HMM——三个基本问题的计算

概率计算问题

继续上面的例子。现在模型已经给定，观测序列也已经知道了，我们要计算的是 O = (红宝石,珍珠,珊瑚) 的出现概率，我们要求的是 $P(O|\lambda)$ 。

直接计算

用直接计算法来求 λ 情况下长度为 T 的观测序列 O 的概率：

$P(O|\lambda) = \sum_{S \in S_T} P(O, S|\lambda)$

其中 S_T 表示所有长度为 T 的状态序列的集合，S 为其中一个状态序列。

对所有长度为 T 的状态序列 S_t 和观测序列 O 求以 λ 为条件的联合概率，然后对所有可能的状态序列求和，就得到了 $P(O|\lambda)$ 的值。

因为 $P(O, S|\lambda) = P(O|S, \lambda)P(S|\lambda)$;

又因为 $P(O|S, \lambda) = b_{11}b_{22}\dots b_{TT}$;

而 $P(S|\lambda) = \pi_1 a_{12}a_{23}\dots a_{(T-1)T}$，其中 a_{ij} 为矩阵 A 中的元素;

所以 $P(O|\lambda) = \sum_{s_1, s_2, \dots, s_T} \pi_1 b_{11} a_{12} b_{22} a_{23} \dots a_{(T-1)T} b_{TT}$。

理论上讲，我们可以把所有状态序列都按照上述公式计算一遍，但它的时间复杂度是 $O(TN^T)$，计算量太大了，基本上不可行。

前向-后向算法

如果不是直接计算，还有什么办法可以算出一个观测序列出现的概率吗?

当然有，那就是**前向-后向算法**。

前向—后向算法是一种**动态规划算法**。它分两条路径来计算观测序列概率，一条从前向后（前向），另一条从后向前（后向）。这两条路径，都可以分别计算观测序列出现的概率。

在实际应用中，选择其中之一来计算就可以。

所以，前向-后向算法其实也可以被看作两个算法：前向算法和后向算法，它们可以分别用来求解 $P(O|\lambda)$。

前向算法

设$\alpha_t(i) = P(o_1, o_2, \dots, o_t, s_t = S_i|\lambda)$为**前向概率**。

它表示的是：给定 λ 的情况下，到时刻 t 时，已经出现的观测序列为 $o_1, o_2, \dots o_t$且此时状态值为 S_i 的概率。

此处写法有点 Confusing，这里具体解释一下。

现在是 t 时刻，到此刻为止，我们获得的观测序列是 $(o_1, o_2, \dots, o_t)$，这些 $o_1, o_2, \dots, o_t$的下标标志是时刻（而非观测空间的元素下标），具体某个 o_i的取值才是观测空间中的一项。因此观测序列中很可能出现 $o_i = o_j$的状况。

S_i 为一个具体的状态值。当前 HMM 的状态空间一共有 N 个状态值：$(S_1, S_2, \dots, S_N)$，S_i是其中的一项。

s_t 在此处表示 t 时刻的状态，它的具体取值是 S_i。

因此有：

(1)$\alpha_1(i)=\pi_i b_i(o_1),\ \ i=1,2,\ldots,N;$

(2)递推，对于 $t=1,\ 2,\ \ldots,T-1$，有 $\alpha_{t+1}(i)=[\sum_{j=1}^{N}\alpha_t(j)a_{ji}]b_i(o_{t+1}),\ \ i=1,2,\ldots,N;$

(3)最终 $P(O|\lambda)=\sum_{i=1}^{N}\alpha_T(i),\ \ i=1,2,\ldots,N$。

如此一来，概率计算问题的计算复杂度就变成了$O(N^2T)$。

后向算法

设 $\beta_t(i)=P(o_{t+1},o_{t+2},\ldots,o_T|s_t=S_i,\lambda)$ 为**后向概率。**

它表示的是：给定 λ 的情况下，到时刻 t 时，状态为 S_i 的条件下，从 t+1 到 T 时刻出现的观察序列为 o_{t+1}，$o_{t+2},\ldots,o_T$的概率。

(1)$\beta_T(i)=1,\ \ i=1,2,\ldots,N$——到了最终时刻，无论状态是什么，我们都规定当时的后向概率为1；

(2)对 $t=T-1,T-2,\ldots,1$，有 $\beta_t(i)=\sum_{j=1}^{N}a_{ij}b_j(o_{t+1})\beta_{t+1}(j),\ \ i=1,2,\ldots,N;$

(3)$P(O|\lambda)=\sum_{i=1}^{N}\pi_i b_i(o_1)\beta_1(i),\ \ i=1,2,\ldots,N$。

结合前向后向算法，定义 $P(O|\lambda)$：

$P(O|\lambda)=\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_t(i)a_{ij}b_j(o_{t+1})\beta_{t+1}(j),\ \ t=1,2,\ldots,T-1$

预测算法

直接求解

预测算法是在λ 既定，观测序列已知的情况下，找出最有可能产生如此观测序列的状态序列的算法。

比如，上面的例子中，真的出现了 O= (红宝石,珍珠,珊瑚)，那么，最有可能导致 O 出现的 S 是什么?

比较**直接的算法**是：在每个时刻 t，选择在该时刻最有可能出现的状态 s_t^*，从而得到一个状态序列 $S^*=(s_1^*,s_2^*,\ldots,s_T^*)$，直接就把它作为预测结果。

给定 λ 和观测序列 O 的情况下，t 时刻处于状态 S_i 的概率为：

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

因为 $\gamma_t(i)=P(s_t=S_i|O,\lambda)=\frac{P(s_t=S_i,O|\lambda)}{P(O|\lambda)}$，通过前向概率 $\alpha_t(i)$ 和后向概率 $\beta_t(i)$定义可知：

$\alpha_t(i)\beta_t(i)=P(s_t=S_i,O|\lambda)$。

所以有：$\gamma_t(i)=\frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)}=\frac{\alpha_t(i)\beta_t(i)}{\sum_{j=1}^{N}\alpha_t(j)\beta_t(j)}$。

有了这个公式，我们完全可以求出 t 时刻，状态空间中每一个状态值$S_1,S_2,\ldots,S_N$ 所对应的$\gamma_t(i)$，然后选取其中概率最大的作为 t 时刻的状态即可。

这种预测方法在每一个点上都求最优，然后再“拼”成整个序列。

它的优点是简单明了，缺点同样非常明显：不能保证状态序列对于观测序列的支持整体最优。

维特比算法　全局最优解：动规+回溯

如何避免掉那些实际上不会发生的状态序列，从而求得整体上的“最有可能”呢？

有一个很有名的算法：**维特比算法——用动态规划求解概率最大路径。**

维特比算法是求解 HMM 预测问题的经典算法。不过这个算法我们就不在本课中讲解了，大家可以自己找资料学习。

> 注意：本课中有不少地方会用到动态规划算法（比如前面的 SMO）。
>
> 因为动态规划是一类相对独立且不算非常“基础”的算法，对它我们不在本课中展开介绍。即使 SMO，也是一带而过。

学习算法

HMM 的学习算法根据训练数据的不同，可以分为**有监督学习**和**无监督学习**两种。

如果训练数据既包括观测序列，又包括对应的状态序列，且两者之间的对应关系已经明确标注了出来，那么就可以用有监督学习算法。

否则，如果只有观测序列而没有明确对应的状态序列，就需要用无监督学习算法训练。

有监督学习

训练数据是若干观测序列和对应的状态序列的样本对（Pair）。也就是说训练数据不仅包含观测序列，同时每个观测值对应的状态值是什么，也都是已知的。

那么，最简单的，我们可以用频数来估计概率。

我们经过统计训练数据中所有的状态值和观测值，可以得到状态空间 $(S_1, S_2, \ldots, S_N)$ 和观测空间 $(O_1, O_2, \ldots, O_M)$。

假设样本中时刻 t 处于状态S_i，到了 t+1 时刻，状态属于 S_j 的频数为 A_{ij}，那么状态转移概率 a_{ij}的估计是：

$$\hat{a_{ij}} = \frac{A_{ij}}{\sum_{j=1}^{N}(A_{ij})}, \quad i = 1, 2, \ldots, N; \quad j = 1, 2, \ldots, N$$

假设样本状态为 S_j，观测为 O_k 的频数为 B_{jk}，观测概率 b_{jk} 的估计是：

$$\hat{b_{jk}} = \frac{B_{jk}}{\sum_{k=1}^{M}(B_{jk})}, \quad j = 1, 2, \ldots, N; \quad k = 1, 2, \ldots, M$$

初始状态概率 π_i 的估计 $\hat{\pi_i}$为所有样本中初始状态为 S_i 的频率。

这样，通过简单的统计估计，就得到了$\lambda = (\hat{A}, \hat{B}, \hat{\pi})$。

无监督学习

当训练数据仅有观测序列（设观测序列为 O），而没有与其对应的状态序列时，状态序列 S 实则处于隐藏状态。

这种情况下，我们是不可能直接用频数来估计概率的。

有专门的 `Baum-Welch` 算法，针对这种情况求$\lambda = (A, B, \pi)$。

`Baum-Welch` 算法利用了前向-后向算法，同时还是 EM 算法的一个特例。简单点形容，大致是一个嵌套了 EM 算法的前向-后向算法。

因为 EM 算法我们在无监督学习部分会专门介绍，此处不再展开解释 `Baum-Welch` 算法。感兴趣的朋友可以自学。

HMM 实例

用代码实现前述小马轮盘赌问题：

```
from __future__ import division
import numpy as np

from hmmlearn import hmm

def calculateLikelyHood(model, X):
    score = model.score(np.atleast_2d(X).T)

    print "\n\n[CalculateLikelyHood]: "
    print "\nobservations:"
    for observation in list(map(lambda x: observations[x], X)):
        print " ", observation

    print "\nlikelyhood:", np.exp(score)

def optimizeStates(model, X):
    Y = model.decode(np.atleast_2d(X).T)
    print"\n\n[OptimizeStates]:"
    print "\nobservations:"
    for observation in list(map(lambda x: observations[x], X)):
        print " ", observation

    print "\nstates:"
    for state in list(map(lambda x: states[x], Y[1])):
        print " ", state


states = ["Gold", "Silver", "Bronze"]
n_states = len(states)

observations = ["Ruby", "Pearl", "Coral", "Sapphire"]
n_observations = len(observations)

start_probability = np.array([0.3, 0.3, 0.4])

transition_probability = np.array([
    [0.1, 0.5, 0.4],
    [0.4, 0.2, 0.4],
    [0.5, 0.3, 0.2]
])

emission_probability = np.array([
    [0.4, 0.2, 0.2, 0.2],
    [0.25, 0.25, 0.25, 0.25],
    [0.33, 0.33, 0.33, 0]
])


model = hmm.MultinomialHMM(n_components=3)

# 直接指定pi: startProbability, A: transmationProbability 和B: emissionProbability
```

```
model.startprob_ = start_probability
model.transmat_ = transition_probability
model.emissionprob_ = emission_probability


X1 = [0,1,2]

calculateLikelyHood(model, X1)
optimizeStates(model, X1)

X2 = [0,0,0]

calculateLikelyHood(model, X2)
optimizeStates(model, X2)
```

输出结果：

```
[CalculateLikelyHood]:

observations:
  Ruby
  Pearl
  Coral

likelyhood: 0.021792431999999997


[OptimizeStates]:

observations:
  Ruby
  Pearl
  Coral

states:
  Gold
  Silver
  Bronze


[CalculateLikelyHood]:

observations:
  Ruby
  Ruby
  Ruby

likelyhood: 0.03437683199999999


[OptimizeStates]:
```

```
observations:
  Ruby
  Ruby
  Ruby

states:
  Bronze
  Gold
  Bronze
```

> 注意：之前我们讲解过的线性回归、朴素贝叶斯、逻辑回归和决策树，其背后的数学原理的难度属于初级。
>
> 随后的 SVM 和 SVR 对应的数学原理难度就上升到了中级。
>
> 而现在讲解的 HMM 和之后要讲的 CRF，难度则达到了高级，比 SVM 更是提升了一个层次。
>
> 之前中级的 SVM，我们已经用了5节课来专门讲解其背后的原理，HMM 和 CRF 的数学原理及推导过程更是涉及场论、变分等高阶数学知识。
>
> 指望再像之前一样，只借助极少的微积分概率论和线性代数知识，根据直觉进行理解已不切实际。
>
> 因此，对于 HMM 和 CRF 的用法和学习过程，仅点到为止，不再深入到一个个细节。