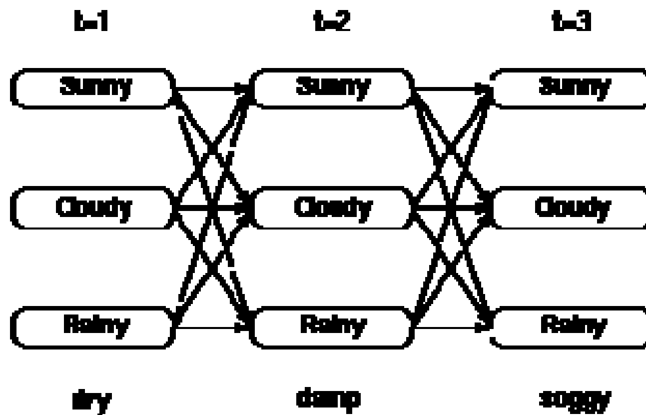
给定一个隐马尔科夫模型（HMM），我们将考虑递归地计算一个观察序列的概率。我们首先定义局部概率（partial probability）,它是到达网格中的某个中间状态时的概率。然后，我们将介绍如何在 t=1 和 t=n(>1)时计算这些局部概率。

假设一个 T-长观察序列是：

$$(Y_{k_1}, Y_{k_2}, \ldots\ldots, Y_{k_T})$$

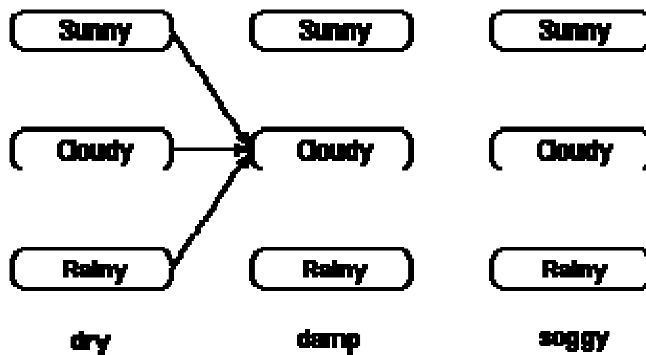
2a.局部概率(α’s)

考虑下面这个网格，它显示的是天气状态及对于观察序列干燥，湿润及湿透的一阶状态转移情况：



我们可以将计算到达网格中某个中间状态的概率作为所有到达这个状态的可能路径的概率求和问题。

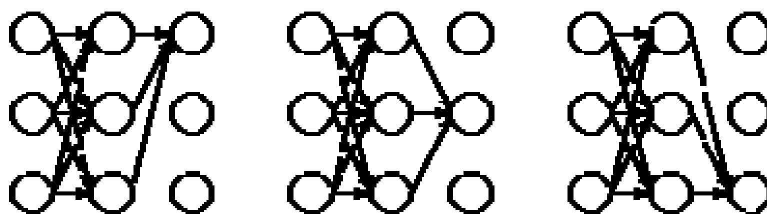
例如，t=2 时位于“多云”状态的局部概率通过如下路径计算得出：



我们定义 t 时刻位于状态 j 的局部概率为 at(j)——这个局部概率计算如下：

αt (j)= Pr(观察状态 | 隐藏状态 j) x Pr(t 时刻所有指向 j 状态的路径)

对于最后的观察状态，其局部概率包括了通过所有可能的路径到达这些状态的概率——例如，对于上述网格，最终的局部概率通过如下路径计算得出：



由此可见，对于这些最终局部概率求和等价于对于网格中所有可能的路径概率求和，也就求出了给定隐马尔科夫模型(HMM)后的观察序列概率。

第 3 节给出了一个计算这些概率的动态示例。

2b.计算 t=1 时的局部概率α's

我们按如下公式计算局部概率：

αt (j)= Pr(观察状态 | 隐藏状态 j) x Pr(t 时刻所有指向 j 状态的路径)

特别当 t=1 时，没有任何指向当前状态的路径。故 t=1 时位于当前状态的概

率是初始概率，即 Pr(state|t=1)=P(state)，因此，t=1 时的局部概率等于当前状态的初始概率乘以相关的观察概率：

$$\alpha_1(j) = \pi(j) . b_{jk_1}$$

所以初始时刻状态 j 的局部概率依赖于此状态的初始概率及相应时刻我们所见的观察概率。

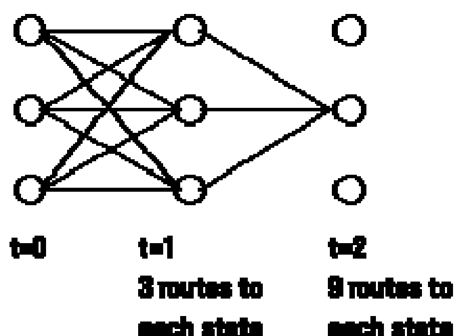
2c. 计算 t>1 时的局部概率 α’s

我们再次回顾局部概率的计算公式如下：

αt (j)= Pr(观察状态 | 隐藏状态 j) x Pr(t 时刻所有指向 j 状态的路径)

我们可以假设（递归地），乘号左边项“Pr(观察状态 | 隐藏状态 j)”已经有了，现在考虑其右边项“Pr(t 时刻所有指向 j 状态的路径）”。

为了计算到达某个状态的所有路径的概率，我们可以计算到达此状态的每条路径的概率并对它们求和，例如：



计算α所需要的路径数目随着观察序列的增加而指数级递增，但是 t-1 时刻α’s 给出了所有到达此状态的前一路径概率，因此，我们可以通过 t-1 时刻的局部概率定义 t 时刻的α’s，即：

$$\alpha_{t+1}(j) = b_{jk_{t+1}} \sum_{i=1}^{n} \alpha_t(i) a_{ij}$$

故我们所计算的这个概率等于相应的观察概率（亦即，t+1 时在状态 j 所观察到的符号的概率）与该时刻到达此状态的概率总和——这来自于上一步每一个局部概率的计算结果与相应的状态转移概率乘积后再相加——的乘积。

注意我们已经有了一个仅利用 t 时刻局部概率计算 t+1 时刻局部概率的表达式。

现在我们就可以递归地计算给定隐马尔科夫模型(HMM)后一个观察序列的概率了——即通过 t=1 时刻的局部概率α’s 计算 t=2 时刻的α’s，通过 t=2 时刻的α’s 计算 t=3 时刻的α’s 等等直到 t=T。给定隐马尔科夫模型(HMM)的观察序列的概率就等于 t=T 时刻的局部概率之和。

2d. 降低计算复杂度

我们可以比较通过穷举搜索（评估）和通过递归前向算法计算观察序列概率的时间复杂度。

我们有一个长度为 T 的观察序列 O 以及一个含有 n 个隐藏状态的隐马尔科夫

模型 l=(π,A,B)。

穷举搜索将包括计算所有可能的序列：

$$X_i = (X_{i_1}, X_{i_2}, ..., X_{i_T})$$

公式

$$\sum_X \pi(i_1) b_{i_1 k_1} \prod_{j=2}^{T} a_{i_{j-1} i_j} b_{i_j k_j}$$

对我们所观察到的概率求和——注意其复杂度与 T 成指数级关系。相反的，使用前向算法我们可以利用上一步计算的信息，相应地，其时间复杂度与 T 成线性关系。

注：穷举搜索的时间复杂度是 $2TN^T$，前向算法的时间复杂度是 N^2T，其中 T 指的是观察序列长度，N 指的是隐藏状态数目。

3.总结

我们的目标是计算给定隐马尔科夫模型 HMM 下的观察序列的概率——Pr(observations |λ)。

我们首先通过计算局部概率（α’s）降低计算整个概率的复杂度，局部概率表示的是 t 时刻到达某个状态 s 的概率。

t=1 时，可以利用初始概率(来自于 P 向量）和观察概率 Pr(observation|state)（来自于混淆矩阵）计算局部概率；而 t>1 时的局部概率可以利用 t-时的局部概率计算。

因此，这个问题是递归定义的，观察序列的概率就是通过依次计算 t=1,2,…,T 时的局部概率，并且对于 t=T 时所有局部概率α’s 相加得到的。

注意，用这种方式计算观察序列概率的时间复杂度远远小于计算所有序列的概率并对其相加（穷举搜索）的时间复杂度。

我们使用前向算法计算 T 长观察序列的概率：

$$Y^{(k)} = y_{k_1}, ..., y_{k_T}$$

其中 y 的每一个是观察集合之一。局部（中间）概率（α’s)是递归计算的，首先通过计算 t=1 时刻所有状态的局部概率α：

$$\alpha_1(j) = \pi(j).b_{jk_1}$$

然后在每个时间点，t=2，… ，T 时，对于每个状态的局部概率，由下式计算局部概率α：

$$\alpha_{t+1}(j) = \sum_{i=1}^{n} \left(\alpha_t(i) a_{ij}\right) b_{jk_t}$$

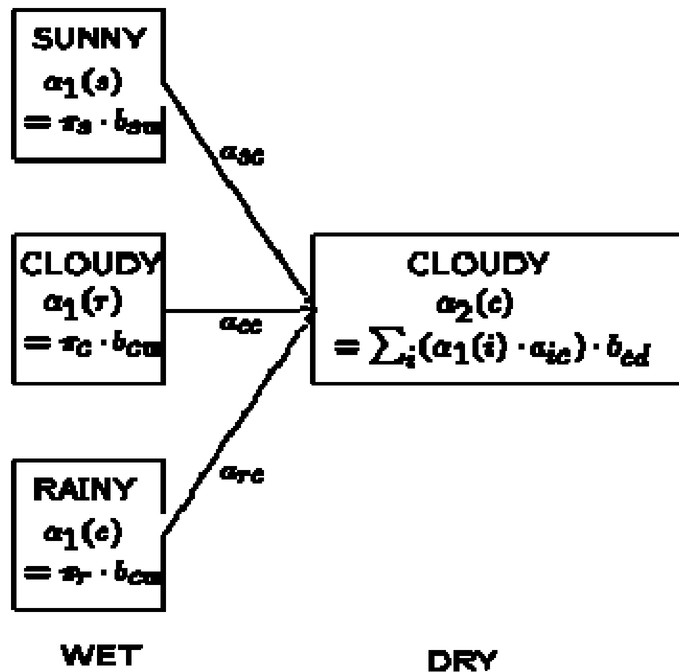
也就是当前状态相应的观察概率与所有到达该状态的路径概率之积，其递归地利用了上一个时间点已经计算好的一些值。

最后，给定 HMM，λ，观察序列的概率等于 T 时刻所有局部概率之和：

$$Pr\left(Y^{(k)}\right) = \sum_{j=1}^{n} \alpha_T(j)$$

再重复说明一下，每一个局部概率（t > 2 时）都由前一时刻的结果计算得出。

对于“天气”那个例子，下面的图表显示了 t = 2 为状态为多云时局部概率 α的计算过程。这是相应的观察概率 b 与前一时刻的局部概率与状态转移概率 a 相乘后的总和再求积的结果：



总结（Summary）

我们使用前向算法来计算给定隐马尔科夫模型（HMM）后的一个观察序列的概率。它在计算中利用递归避免对网格所有路径进行穷举计算。

给定这种算法，可以直接用来确定对于已知的一个观察序列，在一些隐马尔科夫模型（HMMs）中哪一个 HMM 最好的描述了它——先用前向算法评估每一个（HMM），再选取其中概率最高的一个。

首先需要说明的是，本节不是这个系列的翻译，而是作为前向算法这一章的补充，希望能从实践的角度来说明前向算法。除了用程序来解读hmm的前向算法外，还希望将原文所举例子的问题拿出来和大家探讨。

文中所举的程序来自于UMDHMM这个C语言版本的HMM工具包，具体见《几种不同程序语言的HMM版本》。先说明一下UMDHMM这个包的基本情况，在linux环境下，进入umdhmm-v1.02 目录，“make all”之后会产生 4 个可执行文件，分别是：

genseq: 利用一个给定的隐马尔科夫模型产生一个符号序列（Generates a symbol sequence using the specified model sequence using the specified model）

testfor: 利用前向算法计算log Prob(观察序列| HMM模型)（Computes log

Prob(observation|model) using the Forward algorithm.）

testvit：对于给定的观察符号序列及HMM，利用Viterbi 算法生成最可能的隐藏状态序列（Generates the most like state sequence for a given symbol sequence, given the HMM, using Viterbi）

esthmm：对于给定的观察符号序列，利用BaumWelch算法学习隐马尔科夫模型HMM（Estimates the HMM from a given symbol sequence using BaumWelch）。

这些可执行文件需要读入有固定格式的HMM文件及观察符号序列文件，格式要求及举例如下：

HMM 文件格式：

```
M= number of symbols
N= number of states
A:
a11 a12 … a1N
a21 a22 … a2N
. . . .
 . . . .
 . . . .
aN1 aN2 … aNN
B:
b11 b12 … b1M
b21 b22 … b2M
. . . .
. . . .
 . . . .
bN1 bN2 … bNM
pi:
pi1 pi2 … piN
```

HMM 文件举例：

```
M= 2
N= 3
A:
0.333 0.333 0.333
0.333 0.333 0.333
0.333 0.333 0.333
B:
0.5 0.5
0.75 0.25
0.25 0.75
pi:
```

```
        0.333 0.333 0.333
```

观察序列文件格式：

```
        T=seqence length
        o1 o2 o3 . . . oT
```

观察序列文件举例：

```
        T= 10
        1 1 1 1 2 1 2 2 2 2
```

对于前向算法的测试程序 testfor 来说，运行：

testfor model.hmm（HMM 文件） obs.seq（观察序列文件）

就可以得到观察序列的概率结果的对数值，这里我们在 testfor.c 的第 58 行对数结果的输出下再加一行输出：

fprintf(stdout, “prob(O| model) = %fn”, proba);

就可以输出运用前向算法计算观察序列所得到的概率值。至此，所有的准备工作已结束，接下来，我们将进入具体的程序解读。

首先，需要定义 HMM 的数据结构，也就是 HMM 的五个基本要素，在 UMDHMM 中是如下定义的（在 hmm.h 中）：

```
typedef struct
{
int N; /* 隐藏状态数目;Q={1,2,…,N} */
int M; /* 观察符号数目; V={1,2,…,M}*/
double **A; /* 状态转移矩阵 A[1..N][1..N]. a[i][j] 是从 t 时刻状态 i 到
t+1 时刻状态 j 的转移概率 */
double **B; /* 混淆矩阵 B[1..N][1..M]. b[j][k]在状态 j 时观察到符合 k 的
概率。*/
double *pi; /* 初始向量 pi[1..N]，pi[i] 是初始状态概率分布 */
} HMM;
```

前向算法程序示例如下（在 forward.c 中）：

```
/*
  函数参数说明：
  *phmm：已知的 HMM 模型；T：观察符号序列长度；
  *O：观察序列；**alpha：局部概率；*pprob：最终的观察概率
*/
void Forward(HMM *phmm, int T, int *O, double **alpha, double *pprob)
{
    int i, j;     /* 状态索引 */
```

```
    int t;      /* 时间索引 */
    double sum; /*求局部概率时的中间值 */

    /* 1. 初始化：计算 t=1 时刻所有状态的局部概率α：  */
    for (i = 1; i <= phmm->N; i++)
        alpha[1][i] = phmm->pi[i]* phmm->B[i][O[1]];

    /* 2. 归纳：递归计算每个时间点，t=2，… ，T 时的局部概率 */
    for (t = 1; t < T; t++)
    {
        for (j = 1; j <= phmm->N; j++)
        {
            sum = 0.0;
            for (i = 1; i <= phmm->N; i++)
                sum += alpha[t][i]* (phmm->A[i][j]);
            alpha[t+1][j] = sum*(phmm->B[j][O[t+1]]);
        }
    }

    /* 3. 终止：观察序列的概率等于 T 时刻所有局部概率之和*/
    *pprob = 0.0;
    for (i = 1; i <= phmm->N; i++)
        *pprob += alpha[T][i];
}
```

下一节我将用这个程序来验证英文原文中所举前向算法演示例子的问题。

在HMM这个翻译系列的原文中，作者举了一个前向算法的交互例子，这也是这个系列中比较出彩的地方，但是，在具体运行这个例子的时候，却发现其似乎有点问题。

先说一下如何使用这个交互例子， 运行时需要浏览器支持java，我用的是firefox。首先在Set按钮前面的对话框里上观察序列，如“Dry,Damp, Soggy”或“Dry Damp Soggy”，观察符号间用逗号或空格隔开；然后再点击Set按钮，这样就初始化了观察矩阵；如果想得到一个总的结果，即Pr(观察序列|隐马尔科夫模 型)，就点旁边的Run按钮；如果想一步一步观察计算过程，即每个节点的局部概率，就单击旁边的Step按钮。

原文交互例子（即天气这个例子）中所定义的已知隐马尔科夫模型如下：

1、隐藏状态 (天气)：Sunny，Cloudy，Rainy；

2、观察状态（海藻湿度）：Dry，Dryish，Damp，Soggy；

3、初始状态概率： Sunny（0.63）， Cloudy（0.17）， Rainy（0.20）；

4、状态转移矩阵：

```
                        weather today
                        Sunny Cloudy Rainy
   weather   Sunny 0.500 0.375 0.125
```

```
    yesterday Cloudy 0.250 0.125 0.625
                Rainy   0.250 0.375 0.375
```

5、混淆矩阵：

```
                     observed states
                     Dry Dryish Damp Soggy
            Sunny 0.60 0.20 0.15 0.05
hidden   Cloudy 0.25 0.25 0.25 0.25
states    Rainy 0.05 0.10 0.35 0.50
```

为了 UMDHMM 也能运行这个例子，我们将上述天气例子中的隐马尔科夫模型转化为如下的 UMDHMM 可读的 HMM 文件 weather.hmm：

```
M= 4
N= 3
A:
0.500 0.375 0.125
0.250 0.125 0.625
0.250 0.375 0.375
B:
0.60 0.20 0.15 0.05
0.25 0.25 0.25 0.25
0.05 0.10 0.35 0.50
pi:
0.63 0.17 0.20
```

在运行例子之前，如果读者也想观察每一步的运算结果，可以将 umdhmm-v1.02 目录下 forward.c 中的 void Forward(…)函数替换如下：

```
void Forward(HMM *phmm, int T, int *O, double **alpha, double *pprob)
{
    int i, j; /* state indices */
    int t; /* time index */
    double sum; /* partial sum */

    /* 1. Initialization */
    for (i = 1; i <= phmm->N; i++)
    {
        alpha[1][i] = phmm->pi[i]* phmm->B[i][O[1]];
        printf( “a[1][%d] = pi[%d] * b[%d][%d] = %f * %f = %f\n”,i, i,
i, O[i], phmm->pi[i], phmm->B[i][O[1]], alpha[1][i] );
    }

    /* 2. Induction */
```

```
    for (t = 1; t < T; t++)
    {
        for (j = 1; j <= phmm->N; j++)
        {
            sum = 0.0;
            for (i = 1; i <= phmm->N; i++)
            {
                sum += alpha[t][i]* (phmm->A[i][j]);
                printf( "a[%d][%d] * A[%d][%d] = %f * %f = %f\n", t,
i, i, j, alpha[t][i], phmm->A[i][j], alpha[t][i]* (phmm->A[i][j]));
                printf( "sum = %f\n", sum );
            }
            alpha[t+1][j] = sum*(phmm->B[j][O[t+1]]);
            printf( "a[%d][%d] = sum * b[%d][%d]] = %f * %f = %f\n",t+1,
j, j, O[t+1], sum, phmm->B[j][O[t+1]], alpha[t+1][j] );
        }
    }

    /* 3. Termination */
    *pprob = 0.0;
    for (i = 1; i <= phmm->N; i++)
    {
        *pprob += alpha[T][i];
        printf( "alpha[%d][%d] = %f\n", T, i, alpha[T][i] );
        printf( "pprob = %f\n", *pprob );
    }
}
```

替换完毕之后，重新“make clean”，“make all”，这样新的 testfor 可执行程序就可以输出前向算法每一步的计算结果。

现在我们就用 testfor 来运行原文中默认给出的观察序列“Dry,Damp,Soggy”，其所对应的 UMDHMM 可读的观察序列文件 test1.seq：

```
T=3
1 3 4
```

好了，一切准备工作就绪，现在就输入如下命令：

```
testfor weather.hmm test1.seq > result1
```

result1 就包含了所有的结果细节：

```
Forward without scaling
a[1][1] = pi[1] * b[1][1] = 0.630000 * 0.600000 = 0.378000
a[1][2] = pi[2] * b[2][3] = 0.170000 * 0.250000 = 0.042500
a[1][3] = pi[3] * b[3][4] = 0.200000 * 0.050000 = 0.010000
```

…
pprob = 0.026901
log prob(O| model) = -3.615577E+00
prob(O| model) = 0.026901
…

黑体部分是最终的观察序列的概率结果，即本例中的 Pr(观察序列|HMM) = 0.026901。

但是，在原文中点 Run 按钮后，结果却是：Probability of this model = 0.027386915。

这其中的差别到底在哪里？我们来仔细观察一下中间运行过程：

在初始化亦 t=1 时刻的局部概率计算两个是一致的，没有问题。但是，t=2 时，在隐藏状态“Sunny”的局部概率是不一致的。英文原文给出的例子的运行结果是：

Alpha = (((0.37800002*0.5) + (0.0425*0.375) + (0.010000001*0.125)) * 0.15) = 0.03092813

而 UMDHMM 给出的结果是：

```
a[1][1] * A[1][1] = 0.378000 * 0.500000 = 0.189000
sum = 0.189000
a[1][2] * A[2][1] = 0.042500 * 0.250000 = 0.010625
sum = 0.199625
a[1][3] * A[3][1] = 0.010000 * 0.250000 = 0.002500
sum = 0.202125
a[2][1] = sum * b[1][3]] = 0.202125 * 0.150000 = 0.030319
```

区别就在于状态转移概率的选择上，原文选择的是状态转移矩阵中的第一行，而 UMDHMM 选择的则是状态转移矩阵中的第一列。如果从原文给出的状态转移 矩阵来看，第一行代表的是从前一时刻的状态“Sunny”分别到当前时刻的状态“Sunny”，“Cloudy”，“Rainy”的概率；而第一列代表的 是从前一时刻的状态“Sunny”，“Cloudy”，“Rainy”分别到当前时刻状态“Sunny”的概率。这样看来似乎原文的计算过程有误，读者不 妨多试几个例子看看，前向算法这一章就到此为止了。

六、维特比算法（Viterbi Algorithm）

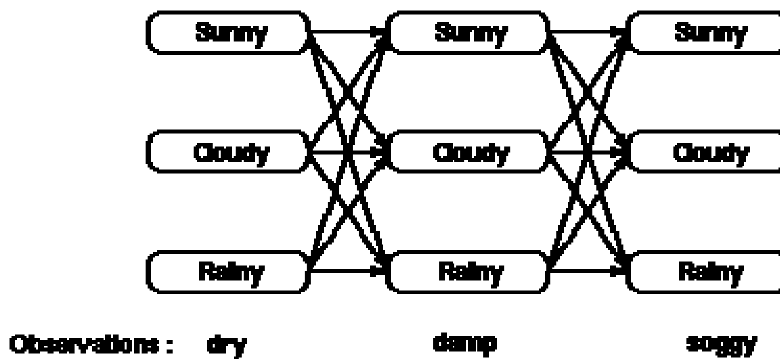
寻找最可能的隐藏状态序列(Finding most probable sequence of hidden states)

对于一个特殊的隐马尔科夫模型(HMM)及一个相应的观察序列，我们常常希望能找到生成此序列最可能的隐藏状态序列。

1.穷举搜索

我们使用下面这张网格图片来形象化的说明隐藏状态和观察状态之间的关

系：



我们可以通过列出所有可能的隐藏状态序列并且计算对于每个组合相应的观察序列的概率来找到最可能的隐藏状态序列。最可能的隐藏状态序列是使下面这个概率最大的组合：

Pr（观察序列|隐藏状态的组合）

例如，对于网格中所显示的观察序列，最可能的隐藏状态序列是下面这些概率中最大概率所对应的那个隐藏状态序列：

Pr(dry,damp,soggy | sunny,sunny,sunny), Pr(dry,damp,soggy | sunny,sunny,cloudy), Pr(dry,damp,soggy | sunny,sunny,rainy), ... Pr(dry,damp,soggy | rainy,rainy,rainy)

这种方法是可行的，但是通过穷举计算每一个组合的概率找到最可能的序列是极为昂贵的。与前向算法类似，我们可以利用这些概率的时间不变性来降低计算复杂度。

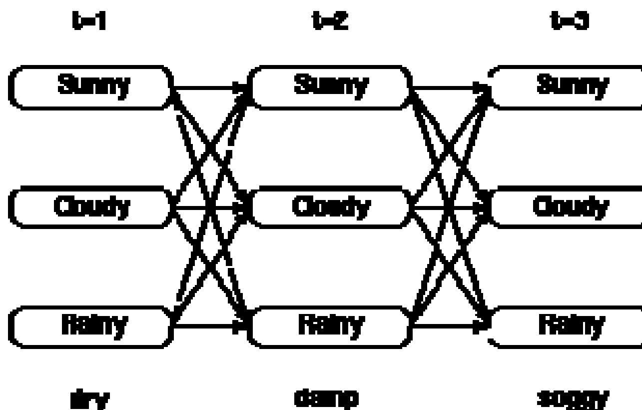
2.使用递归降低复杂度

给定一个观察序列和一个隐马尔科夫模型（HMM），我们将考虑递归地寻找最有可能的隐藏状态序列。我们首先定义局部概率δ,它是到达网格中的某个特殊的中间状态时的概率。然后，我们将介绍如何在 t=1 和 t=n(>1)时计算这些局部概率。

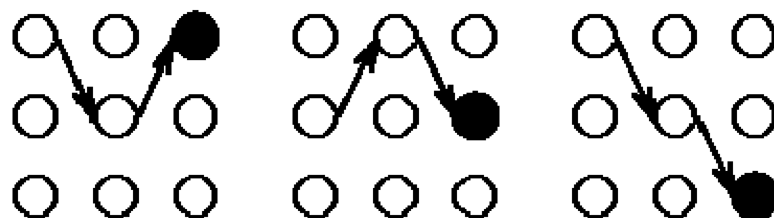
这些局部概率与前向算法中所计算的局部概率是不同的，因为它们表示的是时刻 t 时到达某个状态最可能的路径的概率，而不是所有路径概率的总和。

2a.局部概率δ's 和局部最佳途径

考虑下面这个网格，它显示的是天气状态及对于观察序列干燥，湿润及湿透的一阶状态转移情况：



对于网格中的每一个中间及终止状态，都有一个到达该状态的最可能路径。举例来说，在 t=3 时刻的 3 个状态中的每一个都有一个到达此状态的最可能路径，或许是这样的：



我们称这些路径局部最佳路径(partial best paths)。其中每个局部最佳路径都有一个相关联的概率，即局部概率或δ。与前向算法中的局部概率不同，δ是到达该状态（最可能）的一条路径的概率。

因而δ(i,t)是 t 时刻到达状态 i 的所有序列概率中最大的概率，而局部最佳路径是得到此最大概率的隐藏状态序列。对于每一个可能的 i 和 t 值来说，这一类概率（及局部路径）均存在。

特别地，在 t=T 时每一个状态都有一个局部概率和一个局部最佳路径。这样我们就可以通过选择此时刻包含最大局部概率的状态及其相应的局部最佳路径来确定全局最佳路径（最佳隐藏状态序列）。

2b. 计算 t=1 时刻的局部概率δ’s

我们计算的局部概率δ是作为最可能到达我们当前位置的路径的概率（已知的特殊知识如观察概率及前一个状态的概率）。当 t=1 的时候，到达某状态的最可能路径明显是不存在的；但是，我们使用 t=1 时的所处状态的初始概率及相应的观察状态 k1 的观察概率计算局部概率δ；即

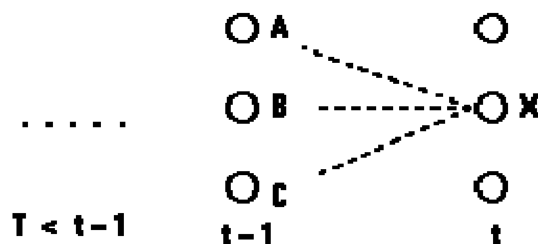
$$\delta_1(i) = \pi(i) b_{ik_1}$$

——与前向算法类似，这个结果是通过初始概率和相应的观察概率相乘得出的。

2c. 计算 t>1 时刻的局部概率δ’s

现在我们来展示如何利用 t-1 时刻的局部概率δ计算 t 时刻的局部概率δ。

考虑如下的网格：



我们考虑计算 t 时刻到达状态 X 的最可能的路径；这条到达状态 X 的路径将通过 t-1 时刻的状态 A，B 或 C 中的某一个。

因此，最可能的到达状态 X 的路径将是下面这些路径的某一个

（状态序列），…，A，X

（状态序列），…，B，X

或　　　　　　（状态序列），…，C，X

我们想找到路径末端是 AX,BX 或 CX 并且拥有最大概率的路径。

回顾一下马尔科夫假设：给定一个状态序列，一个状态发生的概率只依赖于前 n 个状态。特别地，在一阶马尔可夫假设下，状态 X 在一个状态序列后发生的概率只取决于之前的一个状态，即

Pr（到达状态 A 最可能的路径）.Pr（X ｜ A）. Pr（观察状态 ｜ X）

与此相同，路径末端是 AX 的最可能的路径将是到达 A 的最可能路径再紧跟 X。相似地，这条路径的概率将是：

Pr（到达状态 A 最可能的路径）.Pr（X ｜ A）. Pr（观察状态 ｜ X）

因此，到达状态 X 的最可能路径概率是：

$$Pr(X \ at \ time \ t) = \max_{i=A,B,C} Pr(i \ at \ time \ (t-1)) \times Pr(X|i) \times Pr(obs. \ at \ time \ t|X)$$

其中第一项是 t-1 时刻的局部概率δ，第二项是状态转移概率以及第三项是观察概率。

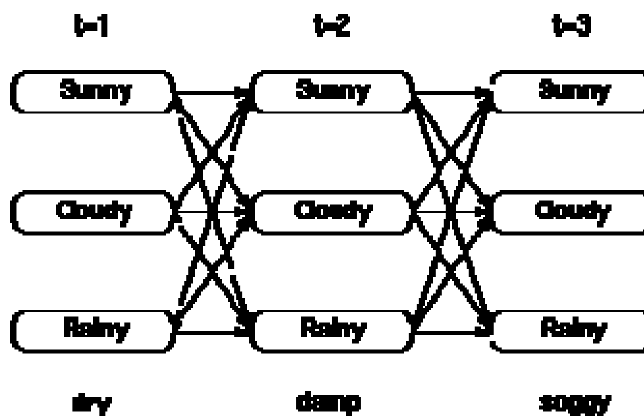
泛化上述公式，就是在 t 时刻，观察状态是 kt，到达隐藏状态 i 的最佳局部路径的概率是：

$$\delta_t(i) = \max_j(\delta_{t-1}(j) a_{ji} b_{ik_t})$$

这里，我们假设前一个状态的知识（局部概率）是已知的，同时利用了状态转移概率和相应的观察概率之积。然后，我们就可以在其中选择最大的概率了（局部概率δ）。

2d.反向指针，φ’s

考虑下面这个网格



在每一个中间及终止状态我们都知道了局部概率，δ(i,t)。然而我们的目标是在给定一个观察序列的情况下寻找网格中最可能的隐藏状态序列——因此，我们需要一些方法来记住网格中的局部最佳路径。

回顾一下我们是如何计算局部概率的，计算 t 时刻的δ’s 我们仅仅需要知道 t-1 时刻的δ’s。在这个局部概率计算之后，就有可能记录前一时刻哪个状态生成了δ(i,t)——也就是说，在 t-1 时刻系统必须处于某个状态，该状态导致了系

统在 t 时刻到达状态 i 是最优的。这种记录（记忆）是通过对每一个状态赋予一个反向指针ϕ完成的，这个指针指向最优的引发当前状态的前一时刻的某个状态。

形式上，我们可以写成如下的公式

$$\phi_t(i) = argmax_j(\delta_{t-1}(j)a_{ji})$$

其中 argmax 运算符是用来计算使括号中表达式的值最大的索引 j 的。

请注意这个表达式是通过前一个时间步骤的局部概率δ’s 和转移概率计算的，并不包括观察概率（与计算局部概率δ’s 本身不同）。这是因为我们希望这些ϕ’s 能回答这个问题“如果我在这里，最可能通过哪条路径到达下一个状态？”——这个问题与隐藏状态有关，因此与观察概率有关的混淆（矩阵）因子是可以被忽略的。

2e.维特比算法的优点

使用 Viterbi 算法对观察序列进行解码有两个重要的优点：

1. 通过使用递归减少计算复杂度——这一点和前向算法使用递归减少计算复杂度是完全类似的。

2.维特比算法有一个非常有用的性质，就是对于观察序列的整个上下文进行了最好的解释（考虑）。事实上，寻找最可能的隐藏状态序列不止这一种方法，其他替代方法也可以，譬如，可以这样确定如下的隐藏状态序列：

$$X_i = (X_{i_1}, X_{i_2}, \ldots, X_{i_T})$$

其中

$$i_1 = argmax_j(\pi(j)b_{jk_1})$$
$$i_t = argmax_j(a_{i_{t-1}k_t}b_{jk_t})$$

这里，采用了“自左向右”的决策方式进行一种近似的判断，其对于每个隐藏状态的判断是建立在前一个步骤的判断的基础之上（而第一步从隐藏状态的初始向量π开始）。

这种做法，如果在整个观察序列的中部发生“噪音干扰”时，其最终的结果将与正确的答案严重偏离。

相反， 维特比算法在确定最可能的终止状态前将考虑整个观察序列，然后通过ϕ指针“回溯”以确定某个隐藏状态是否是最可能的隐藏状态序列中的一员。这是非常有用的，因为这样就可以孤立序列中的“噪音”，而这些“噪音”在实时数据中是很常见的。

3.小结

维特比算法提供了一种有效的计算方法来分析隐马尔科夫模型的观察序列，并捕获最可能的隐藏状态序列。它利用递归减少计算量，并使用整个序列的上下文来做判断，从而对包含“噪音”的序列也能进行良好的分析。

在使用时，维特比算法对于网格中的每一个单元(cell)都计算一个局部概率，同时包括一个反向指针用来指示最可能的到达该单元的路径。当完成整个计算过程后，首先在终止时刻找到最可能的状态，然后通过反向指针回溯到 t=1 时刻，这样回溯路径上的状态序列就是最可能的隐藏状态序列了。

1、维特比算法的形式化定义

维特比算法可以形式化的概括为：

对于每一个 i，i = 1，… ，n，令：

$$\mathbf{X}_i = (X_{i_1}, X_{i_2}, \ldots, X_{i_T})$$

——这一步是通过隐藏状态的初始概率和相应的观察概率之积计算了 t=1 时刻的局部概率。

对于 t=2，…，T 和 i=1，…，n,令：

$$\delta_t(i) = \max_j(\delta_{t-1}(j)a_{ji}b_{ik_t})$$

$$\phi_t(i) = argmax_j(\delta_{t-1}(j)a_{ji})$$

——这样就确定了到达下一个状态的最可能路径，并对如何到达下一个状态做了记录。具体来说首先通过考察所有的转移概率与上一步获得的最大的局部概率之积，然后记录下其中最大的一个，同时也包含了上一步触发此概率的状态。

令：

$$i_t = argmax(\delta_T(i))$$

——这样就确定了系统完成时(t=T)最可能的隐藏状态。

对于 t=T-1，…，1

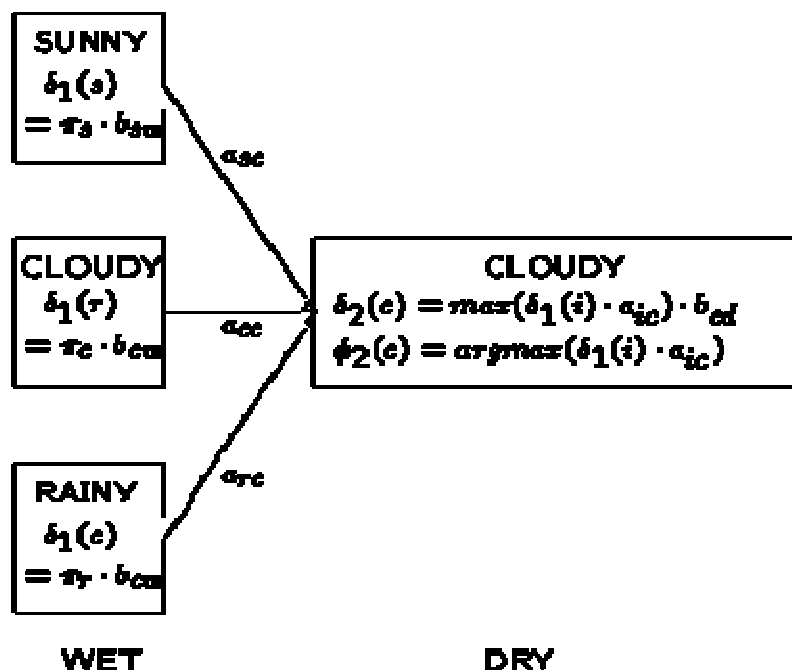
令：

$$i_t = \phi_{t+1}(i_{t+1})$$

——这样就可以按最可能的状态路径在整个网格回溯。回溯完成时，对于观察序列来说，序列 i1 … iT 就是生成此观察序列的最可能的隐藏状态序列。

2.计算单独的δ’s和φ’s

维特比算法中的局部概率δ’s的计算与前向算法中的局部概率α’s的很相似。下面这幅图表显示了δ’s和φ’s的计算细节，可以对比一下前向算法 3中的计算局部概率α’s的那幅图表：



唯一不同的是前向算法中计算局部概率α’s时的求和符号（Σ）在维特比算法中计算局部概率δ’s时被替换为*max*——这一个重要的不同也说明了在维特比算法中我们选择的是到达当前状态的最可能路径，而不是总的概率。我们在维特比算法中维护了一个“反向指针”记录了到达当前状态的最佳路径，即在计算φ’s时通过*argmax*运算符获得。

总结(Summary)

对于一个特定的隐马尔科夫模型，维特比算法被用来寻找生成一个观察序列的最可能的隐藏状态序列。我们利用概率的时间不变性，通过避免计算网格中每一条路径的概率来降低问题的复杂度。维特比算法对于每一个状态(t>1)都保存了一个反向指针(φ)，并在每一个状态中存储了一个局部概率(δ)。

局部概率δ是由反向指针指示的路径到达某个状态的概率。

当 t=T 时，维特比算法所到达的这些终止状态的局部概率δ’s 是按照最优（最可能）的路径到达该状态的概率。因此，选择其中最大的一个，并回溯找出所隐藏的状态路径，就是这个问题的最好答案。

关于维特比算法，需要着重强调的一点是它不是简单的对于某个给定的时间点选择最可能的隐藏状态，而是基于全局序列做决策——因此，如果在观察序列中有一个“非寻常”的事件发生，对于维特比算法的结果也影响不大。

这在语音处理中是特别有价值的，譬如当某个单词发音的一个中间音素出现失真或丢失的情况时，该单词也可以被识别出来。

仍然需要说明的是，本节不是这个系列的翻译，而是作为维特比算法这一章的补充，将UMDHMM这个C语言版本的HMM工具包中的维特比算法程序展示给大家，并运行包中所附带的例子。关于UMDHMM这个工具包的介绍，大家可以参考前向算法 4中的介绍。

维特比算法程序示例如下（在viterbi.c中):

```
void Viterbi(HMM *phmm, int T, int *O, double **delta, int **psi,int *q,
double *pprob)
{
    int i, j; /* state indices */
    int t; /* time index */

    int maxvalind;
    double maxval, val;

    /* 1. Initialization */

    for (i = 1; i <= phmm->N; i++)
    {
        delta[1][i] = phmm->pi[i] * (phmm->B[i][O[1]]);
        psi[1][i] = 0;
    }

    /* 2. Recursion */
    for (t = 2; t <= T; t++)
    {
        for (j = 1; j <= phmm->N; j++)
        {
            maxval = 0.0;
            maxvalind = 1;
            for (i = 1; i <= phmm->N; i++)
            {
                val = delta[t-1][i]*(phmm->A[i][j]);
                if (val > maxval)
                {
                    maxval = val;
                    maxvalind = i;
                }
            }

            delta[t][j] = maxval*(phmm->B[j][O[t]]);
            psi[t][j] = maxvalind;

        }
    }

    /* 3. Termination */

    *pprob = 0.0;
    q[T] = 1;
```

```
    for (i = 1; i <= phmm->N; i++)
    {
        if (delta[T][i] > *pprob)
        {
            *pprob = delta[T][i];
            q[T] = i;
        }
    }

    /* 4. Path (state sequence) backtracking */

    for (t = T - 1; t >= 1; t--)
        q[t] = psi[t+1][q[t+1]];

}
```

在 UMDHMM 包中所生成的 4 个可执行程序中，testvit 是用来测试维特比算法的， 对于给定的观察符号序列及 HMM，利用 Viterbi 算法生成最可能的隐藏状态序列。这里我们利用 UMDHMM 包中 test.hmm 和 test.seq 来测试维特比算法，关于这两个文件，具体如下：

test.hmm：

```
M= 2
N= 3
A:
0.333 0.333 0.333
0.333 0.333 0.333
0.333 0.333 0.333
B:
0.5 0.5
0.75 0.25
0.25 0.75
pi:
0.333 0.333 0.333
```

---

test.seq：

```
T= 10
1 1 1 1 2 1 2 2 2 2
```

对于维特比算法的测试程序 testvit 来说，运行：

```
testvit test.hmm test.seq
```

结果如下：

```
————————————————————
Viterbi using direct probabilities
Viterbi MLE log prob = -1.387295E+01
Optimal state sequence:
T= 10
2 2 2 2 3 2 3 3 3 3
————————————————————
Viterbi using log probabilities
Viterbi MLE log prob = -1.387295E+01
Optimal state sequence:
T= 10
2 2 2 2 3 2 3 3 3 3
————————————————————
The two log probabilites and optimal state sequences
should identical (within numerical precision).
```

序列“2 2 2 2 3 2 3 3 3 3”就是最终所找到的隐藏状态序列。好了，维特比算法这一章就到此为止了。

七、前向-后向算法(Forward-backward algorithm)

根据观察序列生成隐马尔科夫模型(Generating a HMM from a sequence of obersvations)

与 HMM 模型相关的“有用”的问题是评估（前向算法）和解码（维特比算法）——它们一个被用来测量一个模型的相对适用性，另一个被用来推测模型 隐藏的部分在做什么（“到底发生了”什么）。可以看出它们都依赖于隐马尔科夫模型（HMM）参数这一先验知识——状态转移矩阵，混淆（观察）矩阵，以及π向量（初始化概率向量）。

然而，在许多实际问题的情况下这些参数都不能直接计算的，而要需要进行估计——这就是隐马尔科夫模型中的学习问题。前向-后向算法就可以以一个观察序 列为基础来进行这样的估计，而这个观察序列来自于一个给定的集合，它所代表的是一个隐马尔科夫模型中的一个已知的隐藏集合。

一个例子可能是一个庞大的语音处理数据库，其底层的语音可能由一个马尔可夫过程基于已知的音素建模的，而其可以观察的部分可能由可识别的状态（可能通过一些矢量数据表示）建模的，但是没有（直接）的方式来获取隐马尔科夫模型（HMM）参数。

前向-后向算法并非特别难以理解，但自然地比前向算法和维特比算法更复杂。由于这个原因，这里就不详细讲解前向-后向算法了（任何有关 HMM 模型的参考文献都会提供这方面的资料的）。

总之，前向-后向算法首先对于隐马尔科夫模型的参数进行一个初始的估计（这很可能是完全错误的），然后通过对于给定的数据评估这些参数的的价值并减少它们所引起的错误来重新修订这些 HMM 参数。从这个意义上讲，它是以一种梯度下降的形式寻找一种错误测度的最小值。

之所以称其为前向-后向算法，主要是因为对于网格中的每一个状态，它既计算到达此状态的“前向”概率（给定当前模型的近似估计），又计算生成此模型最 终状态的“后向”概率（给定当前模型的近似估计）。 这些都可以通过利用递归进行有利地计算，就像我们已经看到的。可以通过利用近似的 HMM 模型参数来提高这些中间概率进行调整，而这些调整又形成了前向-后 向算法迭代的基础。

要理解前向-后向算法，首先需要了解两个算法：后向算法和 EM 算法。后向算法是必须的，因为前向-后向算法就是利用了前向算法与后向算法中的变 量因子，其得名也因于此；而 EM 算法不是必须的，不过由于前向-后向算法是 EM 算法的一个特例，因此了解一下 EM 算法也是有好处的，说实话，对于 EM 算 法，我也是云里雾里的。好了，废话少说，我们先谈谈后向算法。

1、后向算法（Backward algorithm）

其实如果理解了前向算法，后向算法也是比较好理解的，这里首先重新定义一下前向算法中的局部概率at(i)，称其为前向变量，这也是为前向-后向算法做点准备：

$$\alpha_t(i) = P(O_1 O_2 \cdots O_t, q_t = S_i|\lambda)$$

相似地，我们也可以定义一个后向变量Bt(i)（同样可以理解为一个局部概率）：

$$\beta_t(i) = P(O_{t+1} O_{t+2} \cdots O_T|q_t = S_i, \lambda)$$

后向变量(局部概率)表示的是已知隐马尔科夫模型λ及t时刻位于隐藏状态Si这一事实，从t+1 时刻到终止时刻的局部观察序列的概率。同样与前向算法相似，我们可以从后向前（故称之为后向算法）递归地计算后向变量：

1）初始化，令t=T时刻所有状态的后向变量为 1：

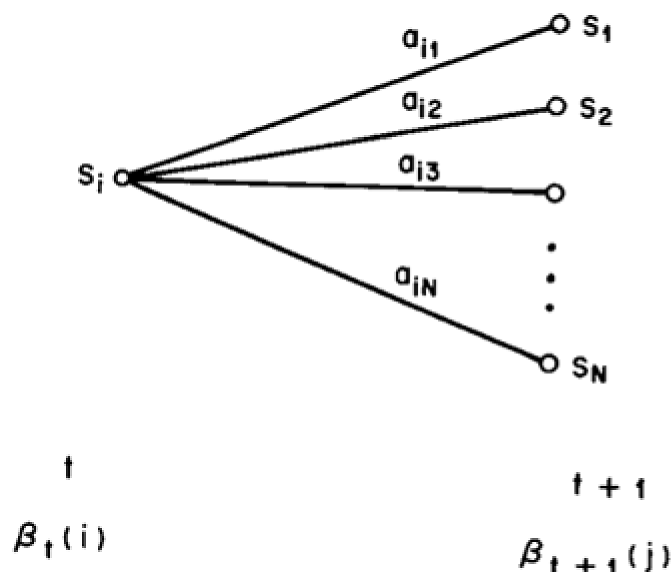
$$\beta_T(i) = 1, \quad 1 \le i \le N$$

2）归纳，递归计算每个时间点，t=T-1,T-2,…,1 时的后向变量：

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(O_{t+1}) \beta_{t+1}(j),$$

$$t = T-1, T-2, \cdots, 1, 1 \le i \le N$$

这样就可以计算每个时间点上所有的隐藏状态所对应的后向变量，如果需要利用后向算法计算观察序列的概率，只需将t=1 时刻的后向变量（局部概率）相加即可。下图显示的是t+1 时刻与t时刻的后向变量之间的关系：



上述主要参考自HMM经典论文《A tutorial on Hidden Markov Models and selected applications in speech recognition》。下面我们给出利用后向算法计算观察序列概率的程序示例，这个程序仍然来自于UMDHMM。

后向算法程序示例如下（在 backward.c 中）：

```
void Backward(HMM *phmm, int T, int *O, double **beta, double *pprob)
{
    int i, j; /* state indices */
    int t; /* time index */
    double sum;

    /* 1. Initialization */
    for (i = 1; i <= phmm->N; i++)
        beta[T][i] = 1.0;

    /* 2. Induction */
    for (t = T - 1; t >= 1; t--)
    {
        for (i = 1; i <= phmm->N; i++)
        {
            sum = 0.0;
            for (j = 1; j <= phmm->N; j++)
                sum += phmm->A[i][j] *
                            (phmm->B[j][O[t+1]])*beta[t+1][j];
            beta[t][i] = sum;
        }
    }
```

```
    /* 3. Termination */
    *pprob = 0.0;
    for (i = 1; i <= phmm->N; i++)
        *pprob += beta[1][i];
}
```

好了，后向算法就到此为止了，下一节我们粗略的谈谈 EM 算法。

前向-后向算法是 Baum 于 1972 年提出来的，又称之为 Baum-Welch 算法，虽然它是 EM(Expectation- Maximization)算法的一个特例,但 EM 算法却是于 1977 年提出的。那么为什么说前向-后向算法是 EM 算法的一个特例呢？这里有两点需要说明 一下。

第一,1977 年 A. P. Dempster、N. M. Laird、D. B. Rubin 在其论文“Maximum Likelihood from Incomplete Data via the EM Algorithm”中首次提出了 EM 算法的概念，但是他们也在论文的介绍中提到了在此之前就有一些学者利用了 EM 算法的思想解决了一些特殊问题，其中就 包括了 Baum 在 70 年代初期的相关工作，只是这类方法没有被总结而已，他们的工作就是对这类解决问题的方法在更高的层次上定义了一个完整的 EM 算法框 架。

第二，对于前向-后向算法与 EM 算法的关系，此后在许多与 HMM 或 EM 相关的论文里都被提及，其中贾里尼克（Jelinek）老先生在 1997 所著的 书“Statistical Methods for Speech Recognition”中对于前向-后向算法与 EM 算法的关系进行了完整的描述，读者有兴趣的话可以找来这本书读读。

关于 EM 算法的讲解，网上有很多，这里我就不献丑了，直接拿目前搜索“EM 算法”在 Google 排名第一的文章做了参考，希望读者不要拍砖：

EM 算法是 Dempster，Laind，Rubin 于 1977 年提出的求参数极大似然估计的一种方法，它可以从非完整数据集中对参数进行 MLE 估计，是一种非常简单实用的学习算法。这种方法可以广泛地应用于处理缺损数据，截尾数据，带有讨厌数据等所谓的不完全数据(incomplete data)。

假定集合$Z = (X,Y)$由观测数据 X 和未观测数据Y 组成，$Z = (X,Y)$和 X 分别称为完整数据和不完整数据。假设Z的联合概率密度被参数化地定义为$P(X, Y|\Theta)$，其中Θ 表示要被估计的参数。Θ 的最大似然估计是求不完整数据的对数似然函数$L(X;\Theta)$的最大值而得到的：

$$L(\Theta; X) = \log p(X|\Theta) = \int \log p(X, Y|\Theta)dY \quad (1)$$

EM算法包括两个步骤：由E步和M步组成，它是通过迭代地最大化完整数据的对数似然函数$Lc(X;\Theta)$的期望来最大化不完整数据的对数似然函数，其中：

$$Lc(X;\Theta) = \log p(X, Y|\Theta) \quad (2)$$

假设在算法第t次迭代后Θ 获得的估计记为$\Theta(t)$ ，则在（t+1）次迭代时，

E-步：计算完整数据的对数似然函数的期望，记为：

$$Q(\Theta|\Theta(t)) = E\{Lc(\Theta;Z)|X;\Theta(t)\}; \quad (3)$$

M-步：通过最大化$Q(\Theta|\Theta(t))$ 来获得新的Θ 。

通过交替使用这两个步骤，EM算法逐步改进模型的参数，使参数和训练样本的似然概率逐渐增大，最后终止于一个极大点。

直观地理解EM算法，它也可被看作为一个逐次逼近算法：事先并不知道模

型的参数，可以随机的选择一套参数或者事先粗略地给定某个初始参数λ0，确定出对应于这组参数的最可能的状态，计算每个训练样本的可能结果的概率，在当前的状态下再由样本对参数修正，重新估计参数λ，并在新的参数下重新确定模型的状态，这样，通过多次的迭代，循环直至某个收敛条件满足为止，就可以使得模型的参数逐渐逼近真实参数。

EM算法的主要目的是提供一个简单的迭代算法计算后验密度函数，它的最大优点是简单和稳定，但容易陷入局部最优。

参考原文见：
http://49805085.spaces.live.com/Blog/cns!145C40DDDB4C6E5!670.entry

注意上面那段粗体字，读者如果觉得EM算法不好理解的话，就记住这段粗体字的意思吧！

有了后向算法和EM算法的预备知识，下一节我们就正式的谈一谈前向-后向算法。

隐马尔科夫模型（HMM）的三个基本问题中，第三个HMM参数学习的问题是最难的，因为对于给定的观察序列O，没有任何一种方法可以精确地找到一组最优的隐马尔科夫模型参数（A、B、π）使P(O|λ)最大。因而，学者们退而求其次，不能使P(O|λ)全局最优，就寻求使其局部最优（最大化）的解决方法，而前向-后向算法（又称之为Baum-Welch算法）就成了隐马尔科夫模型学习问题的一种替代（近似）解决方法。

我们首先定义两个变量。**给定观察序列O及隐马尔科夫模型λ，定义t时刻位于隐藏状态Si的概率变量为：**

$$\gamma_t(i) = P(q_t = S_i|O, \lambda)$$

回顾一下第二节中关于前向变量at(i)及后向变量Bt(i)的定义，我们可以很容易地将上式用前向、后向变量表示为：

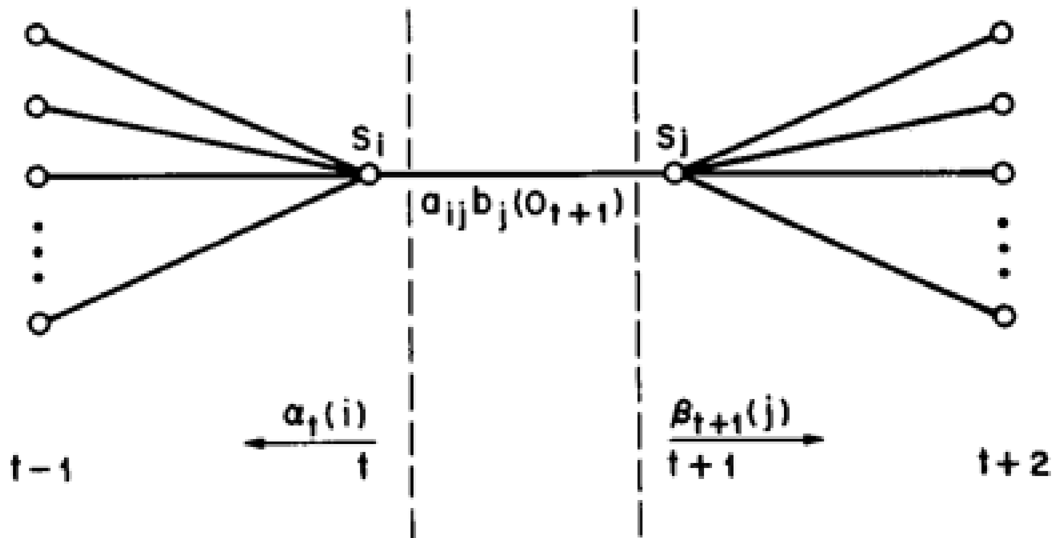
$$\gamma_t(i) = \frac{\alpha_t(i)\,\beta_t(i)}{P(O|\lambda)} = \frac{\alpha_t(i)\,\beta_t(i)}{\sum_{i=1}^{N}\alpha_t(i)\,\beta_t(i)}$$

其中分母的作用是确保：$\sum_{i=1}^{N}\gamma_t(i) = 1$

给定观察序列O及隐马尔科夫模型λ，定义t时刻位于隐藏状态Si及t+1 时刻位于隐藏状态Sj的概率变量为：

$$\xi_t(i, j) = P(q_t = S_i, q_{t+1} = S_j|O, \lambda)$$

该变量在网格中所代表的关系如下图所示：



同样，该变量也可以由前向、后向变量表示：

$$\xi_t(i,j) = \frac{\alpha_t(i)\ a_{ij}b_j(O_{t+1})\ \beta_{t+1}(j)}{P(O|\lambda)}$$

$$= \frac{\alpha_t(i)\ a_{ij}b_j(O_{t+1})\ \beta_{t+1}(j)}{\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_t(i)\ a_{ij}b_j(O_{t+1})\ \beta_{t+1}(j)}$$

而上述定义的两个变量间也存在着如下关系：

$$\gamma_t(i) = \sum_{j=1}^{N} \xi_t(i,j)$$

如果对于时间轴t上的所有$\gamma_t(i)$相加，我们可以得到一个总和，它可以被解释为从其他隐藏状态访问Si的期望值（网格中的所有时间的期望），或者，如果我们求和时不包括时间轴上的t=T时刻，那么它可以被解释为从隐藏状态Si出发的状态转移期望值。相似地，如果对$\xi_t(i,j)$在时间轴t上求和（从t=1 到t=T-1），那么该和可以被解释为从状态Si到状态Sj的状态转移期望值。即：

$$\sum_{t=1}^{T-1} \gamma_t(i) = \text{expected number of transitions from } S_i$$

$$\sum_{t=1}^{T-1} \xi_t(i,j) = \text{expected number of transitions from } S_i \text{ to } S_j$$

上一节我们定义了两个变量及相应的期望值，本节我们利用这两个变量及其期望值来重新估计隐马尔科夫模型（HMM）的参数π，A 及 B：

$$\bar{\pi}_i = \text{expected frequency (number of times) in state } S_i \text{ at time } (t = 1) = \gamma_1(i)$$

$$\bar{a}_{ij} = \frac{\text{expected number of transitions from state } S_i \text{ to state } S_j}{\text{expected number of transitions from state } S_i} = \frac{\sum_{t=1}^{T-1} \xi_t(i, j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

$$\bar{b}_j(k) = \frac{\text{expected number of times in state } j \text{ and observing symbol } v_k}{\text{expected number of times in state } j} = \frac{\sum_{t=1,\ \text{s.t. } O_t = v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}.$$

如果我们定义当前的HMM模型为$\lambda = (A, B, \pi)$，那么可以利用该模型计算上面三个式子的右端；我们再定义重新估计的HMM模型为$\bar{\lambda} = \bar{A}, \bar{B}, \bar{\pi}$，那么上面三个式子的左端就是重估的HMM模型参数。Baum及他的同事在70年代证明了$P(O|\bar{\lambda}) > P(O|\lambda)$，因此如果我们迭代地的计算上面三个式子，由此不断地重新估计HMM的参数，那么在多次迭代后可以得到的HMM模型的一个最大似然估计。不过需要注意的是，前向-后向算法所得的这个结果（最大似然估计）是一个局部最优解。

关于前向-后向算法和EM算法的具体关系的解释，大家可以参考HMM经典论文《A tutorial on Hidden Markov Models and selected applications in speech recognition》，这里就不详述了。下面我们给出UMDHMM中的前向-后向算法示例，这个算法比较复杂，这里只取主函数，其中所引用的函数大家如果感兴趣的话可以自行参考UMDHMM。

前向-后向算法程序示例如下（在 baum.c 中）：

```
void BaumWelch(HMM *phmm, int T, int *O, double **alpha, double **beta,
double **gamma, int *pniter, double *plogprobinit, double *plogprobfinal)
{
    int i, j, k;
    int t, l = 0;

    double logprobf, logprobb, threshold;
    double numeratorA, denominatorA;
    double numeratorB, denominatorB;

    double ***xi, *scale;
    double delta, deltaprev, logprobprev;
```

```
deltaprev = 10e-70;

xi = AllocXi(T, phmm->N);
scale = dvector(1, T);

ForwardWithScale(phmm, T, O, alpha, scale, &logprobf);
*plogprobinit = logprobf; /* log P(O |intial model) */
BackwardWithScale(phmm, T, O, beta, scale, &logprobb);
ComputeGamma(phmm, T, alpha, beta, gamma);
ComputeXi(phmm, T, O, alpha, beta, xi);
logprobprev = logprobf;

do
{

    /* reestimate frequency of state i in time t=1 */
    for (i = 1; i <= phmm->N; i++)
        phmm->pi[i] = .001 + .999*gamma[1][i];

    /* reestimate transition matrix and symbol prob in
            each state */
    for (i = 1; i <= phmm->N; i++)
    {
        denominatorA = 0.0;
        for (t = 1; t <= T - 1; t++)
            denominatorA += gamma[t][i];

        for (j = 1; j <= phmm->N; j++)
        {
            numeratorA = 0.0;
            for (t = 1; t <= T - 1; t++)
                numeratorA += xi[t][i][j];
            phmm->A[i][j] = .001 +
                             .999*numeratorA/denominatorA;
        }

        denominatorB = denominatorA + gamma[T][i];
        for (k = 1; k <= phmm->M; k++)
        {
            numeratorB = 0.0;
            for (t = 1; t <= T; t++)
            {
                if (O[t] == k)
                    numeratorB += gamma[t][i];
            }
```

```
            phmm->B[i][k] = .001 +
                              .999*numeratorB/denominatorB;
        }
    }

    ForwardWithScale(phmm, T, O, alpha, scale, &logprobf);
    BackwardWithScale(phmm, T, O, beta, scale, &logprobb);
    ComputeGamma(phmm, T, alpha, beta, gamma);
    ComputeXi(phmm, T, O, alpha, beta, xi);

    /* compute difference between log probability of
        two iterations */
    delta = logprobf - logprobprev;
    logprobprev = logprobf;
    l++;

}
while (delta > DELTA); /* if log probability does not
                        change much, exit */

*pniter = l;
*plogprobfinal = logprobf; /* log P(O|estimated model) */
FreeXi(xi, T, phmm->N);
free_dvector(scale, 1, T);
}
```

前向-后向算法就到此为止了。

八、总结(Summary)

通常，模式并不是单独的出现，而是作为时间序列中的一个部分——这个过程有时候可以被辅助用来对它们进行识别。在基于时间的进程中，通常都会使 用一些假设——一个最常用的假设是进程的状态只依赖于前面N个状态——这样我们就有了一个N阶马尔科夫模型。最简单的例子是N = 1。

存在很多例子，在这些例子中进程的状态（模式）是不能够被直接观察的，但是可以非直接地，或者概率地被观察为模式的另外一种集合——这样我们就可以定义一类隐马尔科夫模型——这些模型已被证明在当前许多研究领域，尤其是语音识别领域具有非常大的价值。

在实际的过程中这些模型提出了三个问题都可以得到立即有效的解决，分别是：

* 评估：对于一个给定的隐马尔科夫模型其生成一个给定的观察序列的概率是多少。前向算法可以有效的解决此问题。

* 解码：什么样的隐藏（底层）状态序列最有可能生成一个给定的观察序列。维特比算法可以有效的解决此问题。

＊ 学习：对于一个给定的观察序列样本，什么样的模型最可能生成该序列——也就是说，该模型的参数是什么。这个问题可以通过使用前向-后向算法解决。

隐马尔科夫模型（HMM）在分析实际系统中已被证明有很大的价值；它们通常的缺点是过于简化的假设，这与马尔可夫假设相关——即一个状态只依赖于前一个状态，并且这种依赖关系是独立于时间之外的（与时间无关）。

关于隐马尔科夫模型的完整论述，可参阅：

L R Rabiner and B H Juang, `An introduction to HMMs', iEEE ASSP Magazine, 3, 4-16.

全文完！

后记：这个翻译系列终于可以告一段落了，从 6 月 2 日起至今，历史四个多月，期间断断续续翻译并夹杂些自己个人的理解，希望这个系列对于 HMM 的 学习者能有些用处，我个人也就很满足了。接下来，我会结合 HMM 在自然语言处理中的一些典型应用，譬如词性标注、中文分词等，从实践的角度讲讲自己的理 解，欢迎大家继续关注 52nlp。

本文翻译自：
http://www.comp.leeds.ac.uk/roger/HiddenMarkovModels/html_dev/main.html
部分翻译参考：隐马尔科夫模型HMM自学

转载请注明出处“我爱自然语言处理”：www.52nlp.cn