隐马尔可夫模型

1 隐马尔可夫模型

1.1 模型定义

定义： 隐马尔可夫模型是关于时序的概率模型，描述一个由**隐藏的马尔可夫链**生成**不可观测的状态随机序列**，再有各个状态生成一个**观测随机序列**的过程。

其中，隐藏的马尔可夫链随机生成*状态序列*；每个状态生成一个观测，由此产生的观测随机序列，称为*观测序列*。

隐马尔可夫模型可由**初始概率分布、状态转移概率分布和观测概率分布**确定。可按如下数学形式进行严格定义：

设 $Q=\{q_1,q_2,...,q_N\}$ 是所有状态的集合，$V=\{v_1,v_2,...,v_M\}$ 是所有观测的集合。$I=(i_1,i_2,...,i_T)$ 是转态序列，$O=(o_1,o_2,...,o_T)$ 是观测序列。

*状态的转移矩阵*为 $A=[a_{ij}]_{N\times N}$，其中 $a_{ij}=P(i_{t+1}=q_j|i_t=q_i),i=1,2,...,N,j=1,2,...,N$ 表示在时刻 t 下不可观测的状态序列从 q_i 转移到 q_j 的概率。

*状态观测转移的观测概率矩阵*为 $B=[b_j(k)]_{N\times M},j=1,2,...,N,k=1,2,...,M$，其中 $b_j(k)=P(o_t=v_k|i_t=q_j)$ 表示在时刻 t 下不可观测的状态 q_j 转化为可观测的观测 v_k 的概率。

*状态的初始概率向量*为 $\pi=(\pi_i)$，其中 $\pi_i=P(i_1=q_i),i=1,2,...,N$ 表示 t_1 时刻下处于状态 q_i 的概率。

隐马尔可夫模型由初始状态概率向量 π，状态转移概率矩阵 A 和观测概率矩阵 B 决定。其中状态序列由 π,A 决定，观测序列由 B 决定。因此，隐马尔可夫模型可表示为三元组：$\lambda=(A,B,\pi)$。

隐马尔可夫模型具有以下两个基本假设：

1. 马尔可夫假设：$P(i_t|i_{t-1},o_{t-1},...,i_1,o_1)=P(i_t|i_{t-1})$
2. 观测独立性假设：$P(o_t|o_T,i_T,o_{T-1},i_{T-1},...,o_1,i_1)=P(o_t|i_t)$

隐马尔可夫模型具有以下三个基本问题：

1. **概率计算问题**：给定模型 $\lambda=(A,B,\pi)$ 和观测序列 $O=(o_1,o_2,...,o_T)$，计算在 λ 下观测序列 O 出现的概率 $P(O|\lambda)$

2. **学习问题**：已知观测序列 $O=(o_1,o_2,...,o_T)$，估计模型 $\lambda=(A,B,\pi)$ 的参数
3. **预测问题**：也称为解码问题。已知模型 $\lambda=(A,B,\pi)$ 和观测序列 $O=(o_1,o_2,...,o_T)$，求最有可能的状态序列 $I=(i_1,i_2,...,i_T)$

1.2 概率计算算法

1.2.1 直接计算法

对于给定的状态序列 $I=(i_1,i_2,...,i_T)$，其出现概率为

$$P(I|\lambda)=\pi_{i_1}a_{i_1i_2}a_{i_2i_3}...a_{i_{T-1}i_T}$$

对于固定的状态序列 $I=(i_1,i_2,...,i_T)$ 和观测序列 $O=(o_1,o_2,...,o_T)$ 出现的概率为

$$P(O|I)=b_{i_1}(o_1)b_{i_2}(o_2)...b_{i_T}(o_T)$$

因此，O,I 的联合概率为

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

对所有 I 求和即可得到观测序列 O 出现的概率：

$$\begin{aligned}P(O|\lambda)&=\sum_I P(O,I|\lambda)\\&=\sum_{i_1,i_2,...,i_T}\pi_{i_1}b_{i_1}(o_1)a_{i_1i_2}b_{i_2}(o_2)...a_{i_{T-1}i_T}b_{i_T}(o_T)\end{aligned}$$

此计算方法的计算复杂度为 $O(TN^T)$。因此在实际操作中计算量过大，并不可行，需要使用前向-后向算法减少计算量。

1.2.2 前向算法

定义：给定隐马尔可夫模型 λ，定义到时刻 t 部分观测序列为 $o_1,o_2,...,o_t$ 且状态为 q_i 的概率为**前向概率**，记为：

$$\alpha_t(i)=P(o_1,o_2,...,o_t,i_t=q_i|\lambda)$$

因此，可以获得以下递推式：

$$\begin{aligned}\alpha_{t+1}(i) &= P(o_1, o_2, ..., o_{t+1}, i_{t+1} = q_i|\lambda) \\ &= \sum_{j=1}^{N} P(o_1, o_2, ..., o_{t+1}, i_{t+1} = q_i, i_t = q_j|\lambda) \\ &= \sum_{j=1}^{N} P(o_1, o_2, ..., o_t, i_t = q_j|\lambda) P(i_{t+1} = q_i|i_t = q_j, \lambda) P(o_{t+1}|i_{t+1} = q_i\lambda) \\ &= [\sum_{j=1}^{N} \alpha_t(j) a_{ji}] b_i(o_{t+1})\end{aligned}$$

因此，可按如下步骤计算前向概率 $\alpha_t(i)$：

1. 初始化 $\alpha_1(i) = \pi_i b_i(o_1), i = 1, 2, ..., N$
2. 对 $t = 1, 2, ..., T-1$ 递推：

$$\alpha_{t+1}(i) = [\sum_{j=1}^{N} \alpha_t(j) a_{ji}] b_i(o_{t+1}), i = 1, 2, ..., N$$

3. 计算 $P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$

前向算法的计算复杂度是 $O(TN^2)$，其复杂度大幅度减少的原因是每一次计算直接引用了前一次计算的结果。

1.2.3 后向算法

定义：给定隐马尔可夫模型 λ，定义在时刻 t 状态为 q_i 的条件下，从 $t+1$ 到 T 的部分观测序列 $o_{t+1}, o_{t+2}, ..., o_T$ 的概率为**后向概率**，记为：

$$\beta_t(i) = P(o_{t+1}, o_{t+2}, ..., o_T|i_t = q_i, \lambda)$$

同样，对 $\beta_t(i)$ 也可获得如下递推式：

$$\begin{aligned}\beta_t(i) &= P(o_{t+1}, o_{t+2}, ..., o_T|i_t = q_i, \lambda) \\ &= \sum_{j=1}^{N} P(o_{t+1}, o_{t+2}, ..., o_T, i_{t+1} = q_j|i_t = q_i, \lambda) \\ &= \sum_{j=1}^{N} P(o_{t+2}, ..., o_T, |i_{t+1} = q_j, \lambda) P(i_{t+1} = q_j|i_t = q_i, \lambda) P(o_{t+1}|i_{t+1} = q_j, \lambda) \\ &= \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)\end{aligned}$$

因此，可按如下步骤计算后向概率 $\beta_t(i)$：

1. 初始化 $\beta_T(i)=1, i=1,2,...,N$
2. 对 $t=T-1,T-2,...,1$ 递推：

$$\beta_t(i)=\sum_{j=1}^{N}a_{ij}b_j(o_{t+1})\beta_{t+1}(j), i=1,2,...,N$$

3. 计算 $P(O|\lambda)=\sum_{i=1}^{N}\pi_i b_i(o_1)\beta_1(i)$

后向算法的时间复杂度同样为 $O(TN^2)$。

利用前向算法和后向算法，还可以得到关于单个状态和两个状态的概率计算公式

1. 给定模型 λ 和观测 O，模型在时刻 t 处于状态 q_i 的概率定义为 $\gamma_t(i)=P(i_t=q_i|O,\lambda)$。可以得到：

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

2. 给定模型 λ 和观测 O，模型在时刻 t 处于状态 q_i 且在时刻 $t+1$ 处于状态 q_j 的概率定义为 $\xi_t(i,j)=P(i_t=q_i,i_{t+1}=q_j|O,\lambda)$。可以得到：

$$\begin{aligned}\xi_t(i,j)=P(i_t=q_i,i_{t+1}=q_j|O,\lambda)&=\frac{P(i_t=q_i,i_{t+1}=q_j,O|\lambda)}{P(O|\lambda)}\\&=\frac{P(i_t=q_i,i_{t+1}=q_j,O|\lambda)}{\sum_{i=1}^{N}\sum_{j=1}^{N}P(i_t=q_i,i_{t+1}=q_j,O|\lambda)}\\&=\frac{\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}{\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j)}\end{aligned}$$

1.3 学习算法

1.3.1 监督学习算法

当状态序列和观测序列均是已知时，通过序列 $\{(O_1,I_1),(O_2,I_2),...,(O_S,I_S)\}$ 使用极大似然估计可以简单得到 π,A,B 中的参数。

$$\hat{a}_{ij}=\frac{A_{ij}}{\sum_{i=1}^{N}\sum_{j=1}^{N}A_{ij}},\hat{b}_j(k)=\frac{B_{jk}}{\sum_{k=1}^{M}B_{jk}},\hat{\pi}_i=\frac{\Pi_i}{\sum_{j=1}^{N}\Pi_j}$$

其中，A_{ij} 表示从状态 q_i 转移到状态 q_j 的频数，B_{jk} 表示从状态 b_j 变化为观测 o_k 的频数，Π_i 表示首个状态为 q_i 的频数。

1.3.2 Baum-Welch 算法

当只知道观测序列 $\{O_1,O_2,...,O_T\}$，而不知道状态序列 $\{I_1,I_2,...,I_T\}$ 时，需要使用 EM 算法对参数进行估计。此时，$\{O_1,O_2,...,O_T\}$ 是不完全数据，而 $\{(O_1,I_1),(O_2,I_2),...,(O_S,I_S)\}$ 是完全数据。

算法的 E 步是确定 Q 函数 $Q(\lambda, \bar{\lambda})$:

$$\begin{aligned} Q(\lambda, \bar{\lambda}) &= E_I[\log P(O, I|\lambda)|O, \bar{\lambda}] \\ &= \sum_I \log P(O, I|\lambda) P(I|O, \bar{\lambda}) \\ &= \sum_I \log P(O, I|\lambda) \frac{P(O, I|\bar{\lambda})}{P(O|\bar{\lambda})} \\ &\propto \sum_I \log P(O, I|\lambda) P(O, I|\bar{\lambda}) \end{aligned}$$

而 $P(O, I|\lambda) = \pi_{i_1} b_{i_1}(o_1) a_{i_1 i_2} b_{i_2}(o_2) ... a_{i_{T-1} i_T} b_{i_T}(o_T)$，因此：

$$Q(\lambda, \bar{\lambda}) = \sum_I \log \pi_{i_1} P(O, I|\bar{\lambda}) + \sum_I (\sum_{t=1}^{T-1} \log a_{i_t i_{t+1}}) P(O, I|\bar{\lambda}) + \sum_I (\sum_{t=1}^{T} \log b_{i_t}(o_t)) P(O, I|\bar{\lambda})$$

算法的 M 步需要极大化上述 Q 函数 $Q(\lambda, \bar{\lambda})$。注意到 λ 的参数 A, B, π 分别位于 Q 函数的三项之中，相互不影响，因此只要分别极大化每一项即可。注意参数 A, B, π 满足一定的约束条件：$\sum_{i=1}^{N} \pi_i = 1, \sum_{j=1}^{N} a_{ij} = 1, \sum_{k=1}^{M} b_j(k) = 1$，最大化时需要使用拉格朗日乘数法。得到的结果为：

$$\hat{\pi}_i = \frac{P(O, i_1 = i|\bar{\lambda})}{P(O|\bar{\lambda})} = \gamma_1(i)$$

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} P(O, i_t = i, i_{t+1} = j|\bar{\lambda})}{\sum_{t=1}^{T-1} P(O, i_t = i|\bar{\lambda})} = \frac{\sum_{t=1}^{T-1} \xi_t(i, j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

$$\hat{b}_j(k) = \frac{\sum_{t=1}^{T} P(O, i_t = j|\bar{\lambda}) I(o_t = v_k)}{\sum_{t=1}^{T} P(O, i_t = j|\bar{\lambda})} = \frac{\sum_{t=1}^{T} \gamma_t(j) I(o_t = v_k)}{\sum_{t=1}^{T} \gamma_t(j)}$$

EM 算法作用于隐马尔可夫模型的计算如上所示。其被称为 Baum-Welch 算法，具体步骤如下：

输入：观测数据 $O = (o_1, o_2, ..., o_T)$。

1. 初始化，选取 $a_{ij}^{(0)}, b_j(k)^{(0)}, \pi_i^{(0)}$，得到模型 $\lambda^{(0)} = (A^{(0)}, B^{(0)}, \pi^{(0)})$
2. 使用前向-后向算法计算 $\alpha_t(i), \beta_t(i)$
3. 计算 $\gamma_t(j), \xi_t(i, j)$，并更新参数得到 $\lambda^{(n)} = (A^{(n)}, B^{(n)}, \pi^{(n)})$：

$$\hat{\pi}_i = \gamma_1(i)$$

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

$$\hat{b}_j(k) = \frac{\sum_{t=1}^{T} \gamma_t(j) I(o_t = v_k)}{\sum_{t=1}^{T} \gamma_t(j)}$$

4. 重复 2 和 3 直至收敛

输出：隐马尔可夫模型参数 $\hat{A}, \hat{B}, \hat{\pi}$。

1.4 预测算法

1.4.1 近似算法

近似算法在每个时刻 t 分别选择该时刻最有可能出现的状态 i_t^*，从而得到一个状态序列 $I^* = (i_1^*, i_2^*, ..., i_T^*)$。由于 $\gamma_t(i) = \frac{P(i_t=q_i, O|\lambda)}{P(O|\lambda)}$，因此易于发现：

$$i_t^* = arg \max_{1 \le i \le N} \gamma_t(i)$$

近似算法比较简单，但没有从整体上考虑状态序列。

1.4.2 维特比算法

维特比算法使用动态规划求概率最大的路径。其思想是，要使从 i_1^* 到 i_T^* 的路径最优，那么对于任意的 $2 \le t \le T-1$，从 i_t^* 到 i_T^* 的路径都是最优的。因此维特比算法从 $t=1$ 开始，递推地向后计算各条路径的最大概率，直到 $t=T$。在计算完最大概率后，再从 i_T^* 开始，由后向前求解节点 $i_{T-1}^*, ..., i_1^*$，从而得出最优路径 $I^* = (i_1^*, i_2^*, ..., i_T^*)$。

在向前计算最大概率时，需要递推进行计算以提高效率。定义时刻 t 时状态为 i 的所有单个路径 $(i_1, i_2, ..., i_t)$ 中的概率最大值为：

$$\delta_t(i) = \max_{i_1, i_2, ..., i_{t-1}} P(i_t = i, i_{t-1}, ..., i_1, o_t, o_{t-1}, ..., o_1|\lambda), \quad i = 1, 2, ..., N$$

因此有

$$\begin{aligned} \delta_{t+1}(i) &= \max_{i_1, i_2, ..., i_t} P(i_{t+1} = i, i_t, ..., i_1, o_{t+1}, o_t, ..., o_1|\lambda) \\ &= \max_{1 \le j \le N} [\delta_t(j) a_{ji}] b_i(o_{t+1}), \quad i = 1, 2, .., N; t = 1, 2, ..., T-1 \end{aligned}$$

在反向寻找最优节点时，也可定义在时刻 t 时状态为 i 的所有单个路径 $(i_1, i_2, ..., i_{t-1}, i)$ 中概率最大的路径的第 $t-1$ 个节点为：

$$\Psi_t(i) = arg \max_{1 \le j \le N} [\delta_{t-1}(j) a_{ji}], \quad i = 1, 2, ..., N$$

使用函数 δ 和 Ψ，可以得到维特比算法，其步骤如下：

输入：模型 $\lambda = (A, B, \pi)$ 和观测 $O = \{o_1, o_2, ..., o_T\}$

1. 初始化

$$\delta_1(i) = \pi_i b_i(o_1), \quad i = 1, 2, ..., N$$

$$\Psi_1(i) = 0, \quad i = 1, 2, ..., N$$

2. 向前递推，对 $t = 1, 2, ..., T-1$，计算

$$\delta_{t+1}(i) = \max_{1 \le j \le N}[\delta_t(j)a_{ji}]b_i(o_{t+1}), \quad i = 1, 2, ..., N$$

$$\Psi_t(i) = arg \max_{1 \le j \le N}[\delta_{t-1}(j)a_{ji}], \quad i = 1, 2, ..., N$$

3. 找出最大概率 $P^* = \max_{1 \le i \le N} \delta_T(i)$ 并确定最终节点 $i_T^* = arg \max_{1 \le i \le N} \delta_T(i)$
4. 反向找出最优节点，对 $t = T-1, T-2, ..., 1$，计算

$$i_t^* = \Psi_{t+1}(i_{t+1}^*)$$

输出：最优路径 $I^* = (i_1^*, i_2^*, ..., i_T^*)$。

2 代码实现

本次考虑的隐马尔可夫模型参数如下：

$$\pi = (0.2, 0.4, 0.4)^T, A = \begin{bmatrix} 0.5 & 0.2 & 0.3 \\ 0.3 & 0.5 & 0.2 \\ 0.2 & 0.2 & 0.6 \end{bmatrix}, B = \begin{bmatrix} 0.5 & 0.5 \\ 0.4 & 0.6 \\ 0.7 & 0.3 \end{bmatrix}$$

从该隐马尔可夫模型中随机生成 50 个长度为 50 的观测序列作为此次的数据。

2.1 sklearn 实现

sklearn 中的隐马尔可夫模型的模块已被独立到 hmmlearn 包中。使用 MultinomialHMM 函数可以模拟或者拟合离散的隐马尔可夫模型，该函数的主要属性有：

- startprob_：初始状态概率向量
- transmat_：状态转移矩阵
- emissionprob_：观测概率矩阵

首先，根据隐马尔可夫模型的参数模拟出 50 个长度为 50 的观测序列：

```
import numpy as np
from hmmlearn.hmm import MultinomialHMM

np.random.seed(12345)
model = MultinomialHMM(n_components=3)
model.startprob_ = np.array([0.2, 0.4, 0.4])
model.transmat_ = np.array([[0.5, 0.2, 0.3],
                            [0.3, 0.5, 0.2],
                            [0.2, 0.3, 0.5]])
model.emissionprob_ = np.array([[0.5, 0.5],
                                [0.4, 0.6],
                                [0.7, 0.3]])
O = np.concatenate([model.sample(50)[0] for _ in range(50)],axis=0)
```

再创建一个新的类，对其进行拟合并输出结果：

```
remodel = MultinomialHMM(n_components=3)
remodel.fit(O,lengths=[50 for _ in range(50)])
```

```
print('初始状态概率向量:\n',remodel.startprob_,
      '\n状态转移矩阵:\n',remodel.transmat_,
      '\n观测概率矩阵:\n',remodel.emissionprob_)
```

```
## 初始状态概率向量:
##  [0.29689949 0.43891086 0.26418965]
## 状态转移矩阵:
##  [[0.34383868 0.29584149 0.36031983]
##  [0.34050106 0.3076312  0.35186774]
##  [0.34480482 0.29234502 0.36285016]]
## 观测概率矩阵:
##  [[0.60855692 0.39144308]
##  [0.22427814 0.77572186]
##  [0.71926203 0.28073797]]
```

2.2 预测算法

在类中实现以下几个函数：

- compute_alpha,compute_beta：分别为前向计算和后向计算
- fit：使用 Baum-Welch 算法进行学习，不过在样本量较小时学习效果一般

- approx,viterbi：分别使用近似算法和维特比算法进行状态序列的预测

```
class HiddenMarkovModel:

    def __init__(self,n_components=1,max_iter=100,\
                 pi=None,A=None,B=None):
        self.n_components = n_components
        self.max_iter = max_iter
        self.pi = pi
        self.A = A
        self.B = B

    def compute_alpha(self,X):
        T = len(X)
        alpha = np.zeros((T,self.n_components))
        for i in range(T):
            if i==0:
                alpha[i,:] = self.pi*self.B[:,X[i]].reshape(-1)
            else:
                alpha[i,:] = np.sum(self.A.T*alpha[i-1,:],1)*self.B[:,X[i]].reshape(-1)
        return alpha

    def compute_beta(self,X):
        T = len(X)
        beta = np.ones((T,self.n_components))
        for i in range(T-1,0,-1):
            beta[i-1,:] = np.sum(self.A*self.B[:,X[i]].reshape(1,-1)*beta[i,:],0)
        return beta

    def compute_gamma(self):
        return self.alpha*self.beta/np.sum(self.alpha*self.beta,1).reshape(-1,1)

    def compute_xi(self,X):
        T,N = self.alpha.shape
        xi = np.zeros((T-1,N,N))
        for t in range(0,T-1):
            for i in range(0,N):
                for j in range(0,N):
                    xi[t,i,j] = self.alpha[t,i]*self.A[i,j]*self.B[j,X[t+1]]*self.beta[t+1,j]
```

```
        return xi/np.sum(np.sum(xi,2),1).reshape(-1,1,1)

    def fit(self,X):
        N,K,T = self.n_components,len(np.unique(X)),len(X)

        self.pi = np.array([np.random.random() for _ in range(N)])
        self.pi = self.pi/np.sum(self.pi)

        self.A = np.array([np.random.random() for _ in range(N*N)]).reshape(N,N)
        self.A = self.A/np.sum(self.A,1).reshape(-1,1)

        self.B = np.array([np.random.random() for _ in range(N*K)]).reshape(N,K)
        self.B = self.B/np.sum(self.B,1).reshape(-1,1)



        self.alpha = np.zeros((T,N))
        self.beta = np.ones((T,N))
        self.xi = np.zeros((T-1,N,N))

        for _ in range(self.max_iter):
            self.alpha = self.compute_alpha(X)
            self.beta = self.compute_beta(X)
            self.gamma = self.compute_gamma()
            self.xi = self.compute_xi(X)

            self.A = np.sum(self.xi,axis=0).reshape(N,N)/np.sum(self.gamma[:-1,:],axis=0)
            self.B = np.concatenate([(np.sum((X==i)*self.gamma,0)/np.sum(self.gamma,0)).\
                    reshape(N,1) for i in range(K)],1)
            self.pi = self.gamma[0,:].reshape(-1)

    def approx(self,O):
        alpha = self.compute_alpha(O)
        beta = self.compute_beta(O)
        gamma = alpha*beta/np.sum(alpha*beta,1).reshape(-1,1)
        return np.argmax(gamma,1)

    def viterbi(self,O):
        T,N = len(O),self.n_components
```

```
        delta = np.zeros((T,N))
        psi = np.zeros((T,N))
        for t in range(T):
            if t==0:
                delta[t,:] = self.pi*self.B[:,O[t]].reshape(-1)
            else:
                delta[t,:] = np.max(self.A*delta[t-1,:],1)*self.B[:,O[t]].reshape(-1)
                psi[t,:] = np.argmax(self.A*delta[t-1,:],1)
        path = []
        max_prob = np.max(delta[-1,:])
        path.append(np.argmax(delta,1)[-1])
        for t in range(T-1,0,-1):
            path.append(int(psi[t,int(path[-1])]))
        return np.array(path[::-1])
```

模拟一个长度为 10 的序列，预测算法结果如下：

```
X, Z = model.sample(10)

clf = HiddenMarkovModel(n_components=3)
clf.pi = np.array([0.2, 0.4, 0.4])
clf.A = np.array([[0.5, 0.2, 0.3],
                  [0.3, 0.5, 0.2],
                  [0.2, 0.3, 0.5]])
clf.B = np.array([[0.5, 0.5],
                  [0.4, 0.6],
                  [0.7, 0.3]])

print('状态序列:',Z,'\n近似算法估计:',clf.approx(X),'\n维特比算法估计:',clf.viterbi(X))
```

```
## 状态序列: [1 1 2 1 0 1 2 1 1 1]
## 近似算法估计: [1 1 1 1 1 1 1 1 1 1]
## 维特比算法估计: [1 1 1 1 1 1 1 1 1 1]
```