浅析隐马尔科夫模型在中文分词的应用

517030910288 戴昊悦

一.引言

这学期我在实验室里接触了一些机器学习相关的项目，其中一次和和导师谈话聊到，目前人工智能浪潮里，许多大的进展都是基于空间尺度的静态算法，而在时间尺度上技术一直得不到突破。举例就是，用CNN识别静态图像、用GAN生成静态图像这些已经比较完善，但机器人控制技术等与上十年比却并没有跨越式的发展。对于机器人技术等，实际是在模拟一个序列运动，一方面在运动产生之前序列程序就已经编好了（比如弹钢琴），但同时接下来进行的每一步都和之前序列所有步有关（包括对前序列的修正和感官对后序列的预测），放在计算机就是实时的PCA降维。而这种时间序列分析正是当前亟待突破的方向；目前诸如语音识别等，本质上也不是时间序列，而是转成了空间序列，再使用RNN，LSTM等模型处理。

怀着对序列模型的兴趣，我简单了解了其中一个方向：自然语言处理（NLP）。自然语言处理是研究人与人交际中以及人与计算机交际中的语言问题的一门学科。和其他机器学习方向一样，可以分为两类：

- 判别模型：比如中文分词、词性标记、情感分析等，本质上是求条件概率$P(A|B)$，可以节省计算资源（因为对输入抽象），准确率高。
- 生成模型：比如自动写作、上下文预测等，本质上是求联合分布概率$P(A,B)$，可以提供更多的信息，但目前计算强度大，且准确率不太高。

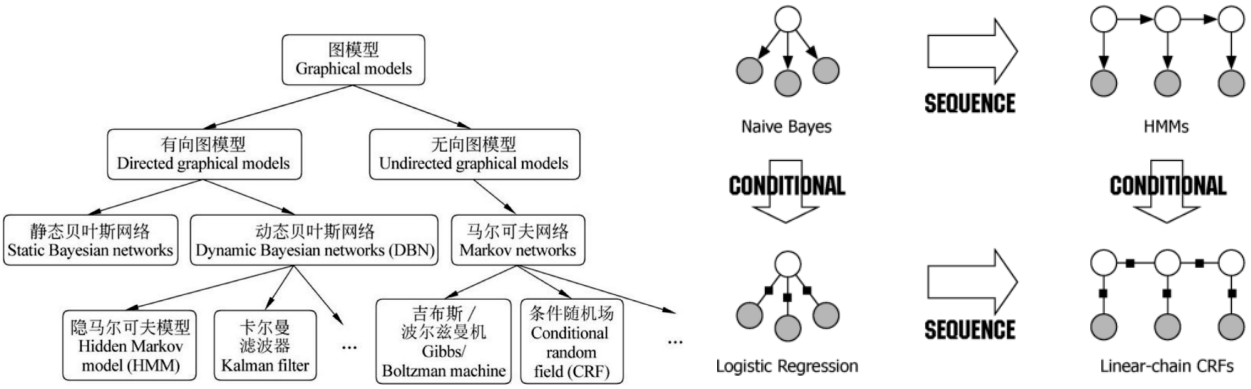
随着学习过程的深入，我发现NLP中许多模型都和概率论中随机过程有着深厚联系。比如一个语言模型通常构建为字符串s的概率分布$p(s)$，其中$p(s)$试图反映的是字符串s作为一个句子出现的频率。而对一个由l个基元（词）的句子s，即$s=\omega_1\cdots\omega_l$，其概率计算公式可以表示为：

$$p(s)=p(\omega_1)p(\omega_2|\omega_1)p(\omega_3|\omega_1\omega_2)\cdots p(\omega_l|\omega_1\cdots\omega_{l-1})=\prod_{i=1}^{l}p(\omega_i|\omega_1\cdots\omega_{i-1})$$

也即产生第i个词的概率是由已经产生的$i-1$个词$\omega_1\cdots\omega_{i-1}$决定的，这$i-1$个词被称为第$i$个词的历史，也就是研究随机过程序列。显然，随着$i$的增大，历史的长度不断增长，计算复杂度指数型增长，因此我们需要只考虑历史中的某些词。比如我们假定，出现在第i个位置上的词ω_i仅与它前面的n-1个历史词有关，称其为n元文法模型（通常取n<=3来避免前面的自由参数过多）：

- n==1：一元文法模型。也就是认为第i个词的出现独立于历史，实际就是朴素贝叶斯模型，基于贝叶斯公式：$P(B|A)=\frac{P(A|B)P(B)}{P(A)}$；
- n==2：二元文法模型。第i个词的出现仅与第$$i-1个词有关，这正是我们学的一阶马尔科夫链，也是本论文将详细讲的部分。
- n更大：如n==3是二阶马尔科夫链；n更大就不是简单选取前n-1个词了，而是有选择地进行记忆和遗

忘，这是RNN和LSTM解决的问题了。

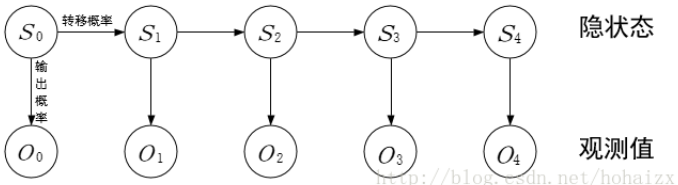


目前我的随机过程知识最能支撑我研究马尔科夫模型的兴趣。因此在接下来的部分中我将着重讲（隐）马尔科夫的相关内容，以及它在中文分词中的应用。

二.HMM模型的相关概念定义

- **What&Why**

首先，什么是隐马尔科夫模型？马尔科夫性质就是我们学到的$p(s_t|s_{t-1},s_{t-2},\dots,s_0)=p(s_t|s_{t-1})$，也就是上文提到的二元文法模型。但是很多时候实际马尔科夫链的状态序列$s_0,s_1,\dots,s_t$无法直接被观测到，称为隐状态，而我们能观测到的是o_t，称为观测值。假设o_t与且仅与当前时刻的隐状态s_t有关，并且s_t外化表现为o_t的概率称为输出概率（发射概率），我们就构建了隐马尔科夫模型。



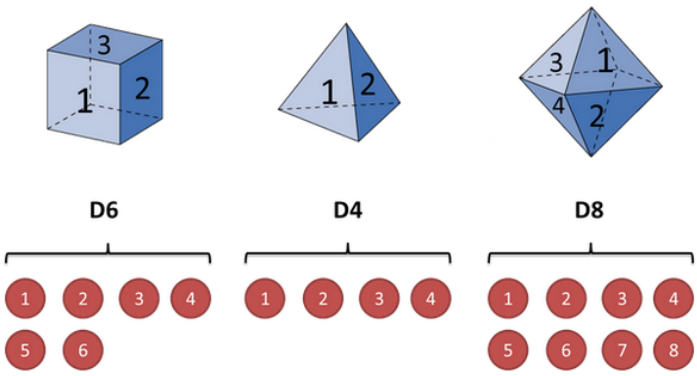
那么哪些时候要用到HMM模型呢？广义来说，NLP中任何一个字符串都是观测序列，而实际要表达的内容是计算机看不见的，也就是状态序列。从观测序列反推状态序列，实际就是HMM的解码过程。放在生活中，打麻将打扑克也是隐马尔科夫过程，要通过目前观测到的状态（已出的牌、自己手上的牌）来判断和预测整个牌局；揣摩女朋友的心思也是隐马尔科夫模型，可观测状态都是一句“再见”，在约会完和吵架完两者的隐藏状态（女朋友表达的意思）显然是不同的。

- 一个HMM模型可以通过五元组（S,K,π,A,B）具体描述：
  - S：隐藏状态的集合，$s_1,\dots,s_N$，N为隐状态个数
  - K：输出状态（观测状态）的集合，$k_1,\dots,k_M$，M为观测状态个数
  - π：隐藏状态的初始化概率，一个N维向量，$\pi_0,\pi_1,\dots,\pi_m$

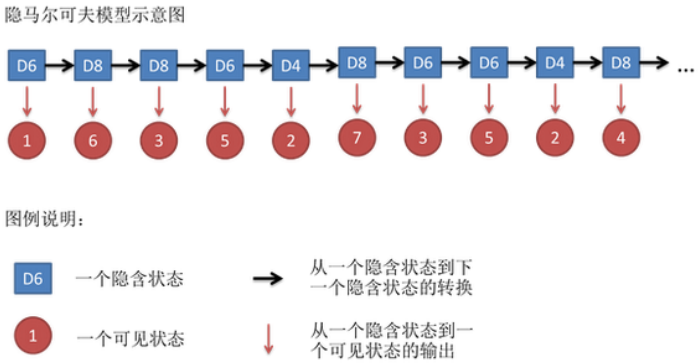
- A：隐藏状态的转移概率矩阵，N*N维，$a_{ij} = p(s_j|s_i)$
- B：隐藏状态到观测状态的发射概率，N*M维，$b_{ij} = p(o_j|s_i)$

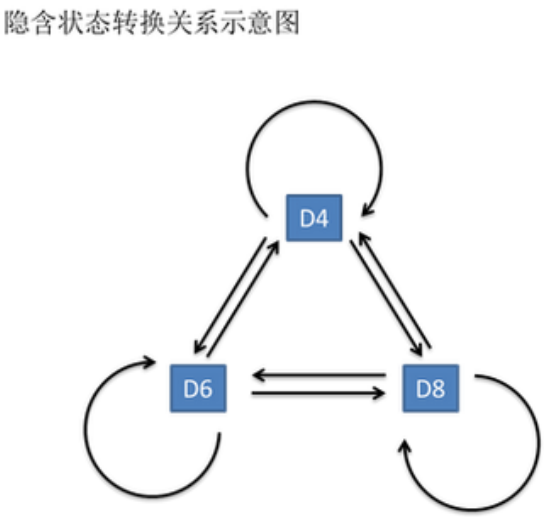
下面引入一个对我理解HMM颇有帮助经典实例——掷骰子：

假设有三个不同的骰子。如图所示，分别为D6，D4，D8，假设每个骰子掷出其每个面的概率都是均等的，并且每次掷其中一枚骰子的概率也都是1/3。



假设掷骰子的过程在黑箱中进行，只知道投出来的结果序列，我们的任务是要由结果序列去推知状态序列，也就是黑箱里每次具体发生了什么。





该HMM模型的五元组显然可以描述出来。以下是python代码初始化：

```
hiddenStates = ("D6", "D4", "D8")
observationsStates = ("1","2","3","4","5","6","7","8")
pi = {"D6": 1/3, "D4": 1/3,"D8": 1/3}
A = {
    "D6": {"D6": 1/3, "D4": 1/3,"D8": 1/3},
    "D4": {"D6": 1/3, "D4": 1/3,"D8": 1/3},
    "D8": {"D6": 1/3, "D4": 1/3,"D8": 1/3},
}
B = {
    "D6": {"1":1/6,"2":1/6,"3":1/6,"4":1/6,"5":1/6,"6":1/6,"7":0,"8":0},
    "D4": {"1":1/4,"2":1/4,"3":1/4,"4":1/4,"5":0,"6":0,"7":0,"8":0},
    "D8": {"1":1/8,"2":1/8,"3":1/8,"4":1/8,"5":1/8,"6":1/8,"7":1/8,"8":1/8},
}
#还需要对状态字符串进行索引，具体实现较简单，仅表示如下：
hStatesIndex = utility.generateStatesIndex(hiddenStates)
oStatesIndex = utility.generateStatesIndex(observationsStates)
A = utility.generateMatrix(A,hStatesIndex,hStatesIndex)
B = utility.generateMatrix(B,hStatesIndex,oStatesIndex)
pi = utility.generatePiVector(pi,hStatesIndex)
h = HMM(A=A,B=B,pi=pi)
```

接下来我用这个示例来演示HMM模型解决的三类主要问题：评估，解码和学习。

- **评估问题**

我已经有了观测到的序列，想知道出现这个序列的概率是多少，怎么算呢?

最简单直观的想法当然就是穷举出能产生当前观测序列的所有隐藏状态的可能，对每种隐藏状态求其产生该观测状态的概率，然后利用全概率公式求得总概率。比如我投出1，状态可能是D6，D4或D8，总概率为 $P(D6)*P(1|D6)+P(D4)*P(1|D4)+P(D8)*P(1|D8)=1/3*1/6+1/3*¼+1/3*1/8$，同样的，我投出1-6，可能的隐藏序列有D6D6,D6D8,D4D6,D4D8,D8D6,D8D8。显然，随着序列长度的增加，这种穷举再相加的算法复杂度会以指数级增长，不现实。

这时人们就考虑到了马尔科夫特性：一段长为n的序列如果知道了产生其前n-1个元素的概率，那么该段序列的概率可由前n-1个的概率和最后一个元素计算，不用对每个元素的组合穷举了。还是刚才的例子：

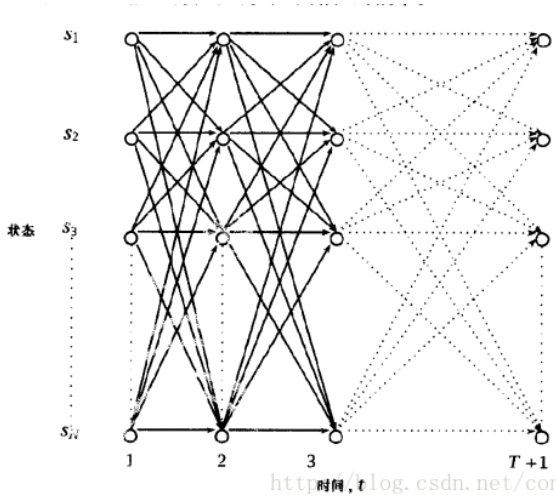
若只投一次骰子得到1，

	P1	P2	P3
D6	$\frac{1}{3} * \frac{1}{6}$		
D4	$\frac{1}{3} * \frac{1}{4}$		
D8	$\frac{1}{3} * \frac{1}{8}$		
TOTAL	0.18		

若投两次骰子得到1-6，可以理解为在第一次得到1的条件下再去得到第二次的6:

	P1	P2	P3
D6	$\frac{1}{3} * \frac{1}{6}$	$0.18 * \frac{1}{3} * \frac{1}{6}$	
D4	$\frac{1}{3} * \frac{1}{4}$	$0.18 * \frac{1}{3} * 0$	
D8	$\frac{1}{3} * \frac{1}{8}$	$0.18 * \frac{1}{3} * \frac{1}{8}$	
TOTAL	0.18	0.0175	

同理可以一直向下拓展，比如投3次得到1-6-3的概率为0.0032。需要注意的是，上面的推算中我直接写了 $0.18 * \frac{1}{3} * \frac{1}{6}$，是因为每次投哪个都是独立等可能的，也就是隐藏状态（骰子种类）的转移矩阵是均匀的。若不均匀，则应该分开来求。这个算法叫做前向算法，示意图如下：



数学表达为：

定义前向变量算子$\alpha_t(i) = p(O_1, O_2, \dots, O_t, X_t = S_i|\mu)$，表示$t$时刻，以$S_i$状态结束时之前路径上的总概率，有

$$\alpha_{t+1}(j) = \sum_{i=1}^{N} \alpha_t(i) a_{ij} b_{jO_{t+1}}, 1 \leqslant t \leqslant T, 1 \leqslant j \leqslant N$$

求和即可得到总概率：

$$p(O|\mu) = \sum_{i=1}^{N} \alpha_{t+1}(i)$$

容易发现，之前的暴力求解时间复杂度为$O(N^T)$，现在仅为$O(T * N^2)$，可接受了。相当于利用了动态规划，每次前向传播需要额外的$O(N)$空间即可。以下是python实现：

```
#计算公式中的alpha二维数组
def forward(observationsSeq):
    T = len(observationsSeq)
    N = len(pi)
    alpha = np.zeros((T,N),dtype=float)
    alpha[0,:] = pi * B[:,observationsSeq[0]]
    for t in range(1,T):
        for n in range(0,N):
        alpha[t,n] = np.dot(alpha[t-1,:],self.A[:,n]) * self.B[n,observationsSeq[t]] #
    return alpha
```

比如出现1-6-3的概率，前向算子α矩阵为：

```
[[ 0.05555556  0.08333333  0.04166667]
 [ 0.01003086  0.          0.00752315]
 [ 0.00097522  0.00146283  0.00073142]]
```

说了这么多，前向计算状态的概率和“评估”有什么关系呢？比如一个赌徒怀疑赌场里使用的骰子被做了手脚，或者黑箱里每次选用并非随机，他怎么得到这个怀疑并验证之呢？可以多次得到观测序列，若按照前向算法算出得到这些序列的概率都较小，他就有理由怀疑该隐马尔科夫模型的五元组并非这么均匀的。若他已经知道赌场的常用伎俩，比如得到1的概率为0.5，其他都为0.1，再按这个带进去算，发现出现当前序列的概率比用均匀模型算出的要大，那么他就有充分理由相信，赌场做了这样的手脚。这就是“评估”的意义所在。

- **预测问题**

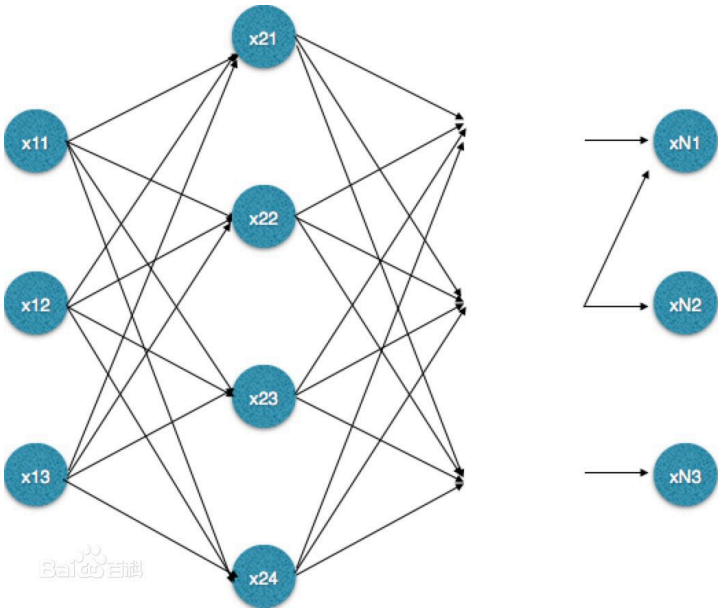
若给定一个观测序列，如何反推出最有可能的隐藏序列，“极大似然序列”呢？这就是我们考虑的预测问题。我们在日常生活中也经常用，比如一个人住在外地，该地天气对我是隐藏的，只知道这个人有习惯：下雨时散步的概率是10%，宅在家里的概率是90%，晴天时则正好相反。如果他昨天去散步了，我几乎可以推测昨天是晴天。这是序列长度为1的情况，如果序列继续变长，并且假设每天的天气是马尔科夫链，我们也可以推断出最有可能的天气链。

还是以掷骰子为例：假设只掷一次，得到1，显然最有可能是D4，即骰子的极大似然序列为D4，这是因为D4产生1的概率大于D6的1/6和D8的1/8；接下来考虑1-6。显然，要取到最大概率，第一个骰子得是D4，这时，第二个骰子取到D6时的最大概率为$P2(D6) = P1(D4) * P(D4 \to 1) * P(D4 \to D6) * P(D6 \to 6) =$，当然，前面已经确定是D4了，我们就只需考虑后面的$P(D4 \to D6) * P(D6 \to 6)$，对三个比较，得到当前最可能的是D4-D6。

仔细想我们可以知道，预测问题的思路和评估问题是类似的，只是不再对每个可能求和，而是求最大值——同样是利用动态规划。有一个理解很重要：发射概率不仅是从隐藏状态到观测层，同时也是从观测状态推知隐藏状态的工具。

预测问题的的动态规划算法叫做维特比算法（viterbi），

> 维特比算法是针对一个特殊的图——篱笆网络的有向图（Lattice)的最短路径问题而提出的。 它之所以重要，是因为凡是使用隐含马尔可夫模型描述的问题都可以用它来解码，包括今天的数字通信、语音识别、机器翻译、拼音转汉字、分词等。——《数学之美》



照着上一节对α算子的定义，这里我们同样定义算子
$\theta_t(j) = max(p(X_1, X_2, .. , , X_t = S_j, O_1, O_2, \dots, O_t|\mu))$，表示的最大概率，那么有：

$$\theta_t(j) = max(\theta_{t-1}(i) * a_{ij} * b_{jo_t}), i \in [1, N]$$

而状态t时刻通过回溯得到上一个最大概率路径。算法时间复杂度同样是$O(T * N^2)$。以下是维特比算法的python实现：

```python
def viterbi(observationsSeq):
    T = len(observationsSeq)
    N = len(pi)
    prePath = np.zeros((T,N),dtype=int)
    dpMatrix = np.zeros((T,N),dtype=float)
    dpMatrix[0,:] = pi * B[:,observationsSeq[0]]

    for t in range(1,T):
        for n in range(N):
            probs = dpMatrix[t-1,:] * A[:,n] * B[n,observationsSeq[t]]
            prePath[t,n] = np.argmax(probs)
            dpMatrix[t,n] = np.max(probs)

    maxProb = np.max(dpMatrix[T-1,:])
    maxIndex = np.argmax(dpMatrix[T-1,:])
    path = [maxIndex]

    for t in reversed(range(1,T)):#反转迭代
        path.append(prePath[t,path[-1]])

    path.reverse()
    return maxProb,path
```

同样的我们测试：

```
>> h.viterby(['1','6','3'])
>> 0.00038580246913580245,
>> ['D4','D6','D4']
>>
>> h.viterby(['1','6','3','8','7','6','2'])
>> 3.1009071914850368e-09,
>> ['D4','D6','D4','D8','D8','D6','D4']
```

Viterbi被广泛应用到分词，词性标注等应用场景。

- **学习问题**

其实前两个算法都是比较容易理解的，HMM模型的第三个经典问题——学习，较难理解。学习，就是给定观察序列，求出模型的参数（π,A,B），一般使用的是前后向算法（baum-welch算法），是一种特殊的EM算法。这里我先利用本学期的凸优化相关知识来写写我对EM算法的理解：

假设u代表了我们要的整个模型参数，我们已经有了长度为t的观察序列O，那么最可能的模型$\hat{u}$，当然使得在该$\hat{u}$下出现O的概率大于其他任何一个u的。我们定义极大似然函数：

$$L(u) = p(O|u) = \prod_{i=1}^{t} p(O_i|u)$$

在HMM问题中，利用条件概率公式，我们在上式中写了连乘，那么就是解最优化问题：

$$\hat{u} = argmax(L(u))$$

现在我们假设甚至连S都不知道，如何求解这个最优问题呢？举例说幼儿园老师有一堆糖果，想让我和小明均分了。那么需要刚开始就用天平称吗？显然不需要。我们先随便分一分，然后比，我的这堆多了就从我的里面抓一堆给小明，小明的多了就再抓一小堆到我的..如此迭代直至收敛。所谓EM算法，Expectation Maximization，就是利用了这样的思想。

我们不妨将模型的所有参数u细化成单变量，比如有θ，z等，我们也可以用分别求偏导的方法，但是EM漂亮之处如下：

假设我们有一个样本集$x^{(1)}, \ldots, x^{(m)}$，包含m个独立的样本。但每个样本i对应的类别$z^{(i)}$是未知的（相当于聚类），也即隐含变量。故我们需要估计概率模型$p(x, z)$的参数θ，对于多的z变量，目标是找到适合的θ和z让$L(\theta)$最大：

$$\sum_i \log p(x^{(i)}|\theta) = \sum_i \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}|\theta) \tag{1}$$

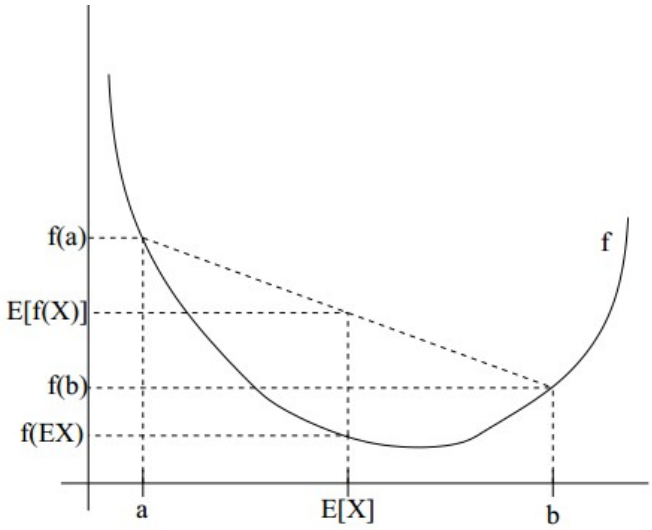
$$= \sum_i \log \sum_{z^{(i)}} Q_i(z^{(i)}) \frac{p(x^{(i)}, z^{(i)}|\theta)}{Q_i(z^{(i)})} \tag{2}$$

$$\geqslant \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}|\theta)}{Q_i(z^{(i)})} \tag{3}$$

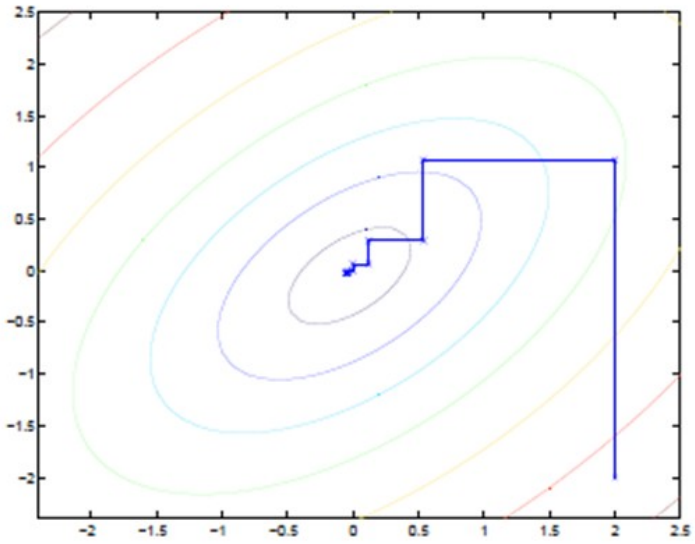
注意到Q函数：参数为模型参数的当前估计值+下一步估计值。我们要最大化(1)式，实际上是概率论中求解联合概率密度下某个变量的边缘概率密度函数，而z也是随机变量，则是对每个$z^(i)$求该联合概率密度和。为什么要加一个log呢？这是一个很巧妙的操作，(1)式中有和的对数，再求导就会很复杂，而为什么(2)式中分子分母上下同乘一个项，到了(3)式就可以转换成对数的和，然后再求导就比较简单了呢？

我们利用凸优化中学到的Jensen不等式：

如果f是凸函数，X是随机变量，那么：$E[f(X)] >= f(E[X])$



而 $E[f(X)] = \sum f(x) * p(x)$，那么(2)式中$\sum_{z^{(i)}} Q_i(z^{(i)}) \frac{p(x^{(i)},z^{(i)}|\theta)}{Q_i(z^{(i)})}$就是$\frac{p(x^{(i)},z^{(i)}|\theta)}{Q_i(z^{(i)})}$的期望，应用Jensen不等式我们可以得到下界(3)式。需要注意的是$f = \log$是凹函数，因此用反Jensen。而通过不断迭代最大化下界，抬高目标函数。



因为每个迭代中只优化一个变量，也可以画成如图平行坐标轴的坐标上升法

至此就是EM算法的大致思路，是局部最优的，至于是不是全局最优我还没看到证明。后来人们证明了baum-welch算法是EM算法的一个特例，并且证明了单调性，以下是节录的baum-welch算法的操作（以下求解部分主要用到了线性优化与凸优化的内容，部分知识点我还未掌握，因此下面的解法是从李航的统计方法中抄的，没有完全推导过）：

2．EM 算法的 E 步：求 Q 函数 $Q(\lambda,\bar{\lambda})$ ①

$$Q(\lambda,\bar{\lambda})=\sum_{I}\log P(O,I\mid\lambda)P(O,I\mid\bar{\lambda}) \tag{10.33}$$

其中，$\bar{\lambda}$ 是隐马尔可夫模型参数的当前估计值，λ 是要极大化的隐马尔可夫模型参数．

$$P(O,I\mid\lambda)=\pi_{i_1}b_{i_1}(o_1)a_{i_1i_2}b_{i_2}(o_2)\cdots a_{i_{T-1}i_T}b_{i_T}(o_T)$$

于是函数 $Q(\lambda,\bar{\lambda})$ 可以写成：

$$\begin{aligned}Q(\lambda,\bar{\lambda})=&\sum_{I}\log\pi_{i_1}P(O,I\mid\bar{\lambda})\\&+\sum_{I}\left(\sum_{t=1}^{T-1}\log a_{i_ti_{t+1}}\right)P(O,I\mid\bar{\lambda})+\sum_{I}\left(\sum_{t=1}^{T}\log b_{i_t}(o_t)\right)P(O,I\mid\bar{\lambda})\end{aligned} \tag{10.34}$$

（1）式 (10.34) 的第 1 项可以写成：

$$\sum_{I}\log\pi_{i_0}P(O,I|\bar{\lambda})=\sum_{i=1}^{N}\log\pi_i P(O,i_1=i|\bar{\lambda})$$

注意到 π_i 满足约束条件 $\sum_{i=1}^{N}\pi_i=1$，利用拉格朗日乘子法，写出拉格朗日函数：

$$\sum_{i=1}^{N}\log\pi_i P(O,i_1=i|\bar{\lambda})+\gamma\left(\sum_{i=1}^{N}\pi_i-1\right)$$

对其求偏导数并令结果为 0

$$\frac{\partial}{\partial\pi_i}\left[\sum_{i=1}^{N}\log\pi_i P(O,i_1=i|\bar{\lambda})+\gamma\left(\sum_{i=1}^{N}\pi_i-1\right)\right]=0 \qquad (10.35)$$

得

$$P(O,i_1=i|\bar{\lambda})+\gamma\pi_i=0$$

对 i 求和得到 γ

$$\gamma=-P(O|\bar{\lambda})$$

代入式 (10.35) 即得

$$\pi_i=\frac{P(O,i_1=i|\bar{\lambda})}{P(O|\bar{\lambda})} \qquad (10.36)$$

$$\pi_i=\frac{P(O,i_1=i|\bar{\lambda})}{P(O|\bar{\lambda})} \qquad (10.36)$$

$$a_{ij}=\frac{\sum_{t=1}^{T-1}P(O,i_t=i,i_{t+1}=j|\bar{\lambda})}{\sum_{t=1}^{T-1}P(O,i_t=i|\bar{\lambda})} \qquad (10.37)$$

$$b_j(k)=\frac{\sum_{t=1}^{T}P(O,i_t=j|\bar{\lambda})I(o_t=v_k)}{\sum_{t=1}^{T}P(O,i_t=j|\bar{\lambda})} \qquad (10.38)$$

那么如何求公式中的概率呢？至此引入了前后向算法：

> 欲求： $p(O,i_1=i|\lambda)$和 $p(O,i_t=i,i_{t+1}=j|\lambda)$
>
> 后向算法与前向算法相同，定义后向变量：
>
> $\beta_t(i)=p(O_{t+1},O_T|i_t=i,\lambda)$,表示在i_t时刻状态为i的情况下，后面到T时刻观测序列的概率情况。
>
> 由于最后一个时刻没有O_{T+1},所以直接规定 $\beta(T)=1$,有递归公式：
>
> $$\beta_1(i)=1_i\in[1,N]$$
>
> $$\beta_t(i)=\sum_j(a_{ij}*b_{jo_t}+\beta_{t+1}(j)),i\in[1,N]\ ,\ t\in[1,T-1]$$
>
> 推导$p(O,i_1=i|\lambda)$
>
> $$\alpha_t(i)=p(O_1,O_2\dots O_t,i_t=i|\lambda)$$

$$\beta_t(i) = p(O_{t+1}\ ,\ \ldots, O_T|i_t = i, \lambda)$$

$$p(O, i_t = i|\lambda) = p(O_1, O_2 \ldots O_t, i_t = i, O_{t+1}, .. O_T|\lambda)$$

$$= p(O_1, O_2 \ldots O_t, i_t = i|\lambda) * p(O_{t+1}, \ldots, O_T|i_t = i, O_1, O_2 \ldots O_t, \lambda)$$

$$= p(O_1, O_2 \ldots O_t, i_t = i|\lambda) * p(O_{t+1} \ldots ., O_T|i_t = i, \lambda)$$

$$= \alpha_t(i) * \beta_t(i)$$

（注意这里的独立性假设）

1．给定模型 λ 和观测 O，在时刻 t 处于状态 q_i 的概率．记

$$\gamma_t(i) = P(i_t = q_i \mid O, \lambda)$$

可以通过前向后向概率计算．事实上，

$$\gamma_t(i) = P(i_t = q_i \mid O, \lambda) = \frac{P(i_t = q_i, O \mid \lambda)}{P(O \mid \lambda)}$$

由前向概率 $\alpha_t(i)$ 和后向概率 $\beta_t(i)$ 定义可知：

$$\alpha_t(i)\beta_t(i) = P(i_t = q_i, O \mid \lambda)$$

于是得到：

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

2．给定模型 λ 和观测 O，在时刻 t 处于状态 q_i 且在时刻 $t+1$ 处于状态 q_j 的概率．记

$$\xi_t(i, j) = P(i_t = q_i, i_{t+1} = q_j \mid O, \lambda) \tag{10.25}$$

可以通过前向后向概率计算：

$$\xi_t(i, j) = \frac{P(i_t = q_i, i_{t+1} = q_j, O \mid \lambda)}{P(O \mid \lambda)} = \frac{P(i_t = q_i, i_{t+1} = q_j, O \mid \lambda)}{\sum_{i=1}^{N}\sum_{j=1}^{N} P(i_t = q_i, i_{t+1} = q_j, O \mid \lambda)}$$

而

$$P(i_t = q_i, i_{t+1} = q_j, O \mid \lambda) = \alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)$$

所以

$$\xi_t(i, j) = \frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)} \tag{10.26}$$

$$a_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

$$b_j(k) = \frac{\sum_{t=1, o_t=v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)}$$

$$\pi_i = \gamma_1(i)$$

以下是baum-welch算法的python实现（参考了hmm库的源码）：

```
#计算公式中的beita二维数组
def backward(observationsSeq):
    T = len(observationsSeq)
    N = len(self.pi)
    beta = np.zeros((T,N),dtype=float)
    beta[T-1,:] = 1
    for t in reversed(range(T-1)):
        for n in range(N):
            beta[t,n] = np.sum(A[n,:] * B[:,observationsSeq[t+1]] * beta[t+1,:])
    return beta

#前后向算法学习参数
def baumWelch(observationsSeq,criterion=0.001):
    T = len(observationsSeq)
    N = len(pi)

    while True:
        # alpha_t(i) = P(O_1 O_2 ... O_t, q_t = S_i | hmm)
        # Initialize alpha
        alpha = forward(observationsSeq)

        # beta_t(i) = P(O_t+1 O_t+2 ... O_T | q_t = S_i , hmm)
        # Initialize beta
        beta = backward(observationsSeq)

        #根据公式求解XIt(i,j) = P(qt=Si,qt+1=Sj | O,λ)
        xi = np.zeros((T-1,N,N),dtype=float)
        for t in range(T-1):
            denominator = np.sum( np.dot(alpha[t,:],A) * B[:,observationsSeq[t+1]] * b
            for i in range(N):
                molecular = alpha[t,i] * A[i,:] * B[:,observationsSeq[t+1]] * beta[t+1
                xi[t,i,:] = molecular / denominator

```

```
        #根据xi就可以求出gamma，注意最后缺了一项要单独补上来
        gamma = np.sum(xi,axis=2)
        prod = (alpha[T-1,:] * beta[T-1,:])
        gamma = np.vstack((gamma, prod /np.sum(prod)))

        newpi = gamma[0,:]
        newA = np.sum(xi,axis=0) / np.sum(gamma[:-1,:],axis=0).reshape(-1,1)
        newB = np.zeros(B.shape,dtype=float)

        for k in range(self.B.shape[1]):
            mask = observationsSeq == k
            newB[:,k] = np.sum(gamma[mask,:],axis=0) / np.sum(gamma,axis=0)

        if np.max(abs(pi - newpi)) < criterion and \
                                np.max(abs(A - newA)) < criterion and \
                                np.max(abs(B - newB)) < criterion:
            break

        A,B,pi = newA,newB,newpi
        return A,B,pi
```

测验如下：

```
>> observations_data=['5','8','4','2','4']
>> A,B,PI=h.baumWelch(observations_data)
>> prob,path=h.viterbi(observations_data)

>> states_data=['D6','D8','D4','D4','D6']
>>A=[[ 0.15954416  0.23931624  0.6011396 ]
     [ 0.30769231  0.46153846  0.23076923]
     [ 0.23792487  0.3568873   0.40518784]]
>>B=[[ 0.          0.20588235  0.          0.41176471  0.38235294  0.          0.
   0.        ]
     [ 0.          0.33333333  0.          0.66666667  0.          0.          0.
   0.        ]
     [ 0.          0.10880829  0.          0.21761658  0.20207254  0.          0.
   0.47150259]]
>>PI=[ 0.57142857  0.          0.42857143]
```

在机器学习中，BW算法学习现有观察序列来生成模型的应用很广泛，如词类标注、语音识别、句子切分、字素音位转换、局部句法剖析、语块分析、命名实体识别、信息抽取等。回到赌场的例子，如果我是一个想耍小聪明的赌场老板，想让某种结果序列出现得格外多，怎么做呢？就可以利用BW算法得出应该怎样设计骰子和黑箱内的掷骰子机制。

三.HMM模型在中文分词的应用

- **概念明确**

我们还是关注建立HMM模型的五元组，其中：

- 观察序列K：观察序列是给定的句子。而观测集合长度则是汉字总数，包括标点，当然常用的仅3500字
- 状态集合S：由四种状态构成：词头F、词中M、词尾E、单字W
- 转移矩阵A：4*4维的矩阵
- 发射矩阵B：4*M的矩阵，表示了在某词性下任何字出现的概率
- 初始向量K：句子的第一个字的词性概率，当然，仅可能是F或W

- **训练方法**

现在假设手头的训练语料，假设是已经分好词了，也就是目前观测序列O对应的隐藏序列S我们已经知道了，那么就不需要用BM来求模型参数了，直接用最大似然即可，其实就是统计词频并除以总数构成参数；而如果是没分好词的，就要用到上面的学习方法，也就是用BW算法来学习。当然，凭直觉我们会发现，如果直接上EM，机器会倾向于将每个词都当成单字W，这是无意义的。所以实际训练时往往采用监督学习和非监督学习相结合的方式：先用人工标记的数据获得预训练参数，也就是监督学习，再在预训练参数下传入未分词的句子，用EM寻找当前最优解。直观来看，就是上下文中，相邻的字同时出现的次数越多，就越可能构成一个词。因此字与字相邻出现的概率或频率能用来反映词的可信度。

所以说，训练实际上是在有大量O（其中部分知道对应的S）的情况下，利用最大似然方法反推模型参数的问题。而分词，则是利用Viterbi算法，对一个序列O求其最大概然S。

在这里我仅代码实现了监督学习的部分，也就是通过下载已经分好词的语料库直接统计参数。后面我将会逐步实现与非监督学习的融合，比如以某个固定epoch为周期。

首先处理语料：

```
def read_corpus_from_file(cls, file_path):
    #读取语料
    f = open(file_path, 'r')
    lines = f.readlines()
    for line in lines:
        cls._words.extend([word for word in line.decode('gbk').strip().split(' ') if w
    f.close()
```

统计生成隐藏序列状态：

```
def word_to_states(cls, word):
    #词对应状态转换
    word_len = len(word)
    if word_len == 1:
        cls._states.append('W')
    else:
        state = ['M'] * word_len
        state[0] = 'F'
        state[-1] = 'E'
        cls._states.append(''.join(state))
```

初始化A,B和K

```
def cal_state(cls):
    #计算三类状态概率
    for word in cls._words:
        cls.word_to_states(word)
    init_state = cls.cal_init_state()
    trans_state = cls.cal_trans_state()
    emit_state = cls.cal_emit_state()
    cls.save_state(init_state, trans_state, emit_state)
```

利用上一部分完成的hmm类，代入

```
h = HMM(A=trans_state,B=emit_state,pi=init_state)
```

然后就可以用Viterbi算法预测了（函数内简单处理隐藏序列至'/'，使分词易于查看）：

```
>> h.viterbi("明天清晨在交大胡法光体育场和我一起学习健身操")
>> "明天/ 清晨/ 在/ 交大/ 胡法光/ 体育场/ 和/ 我/ 一起/ 学习/ 健身操"

>> h.viterbi("床前明月光，疑是地上霜")
>> "床前/ 明月光/ ，/ 疑是/ 地上/ 霜"
```

当然，中文分词的算法还有许多种，比如二阶马尔科夫模型，条件随机场模型（crf）等等。其中crf是最大熵模型的拓展，同时也HMM的条件化求解，在处理未登录词上表现往往更好，比如网上一个例子：

```python
import genius

text = "深夜的穆赫兰道发生一桩车祸，女子丽塔在车祸中失忆了"
seg_list = genius.seg_text(text)
print('/'.join([w.text for w in seg_list]))
# 深夜/的/穆赫兰道/发生/一/桩/车祸/，/女子/丽塔/在/车祸/中/失忆/了  [CRF]
# 深夜/的/穆赫/兰/道/发生/一/桩/车祸/，/女子/丽塔/在/车祸/中/失忆/了  [2-HMM]
# 深夜/的/穆赫兰道/发生/一桩/车祸/，/女子丽塔/在/车祸/中/失忆/了  [HMM]
```

四.结语

隐马尔科夫模型是我在学机器学习时了解到的一个模型，深入探寻，发现非常有趣。

一方面，它的算法（或者说整个机器学习的思想），大部分都是建立在这学期学的两门课上：《概率论与随机过程》和《线性优化与凸优化》，可以见得，必须将数理基础打牢才能在AI方向有大的进展。

另一方面，HMM模型在生活中又是如此常用，这里介绍的NLP中的中文分词只是极小的一个例子。再比如我们平时推理时会不自觉用到，也可以用来分析并预测股票涨跌，金融中的风险评估，NLP中的语音识别...尽管马尔科夫过程只是一个理论假设，但人们一步步丰富模型的外延，将其具象化，理论假设就也能很好地符合现实，为人类发挥作用。这正是数学的魅力所在。

- **参考资料**

  - 《统计学习方法》，李航
  - 一文搞懂HMM（隐马尔可夫模型）[https://www.cnblogs.com/skyme/p/4651331.html]
  - HMM超详细讲解 [https://blog.csdn.net/continueoo/article/details/77893587]
  - 从最大似然到EM算法浅解 [https://blog.csdn.net/zouxy09/article/details/8537620/]
  - 机器学习：HMM隐马尔可夫模型用于中文分词 [https://blog.csdn.net/ztf312/article/details/50982529]