AI学习笔记--skLearn--HMM

- 概述

隐马尔可夫模型（Hidden Markov Model，HMM）是统计模型，它用来描述一个含有隐含未知参数的马尔可夫过程。其难点是从可观察的参数中确定该过程的隐含参数。然后利用这些参数来作进一步的分析，例如模式识别。
隐马尔可夫模型（Hidden Markov Model，HMM）作为一种统计分析模型，创立于20世纪70年代。80年代得到了传播和发展，成为信号处理的一个重要方向，现已成功地用于语音识别，行为识别，文字识别以及故障诊断等领域。

一种HMM可以呈现为最简单的动态贝叶斯网络。隐马尔可夫模型背后的数学是由LEBaum和他的同事开发的。它与早期由RuslanL.Stratonovich提出的最优非线性滤波问题息息相关，他是第一个提出前后过程这个概念的。
在简单的马尔可夫模型（如马尔可夫链），所述状态是直接可见的观察者，因此状态转移概率是唯一的参数。在隐马尔可夫模型中，状态是不直接可见的，但输出依赖于该状态下，是可见的。每个状态通过可能的输出记号有了可能的概率分布。因此，通过一个HMM产生标记序列提供了有关状态的一些序列的信息。注意，“隐藏”指的是，该模型经其传递的状态序列，而不是模型的参数；即使这些参数是精确已知的，我们仍把该模型称为一个“隐藏”的马尔可夫模型。隐马尔可夫模型以它在时间上的模式识别所知，如语音，手写，手势识别，词类的标记，乐谱，局部放电和生物信息学应用。

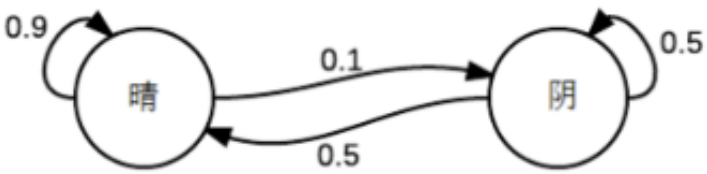
隐马尔可夫模型可以被认为是一个概括的混合模型中的隐藏变量（或变量），它控制的混合成分被选择为每个观察，通过马尔可夫过程而不是相互独立相关。最近，隐马尔可夫模型已推广到两两马尔可夫模型和三重态马尔可夫模型，允许更复杂的数据结构的考虑和非平稳数据建模。

- 算法解析

每个状态的转移只依赖于之前的n个状态，这个过程被称为1个n阶的模型，其中n是影响转移状态的数目。最简单的马尔科夫过程就是一阶过程，每一个状态的转移只依赖于其之前的那一个状态。用数学表达式表示就是：

$$P(X_{n+1}=x|X_1=x_1,X_2=x_2,...,X_n=x_n)=P(X_{n+1}=x|X_n=x_n).$$

假设天气服从马尔科夫链如下：



$$P=\begin{pmatrix}0.9 & 0.1\\ 0.5 & 0.5\end{pmatrix}$$

如果假定今天是晴天，那么明天天气是晴天的概率则可以通过线性代数计算：

$$(1 \quad 0) * \begin{bmatrix} 0.9 & 0.1 \\ 0.5 & 0.5 \end{bmatrix} = (0.9 \quad 0.1)$$

计算得到，如果明天是晴天概率为0.9，为阴天的概率为0.1。按照上述的假设，可以推测出从今天开始的今后一天，阴天和晴天的概率分布情况：

$$q = \lim_{n\to\infty}(1 \quad 0)P^n = (0.833\ 0.167)$$

如果假定n无穷大，由公式：

$$P^{\infty} = \begin{pmatrix} 0.833 & 0.167 \\ 0.833 & 0.167 \end{pmatrix}$$

$$(1\ 0)P^{\infty} = (0\ 1)P^{\infty}$$

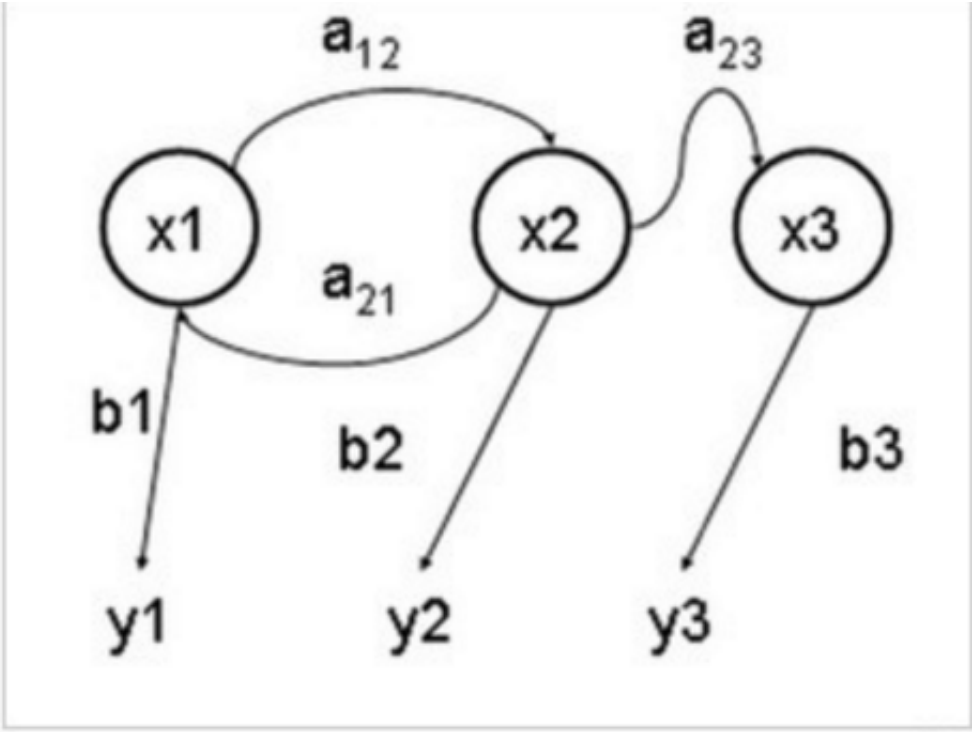
可以得到，不管今天的天气是晴天还是阴天，在多日以后的天气分布收敛到一个固定的分布情况，这种固定的分布情况就叫做稳态分布。在转移状态不变的前提下，很久的未来，每一天的天气情况都 满足q=(0.833 0.167) 的一个样本点。至此，一阶HMM过程定义以下三部分：

1. 状态： 晴天、阴天
2. 初始向量： 定义系统在时间为0的时候状态的概率
3. 状态转移矩阵：每种天气转换的概率

所有可以被这种模型描述的系统，都可以是一个马尔科夫过程。

马尔科夫链存在缺陷

这种数学模型存在前后关系缺失，也带来了信息的缺失。比如股市，如果只是观测市场，我们只能知道当天的价格、成交量等信息，但是并不知道当前股市处于什么样的状态（牛市、熊市、震荡或者反弹等），在这种情况下，我们有两个状态集合，一个可以观察到的状态集合和一个隐式的状态集合。希望依据股市价格、成交量和马尔科夫假设来预测股市的状态。可以观察到状态序列和隐藏的状态序列是概率相关的。我们可以用这种类型的过程建模为一个隐藏的马尔科夫过程和一个与这个隐藏马尔科夫过程概率相关的并且可以观测到的状态集合，这就是马尔科夫模型。



隐马尔可夫模型状态变迁图（例子）

x — 隐含状态

y — 可观察的输出

a — 转换概率（transition probabilities）

b — 输出概率（output probabilities）

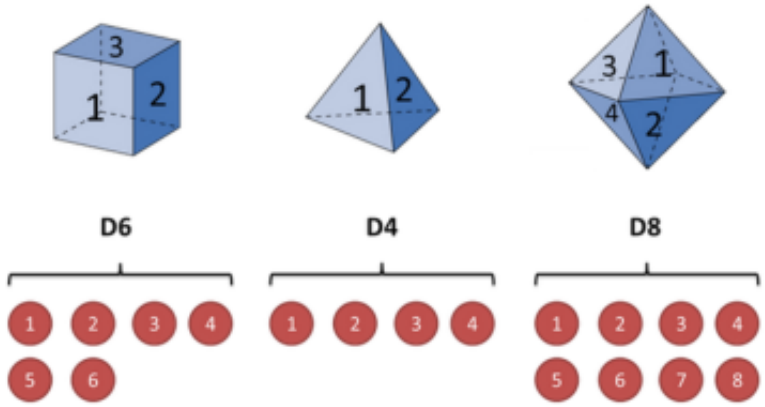
隐马尔科夫链

隐马尔可夫模型（Hidden Markov Model，HMM）是统计模型，它用来描述一个含有隐含未知参数的马尔可夫过程。其难点是从可观察的参数中确定该过程的隐含参数。然后利用这些参数来作进一步的分析，例如模式识别。举一个简单的例子：

第一个骰子6个面（称这个骰子为D6），每个面（1，2，3，4，5，6）出现的概率是1/6。

第二个骰子是个四面体（称这个骰子为D4），每个面（1，2，3，4）出现的概率是1/4。

第三个骰子有八个面（称这个骰子为D8），每个面（1，2，3，4，5，6，7，8）出现的概率是1/8。

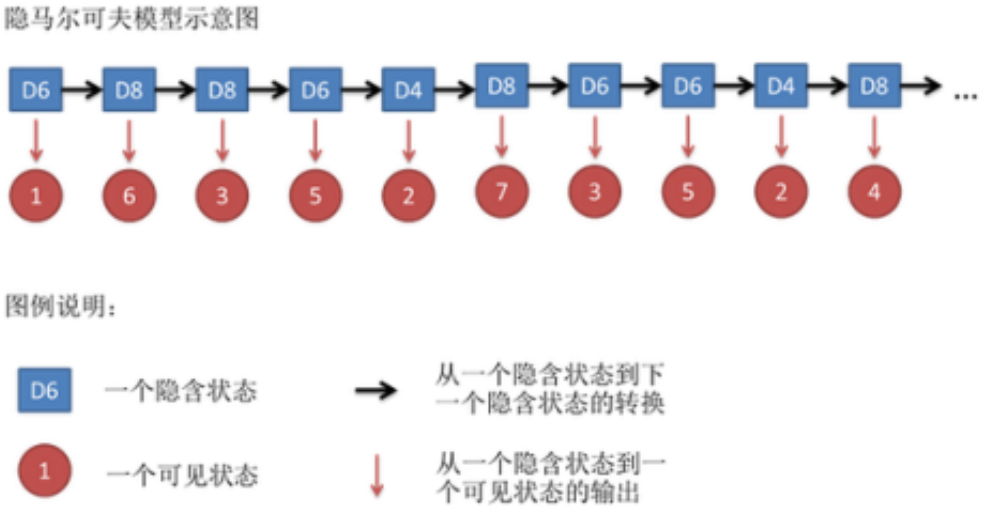


假设我们开始掷骰子，我们先从三个骰子里挑一个，挑到每一个骰子的概率都是1/3。然后我们掷骰子，得到一个数字，1，2，3，4，5，6，7，8中的一个。不停的重复上述过程，我们会得到一串数字，每个数字都是1，2，3，4，5，6，7，8中的一个。例如我们可能得到这么一串数字（掷骰子10次）：1 6 3 5 2 7 3 5 2 4

这串数字叫做可见状态链。但是在隐马尔可夫模型中，我们不仅仅有这么一串可见状态链，还有一串隐含状态链。在这个例子里，这串隐含状态链就是你用的骰子的序列。比如，隐含状态链有可能是：D6 D8 D8 D6 D4 D8 D6 D6 D4 D8

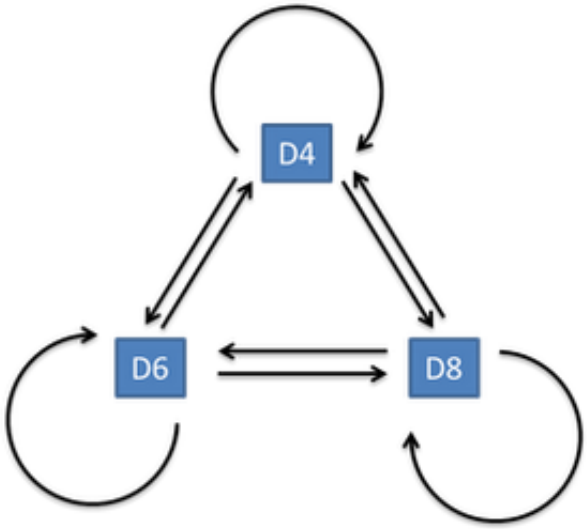
一般来说，HMM中说到的马尔可夫链其实是指隐含状态链，因为隐含状态（骰子）之间存在转换概率（transition probability）。转换概率可能不同。

可见状态之间没有转换概率，但是隐含状态和可见状态之间有一个概率叫做输出概率（emission probability）。就我们的例子来说，六面骰（D6）产生1的输出概率是1/6。



隐含状态关系示意图

隐含状态转换关系示意图

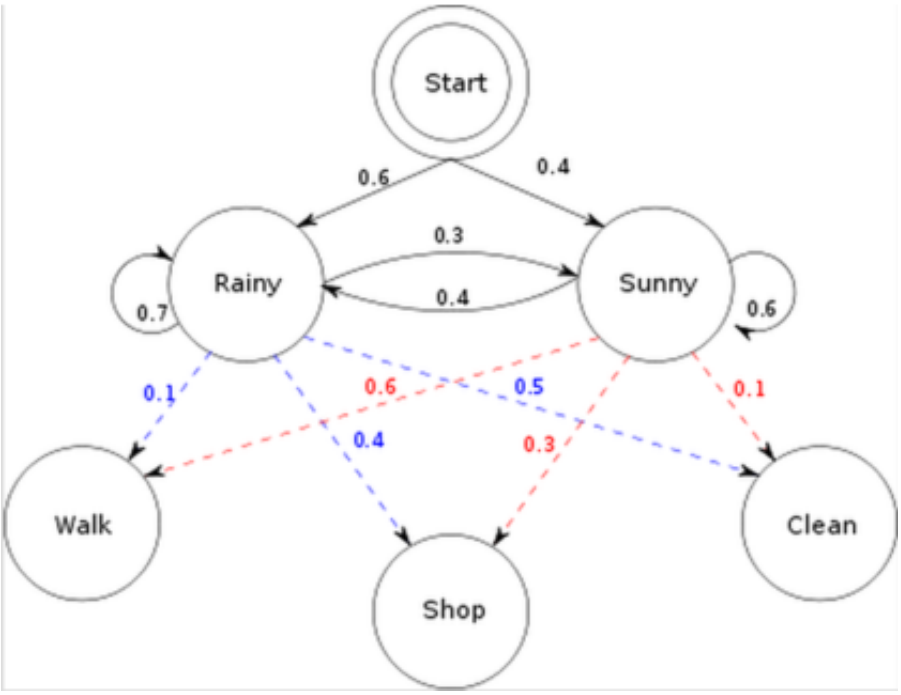


和HMM模型相关的算法主要分为三类，分别解决三种问题：

1. 知道骰子有几种（隐含状态数量），每种骰子是什么（转换概率），根据掷骰子掷出的结果（可见状态链），我想知道每次掷出来的都是哪种骰子（隐含状态链）。 在语音识别领域呢，叫做解码问题。
2. 还是知道骰子有几种（隐含状态数量），每种骰子是什么（转换概率），根据掷骰子掷出的结果（可见状态链），我想知道掷出这个结果的概率。 -反欺诈。
3. 知道骰子有几种（隐含状态数量），不知道每种骰子是什么（转换概率），观测到很多次掷骰子的结果（可见状态链），我想反推出每种骰子是什么（转换概率）。

隐马尔科夫案例

一个东京的朋友每天根据天气{下雨，天晴}决定当天的活动{公园散步,购物,清理房间}中的一种，我每天只能在twitter上看到她发的推“啊，我前天公园散步、昨天购物、今天清理房间了！”，那么我可以根据她发的推特推断东京这三天的天气。在这个例子里，显状态是活动，隐状态是天气。



对应上面的三个问题，可以提出以下几个问题：

1. 已知整个模型，我观测到连续三天做的事情是：散步，购物，收拾，我想猜，这三天的天气是怎么样的。
2. 已知整个模型，我观测到连续三天做的事情是：散步，购物，收拾。那么根据模型，计算产生这些行为的概率是多少。
3. 最复杂的，我只知道这三天做了这三件事儿，而其他什么信息都没有。我得建立一个模型，晴雨转换概率，第一天天气情况的概率分布，根据天气情况选择做某事的概率分布。

针对上述的三个问题，我们可以套用不同的算法去解释这些。

1. 问题一：Viterbi Algorithm，维特比算法。
2. 问题二：Forward Algorithm，向前算法，或者 Backward Algorithm，向后算法。
3. 问题三：Baum-Welch Algorithm，鲍姆-韦尔奇算法

Viterbi Algorithm

找到概率最大的隐含序列，需要计算 $argmax_Q P(Q|O,\lambda)$argmaxQP(Q|O,λ) ，Viterbi算法（动态规划）定义 $\delta_j(t)$δj(t) 是保存在时间t以内，观测值和状态 S_j 的最大可能性。可以归纳为公式：

$$\delta_j(1) = \pi_j b_{jO_1} \qquad 1 \le j \le N$$
$$\delta_j(t+1) = \left(\max_i \delta_i(t) a_{ij}\right) b_{jO_{t+1}} \qquad 1 \le t \le T-1$$

最后使用回溯（backtracking）（保持最大参数对于每个t和j）来发现最优解。

问题2： 遍历算法

三天的天气共有6种可能性，需要遍历所有存在的可能性，计算产生此种观测值的概率。操作的步骤是：

- 需要计算 $P(O|\lambda)$，给出模型 λ 的条件下，可见序列为 $O=O_1O_2...O_T$的可能性。
- 我们能够枚举所有可能的隐含状态序列$Q=q_1q_2...q_T$
- 对于其中一个隐含状态序列Q：

$$P(O|Q,\lambda) = \prod_{t=1}^{T} P(O_t|q_t,\lambda) = \prod_{t=1}^{T} b_{q_t O_t}$$

- 对于这个隐含状态序列 Q 的可能性是：

$$P(Q|\lambda) = P(q_1)\prod_{t=2}^{T} P(q_t|q_{t-1}) = \pi_{q_1}\prod_{t=2}^{T} a_{q_{t-1}q_t}$$

- 因此联合可能性为：

$$P(O,Q|\lambda) = P(Q|\lambda)P(O|Q,\lambda) = \pi_{q_1}\prod_{t=2}^{T} a_{q_{t-1}q_t}\prod_{t=1}^{T} b_{q_t O_t}$$

- 考虑所有可能隐含状态序列

$$P(O|\lambda) = \sum_{Q} \pi_{q_1} b_{q_1 O_1}\prod_{t=2}^{T} a_{q_{t-1}q_t} b_{q_t O_t}$$

对于上述的模型，存在很多算法缺陷：

- 需要 $2TN_T$ 的计算量
- N_T的可能的隐含状态序列
- 每个序列需约2T的计算量

所以可以使用Forword 算法。

Forward Algorithm

定义前向变量 $\alpha_j(t)$ 作为 直到时间t ,和隐藏状态S_j下，部分可见序列的可能性：

$$\alpha_j(t) = P(O_1 O_2 ... O_t, q_t = S_j|\lambda)$$

可用归纳计算：

$$\alpha_j(1) = \pi_j b_{jO_1} \qquad 1 \le j \le N$$

$$\alpha_j(t+1) = \left(\sum_{i=1}^{N} \alpha_i(t) a_{ij}\right) b_{jO_{t+1}} \qquad 1 \le t \le T-1$$

通过 $N2T$复杂度的计算：

$$P(O|\lambda) = \sum_{i=1}^{N} P(O, q_T = S_i|\lambda) = \sum_{i=1}^{N} \alpha_i(T)$$

Backward Algorithm

定义后向变量 $\beta_i(t)$ 作为时间t之后，给定t时刻的隐藏状态S_j下，部分可见序列的可能性：

$$\beta_i(t) = P(O_{t+1} Ot + 2 ... O_T|q_t = S_i, \lambda)$$

可用归纳计算：

$$\beta_i(T) = 1 \qquad 1 \le i \le N$$

$$\beta_i(t-1) = \sum_{j=1}^{N} a_{ij} b_{jO_t} \beta_j(t) \qquad 2 \le t \le T$$

Baum-Welch Algorithm

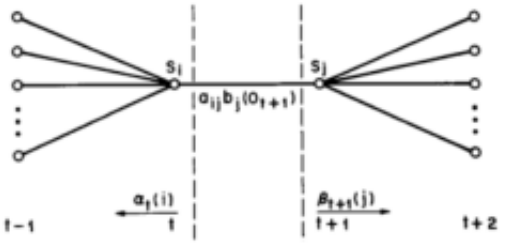
- There is no known way to analytically solve for the model which maximizes the probability of the observation sequence
- We can choose $\lambda = (A, B, \pi)$ which locally maximizes $P(O|\lambda)$
  - gradient techniques
  - Baum-Welch reestimation (equivalent to EM)
- We need to define $\xi_{ij}(t)$, i.e., the probability of being in state S_i at time t and in state S_j at time $t+1$

$$\xi_{ij}(t) = P(q_t = S_i, q_{t+1} = S_j | O, \lambda)$$

$$\xi_{ij}(t) = \frac{\alpha_i(t) a_{ij} b_{jO_{t+1}} \beta_j(t+1)}{P(O|\lambda)} =$$

$$= \frac{\alpha_i(t) a_{ij} b_{jO_{t+1}} \beta_j(t+1)}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i(t) a_{ij} b_{jO_{t+1}} \beta_j(t+1)}$$

Figure: Operations for computing the $\xi_{ij}(t)$



- Recall that $\gamma_i(t)$ is a probability of state S_i at time t, hence

$$\gamma_i(t) = \sum_{j=1}^{N} \xi_{ij}(t)$$

- Now if we sum over the time index t
  - $\sum_{t=1}^{T-1} \gamma_i(t)$ = expected number of times that S_i is visited*
    = expected number of transitions from state S_i
  - $\sum_{t=1}^{T-1} \xi_{ij}(t)$ = expected number of transitions from S_i to S_j

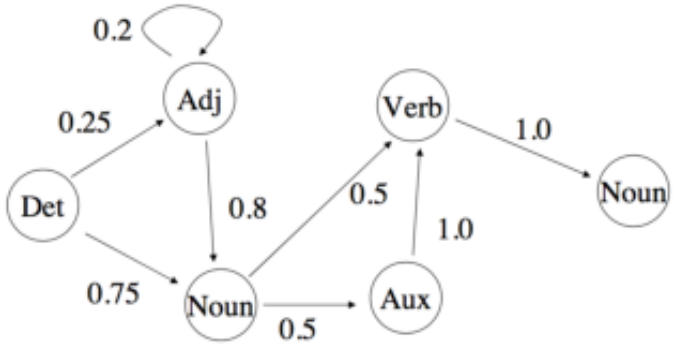
- Reestimation formulas

$$\bar{\pi}_i = \gamma_i(1) \qquad \bar{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)} \qquad \bar{b}_{jk} = \frac{\sum_{O_t = v_k} \gamma_j(t)}{\sum_{t=1}^{T} \gamma_j(t)}$$

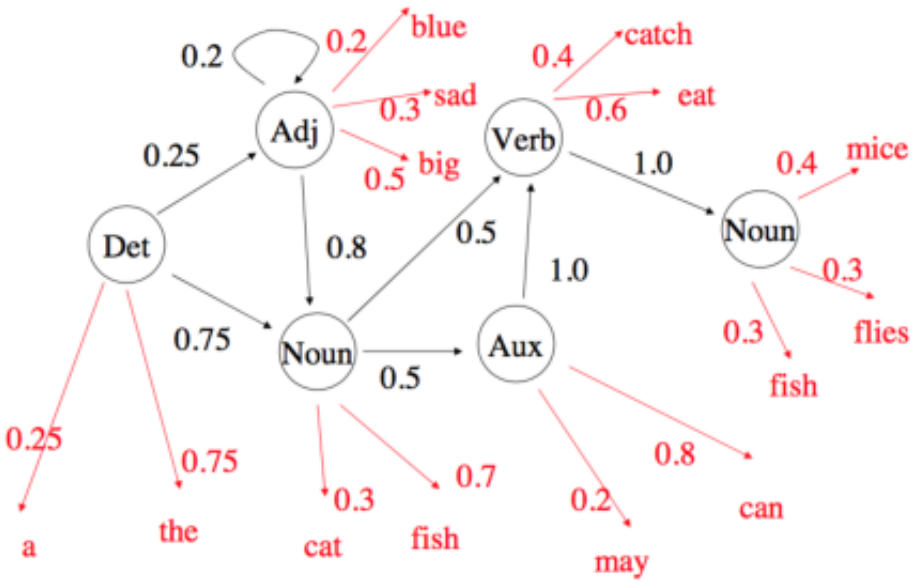
- Baum et al. proved that if current model is $\lambda = (A, B, \pi)$ and we use the above to compute $\bar{\lambda} = (\bar{A}, \bar{B}, \bar{\pi})$ then either
  - $\bar{\lambda} = \lambda$ – we are in a critical point of the likelihood function
  - $P(O|\bar{\lambda}) > P(O|\lambda)$ – model $\bar{\lambda}$ is more likely
- If we iteratively reestimate the parameters we obtain a maximum likelihood estimate of the HMM
- Unfortunately this finds a local maximum and the surface can be very complex

Algorithm 总结

HMM本质是一个有转换概率的有限状态机模型。



一个HMM模型，在特定的状态下会有一定概率发出标记信号：



我们想要计算$P(t_1,...,t_n|w_1,...,w_n)$的概率，依据贝叶斯理论，可以得出公式：

$$P(t_1 \dots t_N|w_1 \dots w_N) = \frac{P(t_1 \dots t_N) \times P(w_1 \dots w_N|t_1 \dots t_N)}{P(w_1 \dots w_N)}$$

通过公式计算出最大的序列t，分母都一样只要分子大就可以得到对应的概率。一般需要遵循几个原则：

- 链式法则：

$$\begin{aligned} P(A,B) &= P(A|B) \times P(B) \\ P(A,B,C) &= P(A|B,C) \times P(B|C) \times P(C) \\ P(A,B,C,D) &= P(A|B,C,D) \times P(B|C,D) \times P(C|D) \times P(D) \\ P(C_1 \dots C_n) &= P(C_1|C_2 \dots C_n) \times P(C_2|C_3 \dots C_n) \times \dots \times P(C_n) \end{aligned}$$

- 标注转移分布：

$$P(t_1 \dots t_N) = \prod_{i=1}^{N} P(t_i| \dots t_{i-1} \dots t_1) \approx \prod_{i=1}^{N} P(t_i|t_{i-1})$$

- 发射词分布：

$$P(w_1 \dots w_N | t_1 \dots t_N) = \prod_{i=1}^{N} P(w_i | w_{i-1} \dots w_1, t_1 \dots t_N)$$
$$\approx \prod_{i=1}^{N} P(w_i | t_i)$$

- 算法Demo

掷筛子问题： 主类hmm.py

```
# -*- coding:utf-8 -*-
# 隐马尔科夫链模型
# By tostq <tostq216@163.com>
# 博客: blog.csdn.net/tostq
import numpy as np
from math import pi,sqrt,exp,pow,log
from numpy.linalg import det, inv
from abc import ABCMeta, abstractmethod
from sklearn import cluster

class _BaseHMM():
    """
    基本HMM虚类，需要重写关于发射概率的相关虚函数
    n_state : 隐藏状态的数目
    n_iter : 迭代次数
    x_size : 观测值维度
    start_prob : 初始概率
    transmat_prob : 状态转换概率
    """
    __metaclass__ = ABCMeta  # 虚类声明

    def __init__(self, n_state=1, x_size=1, iter=20):
        self.n_state = n_state
        self.x_size = x_size
        self.start_prob = np.ones(n_state) * (1.0 / n_state)  # 初始状态概率
        self.transmat_prob = np.ones((n_state, n_state)) * (1.0 / n_state)
# 状态转换概率矩阵
        self.trained = False # 是否需要重新训练
        self.n_iter = iter  # EM训练的迭代次数

    # 初始化发射参数
    @abstractmethod
    def _init(self,X):
        pass

    # 虚函数：返回发射概率
```

```
    @abstractmethod
    def emit_prob(self, x):  # 求x在状态k下的发射概率 P(X|Z)
        return np.array([0])

    # 虚函数
    @abstractmethod
    def generate_x(self, z): # 根据隐状态生成观测值x p(x|z)
        return np.array([0])

    # 虚函数：发射概率的更新
    @abstractmethod
    def emit_prob_updated(self, X, post_state):
        pass

     # 通过HMM生成序列
    def generate_seq(self, seq_length):
        X = np.zeros((seq_length, self.x_size))
        Z = np.zeros(seq_length)
        Z_pre = np.random.choice(self.n_state, 1, p=self.start_prob)  # 采样
初始状态
        X[0] = self.generate_x(Z_pre) # 采样得到序列第一个值
        Z[0] = Z_pre

        for i in range(seq_length):
            if i == 0: continue
              # P(Zn+1)=P(Zn+1|Zn)P(Zn)
            Z_next = np.random.choice(self.n_state, 1,
p=self.transmat_prob[Z_pre,:][0])
            Z_pre = Z_next
            # P(Xn+1|Zn+1)
            X[i] = self.generate_x(Z_pre)
            Z[i] = Z_pre

        return X,Z

    # 估计序列X出现的概率
    def X_prob(self, X, Z_seq=np.array([])):
        # 状态序列预处理
        # 判断是否已知隐藏状态
        X_length = len(X)
        if Z_seq.any():
            Z = np.zeros((X_length, self.n_state))
            for i in range(X_length):
                Z[i][int(Z_seq[i])] = 1
        else:
            Z = np.ones((X_length, self.n_state))
        # 向前向后传递因子
        _, c = self.forward(X, Z)  # P(x,z)
        # 序列的出现概率估计
        prob_X = np.sum(np.log(c))  # P(X)
```

```
        return prob_X

    # 已知当前序列预测未来（下一个）观测值的概率
    def predict(self, X, x_next, Z_seq=np.array([]), istrain=True):
        if self.trained == False or istrain == False:  # 需要根据该序列重新训
练
            self.train(X)

        X_length = len(X)
        if Z_seq.any():
            Z = np.zeros((X_length, self.n_state))
            for i in range(X_length):
                Z[i][int(Z_seq[i])] = 1
        else:
            Z = np.ones((X_length, self.n_state))
        # 向前向后传递因子
        alpha, _ = self.forward(X, Z)  # P(x,z)
        prob_x_next =
self.emit_prob(np.array([x_next]))*np.dot(alpha[X_length -
1],self.transmat_prob)
        return prob_x_next

    def decode(self, X, istrain=True):
        """
        利用维特比算法，已知序列求其隐藏状态值
        :param X: 观测值序列
        :param istrain: 是否根据该序列进行训练
        :return: 隐藏状态序列
        """
        if self.trained == False or istrain == False:  # 需要根据该序列重新训
练
            self.train(X)

        X_length = len(X)  # 序列长度
        state = np.zeros(X_length)  # 隐藏状态

        pre_state = np.zeros((X_length, self.n_state))  # 保存转换到当前隐藏状
态的最可能的前一状态
        max_pro_state = np.zeros((X_length, self.n_state))  # 保存传递到序列某
位置当前状态的最大概率

        _,c=self.forward(X,np.ones((X_length, self.n_state)))
        max_pro_state[0] = self.emit_prob(X[0]) * self.start_prob *
(1/c[0]) # 初始概率

        # 前向过程
        for i in range(X_length):
            if i == 0: continue
              for k in range(self.n_state):
                prob_state = self.emit_prob(X[i])[k] *
```

```
self.transmat_prob[:,k] * max_pro_state[i-1]
                max_pro_state[i][k] = np.max(prob_state)* (1/c[i])
                pre_state[i][k] = np.argmax(prob_state)

        # 后向过程
        state[X_length - 1] = np.argmax(max_pro_state[X_length - 1,:])
        for i in reversed(range(X_length)):
            if i == X_length - 1: continue
              state[i] = pre_state[i + 1][int(state[i + 1])]

        return  state

    # 针对于多个序列的训练问题
    def train_batch(self, X, Z_seq=list()):
        # 针对于多个序列的训练问题，其实最简单的方法是将多个序列合并成一个序列，而唯一需要调整的是初始状态概率
        # 输入X类型：list(array)，数组链表的形式
        # 输入Z类型：list(array)，数组链表的形式，默认为空列表（即未知隐状态情况）
        self.trained = True
        X_num = len(X) # 序列个数
        self._init(self.expand_list(X)) # 发射概率的初始化

        # 状态序列预处理，将单个状态转换为1-to-k的形式
        # 判断是否已知隐藏状态
        if Z_seq==list():
            Z = []  # 初始化状态序列list
            for n in range(X_num):
                Z.append(list(np.ones((len(X[n]), self.n_state))))
        else:
            Z = []
            for n in range(X_num):
                Z.append(np.zeros((len(X[n]),self.n_state)))
                for i in range(len(Z[n])):
                    Z[n][i][int(Z_seq[n][i])] = 1

        for e in range(self.n_iter):  # EM步骤迭代
            # 更新初始概率过程
            #  E步骤
            print "iter: ", e
            b_post_state = []  # 批量累积：状态的后验概率，类型list(array)
            b_post_adj_state = np.zeros((self.n_state, self.n_state)) # 批量累积：相邻状态的联合后验概率，数组
            b_start_prob = np.zeros(self.n_state) # 批量累积初始概率
            for n in range(X_num): # 对于每个序列的处理
                X_length = len(X[n])
                alpha, c = self.forward(X[n], Z[n])  # P(x,z)
                beta = self.backward(X[n], Z[n], c)  # P(x|z)

                post_state = alpha * beta / np.sum(alpha * beta) # 归一化!
                b_post_state.append(post_state)
```

```
                post_adj_state = np.zeros((self.n_state, self.n_state))  #
相邻状态的联合后验概率
                for i in range(X_length):
                    if i == 0: continue
                      if c[i]==0: continue
                      post_adj_state += (1 / c[i]) * np.outer(alpha[i -
1],
                                                             beta[i] *
self.emit_prob(X[n][i])) * self.transmat_prob

                if np.sum(post_adj_state)!=0:
                    post_adj_state = post_adj_state/np.sum(post_adj_state)
# 归一化!
                b_post_adj_state += post_adj_state  # 批量累积：状态的后验概率
                b_start_prob += b_post_state[n][0] # 批量累积初始概率

            # M步骤，估计参数，最好不要让初始概率都为0出现，这会导致alpha也为0
            b_start_prob += 0.001*np.ones(self.n_state)
            self.start_prob = b_start_prob / np.sum(b_start_prob)
            b_post_adj_state += 0.001
            for k in range(self.n_state):
                if np.sum(b_post_adj_state[k])==0: continue
                  self.transmat_prob[k] = b_post_adj_state[k] /
np.sum(b_post_adj_state[k])

            self.emit_prob_updated(self.expand_list(X),
self.expand_list(b_post_state))

    def expand_list(self, X):
        # 将list(array)类型的数据展开成array类型
        C = []
        for i in range(len(X)):
            C += list(X[i])
        return np.array(C)

    # 针对于单个长序列的训练
    def train(self, X, Z_seq=np.array([])):
        # 输入X类型：array，数组的形式
        # 输入Z类型：array，一维数组的形式，默认为空列表（即未知隐状态情况）
        self.trained = True
        X_length = len(X)
        self._init(X)

        # 状态序列预处理
        # 判断是否已知隐藏状态
        if Z_seq.any():
            Z = np.zeros((X_length, self.n_state))
            for i in range(X_length):
                Z[i][int(Z_seq[i])] = 1
        else:
```

```
            Z = np.ones((X_length, self.n_state))

        for e in range(self.n_iter):  # EM步骤迭代
            # 中间参数
            print e, " iter"
              # E步骤
            # 向前向后传递因子
            alpha, c = self.forward(X, Z)  # P(x,z)
            beta = self.backward(X, Z, c)  # P(x|z)

            post_state = alpha * beta
            post_adj_state = np.zeros((self.n_state, self.n_state))  # 相邻
状态的联合后验概率
            for i in range(X_length):
                if i == 0: continue
                  if c[i]==0: continue
                  post_adj_state += (1 / c[i])*np.outer(alpha[i -
1],beta[i]*self.emit_prob(X[i]))*self.transmat_prob

            # M步骤，估计参数
            self.start_prob = post_state[0] / np.sum(post_state[0])
            for k in range(self.n_state):
                self.transmat_prob[k] = post_adj_state[k] /
np.sum(post_adj_state[k])

            self.emit_prob_updated(X, post_state)

    # 求向前传递因子
    def forward(self, X, Z):
        X_length = len(X)
        alpha = np.zeros((X_length, self.n_state))  # P(x,z)
        alpha[0] = self.emit_prob(X[0]) * self.start_prob * Z[0] # 初始值
        # 归一化因子
        c = np.zeros(X_length)
        c[0] = np.sum(alpha[0])
        alpha[0] = alpha[0] / c[0]
        # 递归传递
        for i in range(X_length):
            if i == 0: continue
              alpha[i] = self.emit_prob(X[i]) * np.dot(alpha[i - 1],
self.transmat_prob) * Z[i]
            c[i] = np.sum(alpha[i])
            if c[i]==0: continue
              alpha[i] = alpha[i] / c[i]

        return alpha, c

    # 求向后传递因子
    def backward(self, X, Z, c):
        X_length = len(X)
```

```
        beta = np.zeros((X_length, self.n_state))  # P(x|z)
        beta[X_length - 1] = np.ones((self.n_state))
        # 递归传递
        for i in reversed(range(X_length)):
            if i == X_length - 1: continue
              beta[i] = np.dot(beta[i + 1] * self.emit_prob(X[i + 1]),
self.transmat_prob.T) * Z[i]
            if c[i+1]==0: continue
              beta[i] = beta[i] / c[i + 1]

        return beta

# 二元高斯分布函数
def gauss2D(x, mean, cov):
    # x, mean, cov均为numpy.array类型
    z = -np.dot(np.dot((x-mean).T,inv(cov)),(x-mean))/2.0
    temp = pow(sqrt(2.0*pi),len(x))*sqrt(det(cov))
    return (1.0/temp)*exp(z)

class GaussianHMM(_BaseHMM):
    """
    发射概率为高斯分布的HMM
    参数:
    emit_means: 高斯发射概率的均值
    emit_covars: 高斯发射概率的方差
    """
    def __init__(self, n_state=1, x_size=1, iter=20):
        _BaseHMM.__init__(self, n_state=n_state, x_size=x_size, iter=iter)
        self.emit_means = np.zeros((n_state, x_size))      # 高斯分布的发射概
率均值
        self.emit_covars = np.zeros((n_state, x_size, x_size)) # 高斯分布的发
射概率协方差
        for i in range(n_state): self.emit_covars[i] = np.eye(x_size)  # 初
始化为均值为0，方差为1的高斯分布函数

    def _init(self,X):
        # 通过K均值聚类，确定状态初始值
        mean_kmeans = cluster.KMeans(n_clusters=self.n_state)
        mean_kmeans.fit(X)
        self.emit_means = mean_kmeans.cluster_centers_
        for i in range(self.n_state):
            self.emit_covars[i] = np.cov(X.T) + 0.01 * np.eye(len(X[0]))

    def emit_prob(self, x): # 求x在状态k下的发射概率
        prob = np.zeros((self.n_state))
        for i in range(self.n_state):
            prob[i]=gauss2D(x,self.emit_means[i],self.emit_covars[i])
        return prob

    def generate_x(self, z): # 根据状态生成x p(x|z)
```

```
        return np.random.multivariate_normal(self.emit_means[z]
[0],self.emit_covars[z][0],1)

    def emit_prob_updated(self, X, post_state): # 更新发射概率
        for k in range(self.n_state):
            for j in range(self.x_size):
                self.emit_means[k][j] = np.sum(post_state[:,k] *X[:,j]) /
np.sum(post_state[:,k])

            X_cov = np.dot((X-self.emit_means[k]).T, (post_state[:,k]*(X-
self.emit_means[k]).T).T)
            self.emit_covars[k] = X_cov / np.sum(post_state[:,k])
            if det(self.emit_covars[k]) == 0: # 对奇异矩阵的处理
                self.emit_covars[k] = self.emit_covars[k] +
0.01*np.eye(len(X[0]))


class DiscreteHMM(_BaseHMM):
    """
    发射概率为离散分布的HMM
    参数:
    emit_prob : 离散概率分布
    x_num: 表示观测值的种类
    此时观测值大小x_size默认为1
    """
    def __init__(self, n_state=1, x_num=1, iter=20):
        _BaseHMM.__init__(self, n_state=n_state, x_size=1, iter=iter)
        self.emission_prob = np.ones((n_state, x_num)) * (1.0/x_num)  # 初始
化发射概率均值
        self.x_num = x_num

    def _init(self, X):
        self.emission_prob = np.random.random(size=
(self.n_state,self.x_num))
        for k in range(self.n_state):
            self.emission_prob[k] =
self.emission_prob[k]/np.sum(self.emission_prob[k])

    def emit_prob(self, x): # 求x在状态k下的发射概率
        prob = np.zeros(self.n_state)
        for i in range(self.n_state): prob[i]=self.emission_prob[i]
[int(x[0])]
        return prob

    def generate_x(self, z): # 根据状态生成x p(x|z)
        return np.random.choice(self.x_num, 1, p=self.emission_prob[z][0])

    def emit_prob_updated(self, X, post_state): # 更新发射概率
        self.emission_prob = np.zeros((self.n_state, self.x_num))
        X_length = len(X)
```

```
        for n in range(X_length):
            self.emission_prob[:,int(X[n])] += post_state[n]

        self.emission_prob+= 0.1/self.x_num
        for k in range(self.n_state):
            if np.sum(post_state[:,k])==0: continue
              self.emission_prob[k] =
self.emission_prob[k]/np.sum(post_state[:,k])
```

调用类

```
# -*- coding:utf-8 -*-
# By tostq <tostq216@163.com>
# 博客: blog.csdn.net/tostq
from hmmlearn.hmm import MultinomialHMM
import numpy as np
import hmm

dice_num = 3
x_num = 8
dice_hmm = hmm.DiscreteHMM(3, 8)
dice_hmm.start_prob = np.ones(3)/3.0
dice_hmm.transmat_prob = np.ones((3,3))/3.0
dice_hmm.emission_prob = np.array([[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0,
0.0],
                                   [1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0,
0.0],
                                   [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
1.0]])
# 归一化
dice_hmm.emission_prob = dice_hmm.emission_prob /
np.repeat(np.sum(dice_hmm.emission_prob, 1),8).reshape((3,8))

dice_hmm.trained = True

X = np.array([[1],[6],[3],[5],[2],[7],[3],[5],[2],[4],[3],[6],[1],[5],[4]])
Z = dice_hmm.decode(X) # 问题A
logprob = dice_hmm.X_prob(X) # 问题B

# 问题C
x_next = np.zeros((x_num,dice_num))
for i in range(x_num):
    c = np.array([i])
    x_next[i] = dice_hmm.predict(X, i)

print "state: ", Z
print "logprob: ", logprob
print "prob of x_next: ", x_next
```

运行结果：

```
state:  [1. 2. 1. 0. 1. 2. 1. 0. 1. 0. 1. 2. 1. 0. 0.]
logprob:  -33.169958671729184
prob of x_next:  [[0.05555556 0.08333333 0.04166667]
[0.05555556 0.08333333 0.04166667]
[0.05555556 0.08333333 0.04166667]
[0.05555556 0.         0.04166667]
[0.05555556 0.         0.04166667]
[0.         0.         0.04166667]
[0.         0.         0.04166667]]
```

Demo2:

```
#coding=utf-8
'''
实现隐马尔科夫模型的基本方法，
输入为状态转移矩阵，观测矩阵，初始状态概率向量，观测序列
实现前向算法和后向算法计算观测序列出现的概率
实现维特比算法找当前观测序列下最可能的状态序列
实现在给定模型和观测下，t时刻处于p状态的概率，t,p在main函数中指定
'''
from numpy import *

class HMM:

    def __init__(self):
        #喂数据
        self.A=array([(0.5,0.2,0.3),(0.3,0.5,0.2),(0.2,0.3,0.5)])
        self.B=array([(0.5,0.5),(0.4,0.6),(0.7,0.3)])
        self.pi=array([(0.2),(0.4),(0.4)])
        self.o=[0,1,0]
        self.t=len(self.o)#观测序列长度
        self.m=len(self.A)#状态集合个数
        self.n=len(self.B[0])#观测集合个数

    def forward(self):
        #t时刻部分观测序列为o1,o2,ot,状态为qi的概率用矩阵x表示，
        #则矩阵大小行数为观测序列数，列数为状态个数
        self.x=array(zeros((self.t,self.m)))
        #先计算出时刻1时，观测为o1,状态为qi的概率
        for i in range(self.m):
            self.x[0][i]=self.pi[i]*self.B[i][self.o[0]]
        for j in range(1,self.t):
            for i in range(self.m):
                #前一时刻所有状态的概率乘以转移概率得到i状态概率
                #i状态的概率*i状态到j观测的概率
                temp=0
                for k in range(self.m):
```

```
                temp=temp+self.x[j-1][k]*self.A[k][i]
            self.x[j][i]=temp*self.B[i][self.o[j]]
        result=0
        for i in range(self.m):
            result=result+self.x[self.t-1][i]
        print u"前向概率矩阵及当前观测序列概率如下: "
         print self.x
        print result

    def backward(self):
        #t时刻状态为qi,从t+1到T观测为ot+1,ot+2,oT的概率用矩阵y表示
        #则矩阵大小行数为观测序列数，列数为状态个数
        self.y=array(zeros((self.t,self.m)))
        #下面为对最终时刻的所有状态，接下来的观测序列概率初始化为1
        #(可以理解为接下来没有观测所有为1)
        for i in range(self.m):
            self.y[self.t-1][i]=1
        j=self.t-2
        #j时刻为i状态，转移到k状态，k状态观测为oj+1,
        #再乘以j+1时刻状态为k的B矩阵的值，对k遍历相加,
        #即为j时刻i状态前提下，后面满足观测序列的概率
        while(j>=0):
            for i in range(self.m):
                for k in range(self.m):
                    self.y[j][i]+=self.A[i][k]*self.B[k]
[self.o[j+1]]*self.y[j+1][k]
            j=j-1
        #第一个状态任意，观测为o1,所以对所有第一个状态概率相加
        result=0
        for i in range(self.m):
            result=result+self.pi[i]*self.B[i][self.o[0]]*self.y[0][i]
        print u'后向概率矩阵及当前观测序列概率如下: '
         print self.y
        print result

    def get_stateprobability(self,t,p):
        #打印在观测为self.o的前提下，t时刻，处于状态p的概率,
        #self.x[t][p]表示到t时刻观测为o1,o2,ot,状态为p的概率
        #self.y[t][p]表示在t时刻状态为p的前提下，接下来观测为ot+1,ot+2,oT的概率
        #self.x[t][p]*self.y[t][p]即表示观测为self.o，且t时刻处于状态p的概率,
        #利用贝叶斯公式，除以观测为self.o的概率即为所求
        if(t>self.t or p>self.m):
            print u'输入数据超过范围'
             return
         print u'在时刻'+str(t)+u'处于状态'+str(p)+u'的概率是: '
         temp=self.x[t-1][p-1]*self.y[t-1][p-1]
        total=0
        for i in range(self.m):
            total=total+self.x[t-1][i]*self.y[t-1][i]
        print temp/total
```

```
    def viterbi(self):
        #利用模型和观测序列找出最优的状态序列
        #时刻t时，很多路径可以到达状态i,且观测为self.o,
        #每个路径都有自己的概率，最大的概率用矩阵z记录,前一个状态用d矩阵记录
        self.z=array(zeros((self.t,self.m)))
        self.d=array(zeros((self.t,self.m)))
        for i in range(self.m):
            self.z[0][i]=self.pi[i]*self.B[i][self.o[0]]
            self.d[0][i]=0
        for j in range(1,self.t):
            for i in range(self.m):
                maxnum=self.z[j-1][0]*self.A[0][i]
                node=1
                for k in range(1,self.m):
                    temp=self.z[j-1][k]*self.A[k][i]
                    if(maxnum<temp):
                        maxnum=temp
                        node=k+1
                self.z[j][i]=maxnum*self.B[i][self.o[j]]
                self.d[j][i]=node
        #找到T时刻概率最大的路径
        max_probability=self.z[self.t-1][0]
        last_node=[1]
        temp=0
        for i in range(1,self.m):
            if(max_probability<self.z[self.t-1][i]):
                max_probability=self.z[self.t-1][i]
                last_node[0]=i+1
                temp=i
        i=self.t-1
        #self.d[t][p],t时刻状态为p的时候，t-1时刻的状态
        while(i>=1):
            last_node.append(self.d[i][temp])
            i=i-1
        temp=['o']
        temp[0]=int(last_node[len(last_node)-1])
        j=len(last_node)-2
        while j>=0:
            temp.append(int(last_node[j]))
            j=j-1
        print u'路径节点分别为'
         print temp
        print u'该路径概率为'+str(max_probability)


if __name__ == '__main__':
    test=HMM()
    test.forward()
    test.backward()
    test.get_stateprobability(3,3)
```

```
test.viterbi()
```

实际运行结果：

```
前向概率矩阵及当前观测序列概率如下：
[[0.1      0.16     0.28    ]
[0.077    0.1104   0.0606  ]
[0.04187  0.035512 0.052836]]
0.130218
后向概率矩阵及当前观测序列概率如下：
[[0.2451 0.2622 0.2277]
[0.54   0.49   0.57  ]
[1.     1.     1.    ]]
0.130218
在时刻3处于状态3的概率是：
0.40575035709348933
路径节点分别为
[3, 3, 3]
该路径概率为0.014699999999999998
```