隐马尔可夫模型(HMM)

作者：刘伟杰 日期：2015-12-12
参考：[1]《统计学习方法》 李航 2012年3月第一版

1. 理论

1. 概述：

   隐马尔可夫模型是一个关于时间序列的概率模型，模型由初始状态随机生成不可观测的状态序列（隐藏的马尔可夫链），再由状态序列中的状态随机生成可观测的观测序列。

2. 模型：

   在定义马尔可夫模型前，先定义这个模型相关的一些量。

   所有可能的N个状态的集合Q:

   $$Q = \{q_0, q_1, ...q_{N-1}\}$$

   所有可能的M个观测的集合V:

   $$V = \{v_0, v_1, ...v_{M-1}\}$$

   长度为T的状态序列I，其中i_{?}为序号，对应状态集合Q中的状态:

   $$I = (i_0, i_1, ..., i_{T-1})$$

   对应的观测序列O, 其中0_{?}为序号，对应观测集合V中的观测值：

   $$O = (o_0, o_1, ..., o_{T-1})$$

   隐藏马尔可夫模型可以由初始状态向量pi, 状态转移矩阵A和观测概率矩阵B三个量表示，即一个马尔可夫模型可以表示成hmm=(A,B,pi)

   状态转移矩阵A, 其中a_{ij}表示由状态q_{i}转到状态q_{j}的概率:

   $$A = [a_{ij}]_{N*N}$$

   观测概率矩阵B, 其中b_{ij}表示由状态q_{i}观测到观测值v_{j}的概率:

   $$B = [b_{ij}]_{N*M}$$

   初始状态向量pi，表示t = 0时刻，处于各个状态的概率。pi_{i}表示处于状态q_{i}的概率：

$$\pi = (\pi_0, \pi_1, ..., \pi_{N-1})$$

马尔可夫模型三元表示法：

$$hmm = (A, B, \pi)$$

2. 一些有用的概念

把问题抽象成马尔可夫模型后，就可以用该模型来研究问题。这里有一些比较常用且基本的概念，有助于研究马尔可夫模型。

1. 前向概率alpha:

   定义：前向概率alpha_{t}(i): 给定马尔可夫模型hmm，定义到t时刻，部分观测序列为o_{0},o_{1},…,o_{t}且状态为q_{i}的概率为前向概率，记作

   $$\alpha_t(i) = P(o_0, o_1, ..., o_t, i_t = i | hmm)$$

   求法:

   (1) 设初值

   $$\alpha_0(i) = \pi_i b_{io_0}, \quad i=0,1,2,\cdots,N-1$$

   (2) 递推 对 $t=0,1,2,\cdots,T-2$

   $$\alpha_{t+1}(i) = \left[\sum_{j=1}^{N-1} \alpha_t(j) a_{ij}\right] b_{io_{t+1}}, \quad i=0,1,2,\cdots N-1$$

2. 后向概率beta:

   定义：后向概率beta_{t}(i): 给定马尔可夫模型humm，定义在时刻t状态为q_{i}的条件下，从t＋1到T-1的部分观测序列为o_{t+1},o_{t+2},…,o_{T-1}的概率为后向概率，记作

   $$\beta_t(i) = P(o_t, o_{t+1}, ..., o_{T-1} | i_t = i, hmm)$$

   求法:

   (1) 设初值

   $$\beta_{T-1}(i) = 1, \quad i=0,1,2,\cdots N-1$$

   (2) 递推：对 $t=T-2, T-3, \cdots, 1$

   $$\beta_t(i) = \sum_{j=1}^{N-1} a_{ij} b_{jo_{t+1}} \beta_{t+1}(j) \quad i=0,1,2,\cdots N-1$$

3. ganmma:

   定义：给定模型hmm和观测序列O，在时刻t处于状态q_{i}的概率，记

$$\gamma_t(i) = P(i_t = i|O, hmm)$$

求法：

$$\gamma_t(i) = \frac{\alpha_t(i)\alpha_t(i)}{\sum_{j=0}^{N-1}\alpha_t(j)\alpha_t(j)}$$

4. xi

   定义: 给定模型hmm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

$$\xi_t(i,j) = P(i_t = i, i_{t+1} = j|O, hmm)$$

   求法：

$$\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_{jo_{t+1}}\beta_{t+1}(j)}{\sum_{i=0}^{N-1}\sum_{i=0}^{N-1}\alpha_t(i)a_{ij}b_{jo_{t+1}}\beta_{t+1}(j)}$$

3. 常见的一些问题：

1. 学习问题：

   监督学习：

   已知训练包含S个长度相同的观测序列和对应的状态序列{(O1,I1),(O1,I1),…,(OS,IS)}，那么用极大似然估计法来估计马尔库夫模型的参数，具体方法见[1] p180。这里补充一下，关于初始状态pi的估计，可以根据具体情况设计，例如如果已知初始状态或者初始观测值。

   非监督学习法：

   假定训练数据只包含S个长度为T的观测序列{O1,O2,…OS},而没有对应的状态序列，目标是隐马尔可夫模型的参数。使用Baum-Welch算法, 见[1]p181。

2. 概率问题：

   已知隐马尔可夫模型hmm

   输入:观测序列O； 输出: 该观测序列出现的概率P(O|humm)

   有前向算法和后向算法两种，见[1]p175。

3. 预测问题：

   已知隐马尔可夫模型hmm

   输入:观测序列O； 输出：该序列最有可能对应的状态序列I, 及这个I的概率

维特比算法见[1]p185

4. 实现:

我的实现：(Baum-Welch算法还存在问题，我还需要再仔细研究研究。)

```
# -*-coding:UTF-8-*-
'''
    Created on 2015-12-10
    @author: Liu Weijie
    reference:
        [1] <Statistical Learning> Li Han p171~189
        [2] http://www.tuicool.com/articles/3iENzaV
'''

import numpy as np


class HMM:

    def __init__(self, Ann=[[0]], Bnm=[[0]], pi1n=[0]):
        self.A = np.array(Ann)
        self.B = np.array(Bnm)
        self.pi = np.array(pi1n)
        self.N = self.A.shape[0]
        self.M = self.B.shape[1]

    def printhmm(self):
        print "=================================================="
        print "HMM content: N =", self.N, ",M =", self.M
        for i in range(self.N):
            if i == 0:
                print "hmm.A ", self.A[i, :], " hmm.B ", self.B[i, :]
            else:
                print "      ", self.A[i, :], "       ", self.B[i, :]
        print "hmm.pi", self.pi
        print "=================================================="

    def compute_alpha(self, O):
        """
        function: calculate forward prob alpha(t,i), definition can be seen in [1] p17
        :param O: obersive list in 1-D aeeay like O = np.array([0, 1, 0])
        :return alpha: forward prob in 2-D array, alpha[t,i] means alpha_{t}(i)
        """
        # step1: initialization
        T = len(O)
        alpha = np.zeros((T, self.N), np.float)
        for i in range(self.N):
            alpha[0, i] = self.pi[i] * self.B[i, O[0]]

        # step2: induction
        for t in range(T - 1):
            for j in range(self.N):
                sum_alpha = 0.0
                for i in range(self.N):
                    sum_alpha += alpha[t, i] * self.A[i, j]
                alpha[t + 1, j] = sum_alpha * self.B[j, O[t + 1]]
```

```
        return alpha

    def compute_beta(self, O):
        """
        function: calculate forward prob alpha(t,i), definition can be seen in [1] p17
        :param O: obersive list in 1-D aeeay like O = np.array([0, 1, 0])
        :return beta: beta in 2_D array, beta[t, i] means gamma_{t}(i)
        """
        # step1: initalization
        T = len(O)
        beta = np.zeros((T, self.N), np.float)
        for i in range(self.N):
            beta[T - 1, i] = 1.0

        # step2: induction
        for t in range(T - 2,-1,-1):
            for i in range(self.N):
                sum_beta = 0.0
                for j in range(self.N):
                    sum_beta += self.A[i, j] * self.B[j, O[t + 1]] * beta[t + 1, j]
                beta[t, i] = sum_beta

        return beta

    def compute_gamma(self, O):
        """
        function: calculate gamma, definition can be seen in [1] p179 (10.23)
        :param O: obersive list in 1-D aeeay like O = np.array([0, 1, 0])
        :return gamma: gamma in 2-D array, gamma[t,i] means gamma_{t}(i)
        """
        T = len(O)
        gamma = np.zeros((T, self.N), np.float)
        alpha = self.compute_alpha(O)
        beta = self.compute_beta(O)
        for t in range(T):
            for i in range(self.N):
                sum_N = 0.0
                for j in range(self.N):
                    sum_N += alpha[t, j] * beta[t, j]
                gamma[t, i] = (alpha[t, i] * beta[t, i]) / sum_N
        return gamma

    def compute_xi(self, O):
        """
        function: calculate xi, definition can be seen in [1] p179 (10.25)
        :param O: obersive list in 1-D aeeay like O = np.array([0, 1, 0])
        :return xi: xi in 3-D array, xi[t,i,j] means xi_{t}(i,j)
        """
        T = len(O)
        xi = np.zeros((T - 1, self.N, self.N))
        alpha = self.compute_alpha(O)
```

```
        beta = self.compute_beta(O)

        for t in range(T - 1):
            sum_NN = 0.0
            for i in range(self.N):
                for j in range(self.N):
                    sum_NN = alpha[t, i] * self.A[i, j] * self.B[j, O[t + 1]] * beta[t
            for i in range(self.N):
                for j in range(self.N):
                    xi[t, i, j] = (alpha[t, i] * self.A[i, j] * self.B[j, O[t + 1]] *
        return xi

    def forward(self, O):
        """
        function: forward algorithm to calculate P(O|lamda) with the input obersive li
        :param O: obersive list in 1-D aeeay like O = np.array([0, 1, 0])
        :return pprob: P(O|lamda)
        """
        alpha = self.compute_alpha(O)
        T = len(O)
        sum_N = 0.0
        for i in range(self.N):
            sum_N += alpha[T - 1, i]
        pprob = sum_N
        return pprob

    def backward(self, O):
        """
        function: backward algorithm to calculate P(O|lamda) with the input obersive l
        :param O: obersive list in 1-D aeeay like O = np.array([0, 1, 0])
        :return pprob: P(O|lamda)
        """
        beta = self.compute_beta(O)
        sum_N = 0.0
        for i in range(self.N):
            sum_N += self.pi[i] * self.B[i, O[0]] * beta[0, i]
        pprob = sum_N
        return pprob

    def viterbi(self, O):
        """
        function: used for predict problem, and can be seen in [1] p184
        :param O: obersive list in 1-D aeeay like O = np.array([0, 1, 0])
        :return I: state list of the most possibility
                prob: possibility
        """
        T = len(O)
        # initial
        delta = np.zeros((T, self.N), np.float)
        phi = np.zeros((T, self.N), np.float)
        I = np.zeros(T)
        for i in range(self.N):
```

```
            delta[0, i] = self.pi[i] * self.B[i, O[0]]
            phi[0, i] = 0
        # induction
        for t in range(1, T):
            for i in range(self.N):
                delta[t,i] = self.B[i, O[t]] * np.array([delta[t - 1, j] * self.A[j, i
                phi[t, i] = np.array([delta[t - 1,j] * self.A[j, i] for j in range(sel
        # terminal
        prob = delta[T - 1, :].max()
        I[T - 1] = delta[T - 1, :].argmax()
        # get I
        for t in range(T - 2, -1, -1):
            I[t] = phi[t + 1, I[t + 1]]
        return I, prob

    def baum_welch(self, O, num_observed_value=-1, num_state=-1, num_itera=1000):
        """
        function: baum-welch method (so called EM algorithm) is to train patameters (A
                  by a set of obersive list O, which is a unsupervied learning. refere
        :param num_observed_value:
        :param num_state:
        :param O_set: a set of obersive list O, 2-D array like O_set = np.array([[0,1,
        :output: (A, B, pi) in HMM model
        """
        print " baum_welch function has some problem, please don't use now!!!!!"
        raise ValueError
        # step1: initial
        self.A = np.ones((num_state, num_state), np.float)
        self.B = np.ones((num_state, num_observed_value), np.float)
        self.N = self.A.shape[0]
        self.M = self.B.shape[1]
        self.pi = np.ones((num_state,), np.float)
        self.A = self.A / self.N
        self.B = self.B / self.M
        self.pi = self.pi / self.N
        T = len(O)

        # step2: induction
        for n in range(num_itera):
            xi = self.compute_xi(O)
            gamma = self.compute_gamma(O)
            sum_xi = np.sum(xi, axis=0)
            sum_gamma = np.sum(gamma, axis=0)

            # calculate A[i, j] (n+1)
            for i in range(self.N):
                for j in range(self.N):
                    self.A[i, j] = sum_xi[i, j] / (sum_gamma[i] - gamma[-1, j])

            # calculate B[i,j] (n+1)
            for i in range(self.N):
                for j in range(self.M):
```

```
                sum_gamma_j = 0.0
                for t in range(T):
                    if O[t] == j:
                        sum_gamma_j += gamma[t, j]
                self.B[i, j] = sum_gamma_j / sum_gamma[i]

        # calculate pi
        self.pi = gamma[1, :]

    def train(self, I, O, num_state, num_obserivation, init_observation=-1, init_state
        """
        function: training HMM
        :param I: state list like I = np.array([[0,1,2],[1,0,1],[1,2,0],])
        :param O: observation list like O =     O = np.array([[0,1,1],[1,0,1],[1,1,0],
        :param num_state: the number of state, lke 3
        :param num_obserivation: the number of observation, like 2
        :param init_observation: the index of init observation, like 1
        :param init_state: the index of init starw, like 2
        """
        print "statr training HMM..."
        self.N = num_state
        self.M = num_obserivation

        # count num_A[i,j] standing for the numbers of state i translating to state j
        num_A = np.zeros((num_state, num_state), np.float)
        for i in range(self.N):
            for j in range(self.N):
                num_i2j = 0
                for i_I in range(I.shape[0]):
                    for j_I in range(I.shape[1] - 1):
                        if I[i_I, j_I] == i and I[i_I, j_I + 1] == j:
                            num_i2j += 1
                num_A[i, j] = num_i2j

        # count num_B[i,j] standing for the numbers of state i translating to obsrtvat
        num_B = np.zeros((num_state, num_obserivation), np.float)
        for i in range(self.N):
            for j in range(self.M):
                num_i2j = 0
                for i_I in range(I.shape[0]):
                    for j_I in range(I.shape[1]):
                        if I[i_I, j_I] == i and O[i_I, j_I] == j:
                            num_i2j += 1
                num_B[i, j] = num_i2j

        self.A = num_A / np.sum(np.mat(num_A), axis=1).A
        self.B = num_B / np.sum(np.mat(num_B), axis=1).A

        # calculate pi according init_observation or init_state
        if init_state != -1:
            print "init pi with init_state!"
            pi_temp = np.zeros((self.N,), np.float)
```

```
            self.pi = pi_temp[init_state] = 1.0
        elif init_observation != -1:
            print "init pi with init_observation!"
            self.pi = self.B[:, init_observation] / np.sum(self.B[:, init_observation]
        else:
            print "init pi with state list I!"
            self.pi = np.zeros((self.N,), np.float)
            for i in range(self.N):
                num_state_i = 0
                for line in I:
                    if line[0] == i:
                        num_state_i += 1
                self.pi[i] = num_state_i
            self.pi = self.pi/np.sum(self.pi, axis=0)

        print "finished train successfully! the hmm is:"
        self.printhmm()


if __name__ == "__main__":
    # 已知hmm模型，用来预测
    print "python my HMM"
    A = np.array([
        [0.5, 0.2, 0.3],
        [0.3, 0.5, 0.2],
        [0.2, 0.3, 0.5],
    ])
    B = np.array([
        [0.5, 0.5],
        [0.4, 0.6],
        [0.7, 0.3],
    ])
    pi = np.array([0.2, 0.4, 0.4])
    hmm = HMM(A, B, pi)
    O1 = np.array([0, 0, 0])
    hmm.printhmm()
    print hmm.viterbi(O1)
    print hmm.forward(O1)

    # 已知观测序列与对应的状态序列，训练得到hmm模型
    I = np.array([
        [0,1,2],
        [1,0,1],
        [1,2,0],
    ])
    O = np.array([
        [0,1,1],
        [1,0,1],
        [1,1,0],
    ])
    hmm2 = HMM()
    hmm2.train(I, O, 3, 2)  # 未知初始状态或观测值
```

```
hmm2.train(I, O, 3, 2, init_observation=0)  # 已知初开始观测值
hmm2.train(I, O, 3, 2, init_state=0)  # 已知初始状态
```