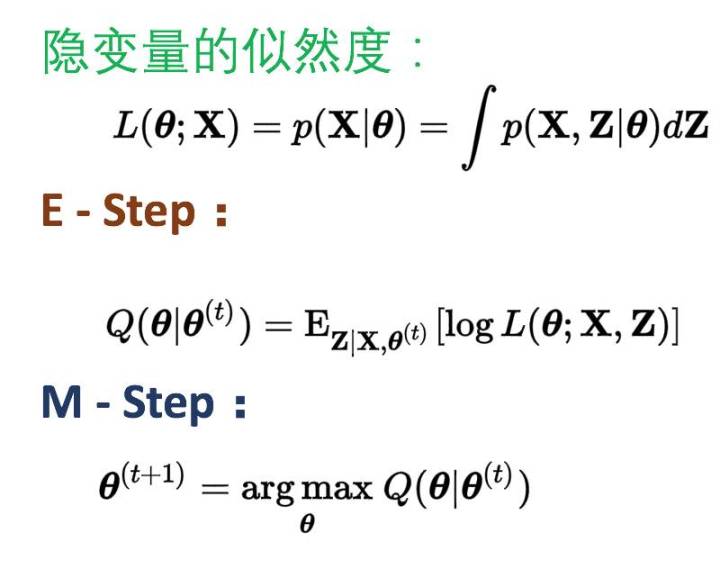
1. （必填）自己提出的问题的理解（罗列全部）：
2. 提出的问题1：176页实例，E步是计算的P(B|θ,y),并不是P(B),为什么迭代过程中好像当成一回事了

讨论后的理解：此处理解有误，M步计算的P(A),P(B),P(C)是根据E步计算的条件概率和观测结果，基于统计结果（符合条件个数/总个数）得出来的。下一次迭代再根据新的theta计算新的条件概率。

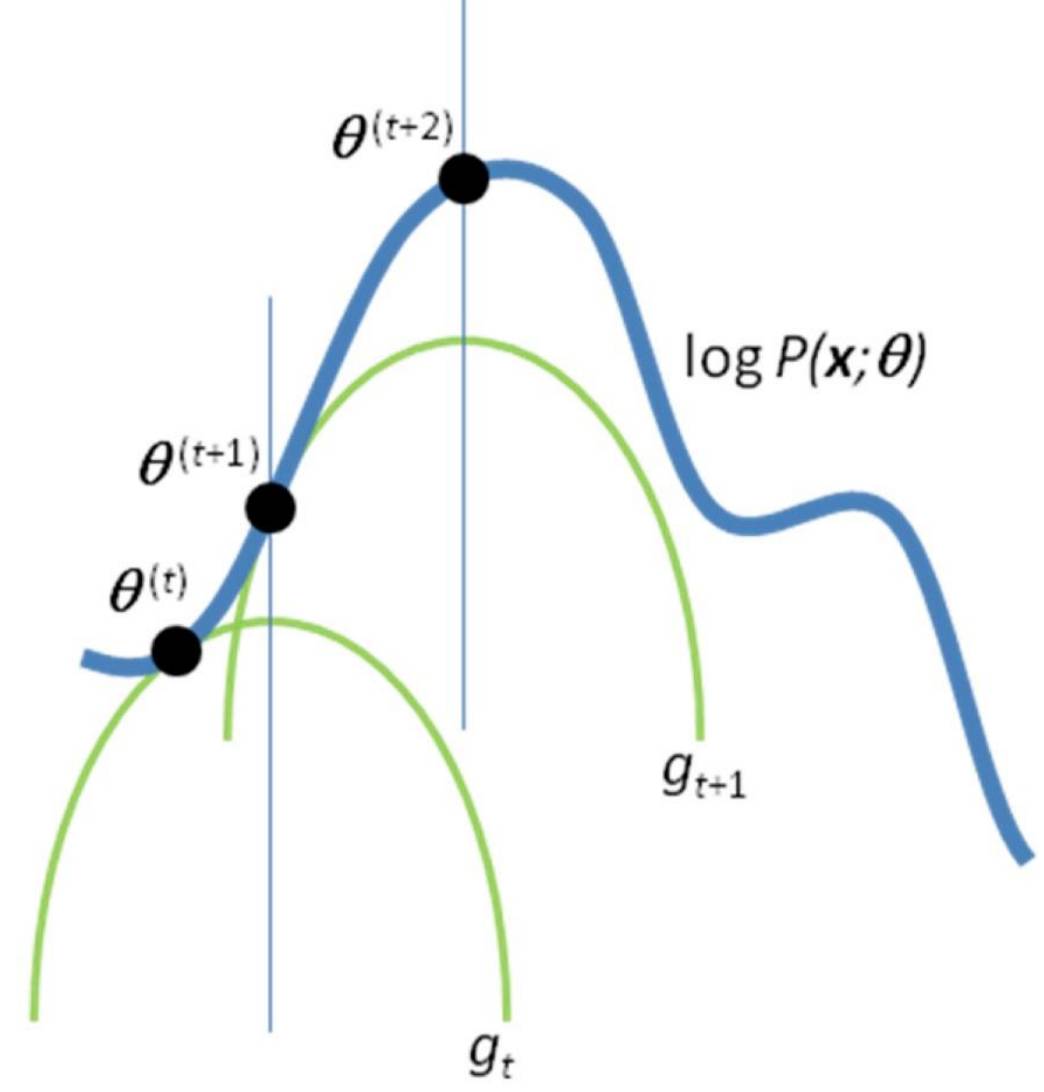
提出的问题2：EM算法存在的意义是什么

讨论后的理解：

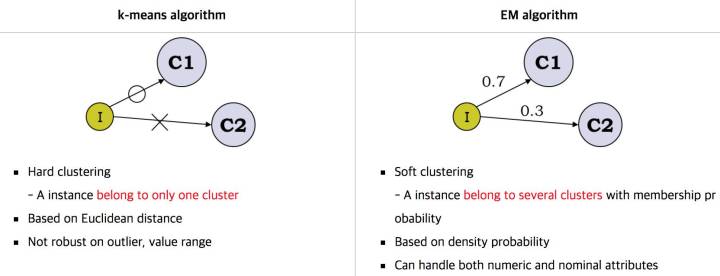
1）EM就是E+M,最经典的例子就是三个硬币，当存在隐变量I的时候， 直接的最大似然估计无法直接搞定。 什么是隐变量？为什么要引入隐变量？ 对隐变量的理解是理解EM算法的第一要义



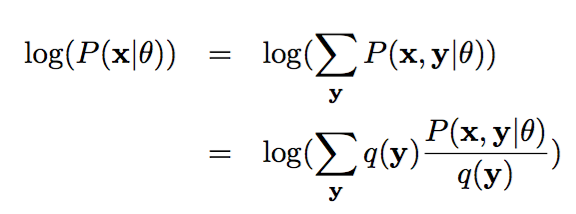
2）EM算法是一种局部下限构造。我们很容易证明，EM算法是在反复的构造新的下限，然后进一步求解。所以，先固定当前参数， 计算得到当前隐变量分布的一个下届函数， 然后优化这个函数， 得到新的参数，然后循环继续。



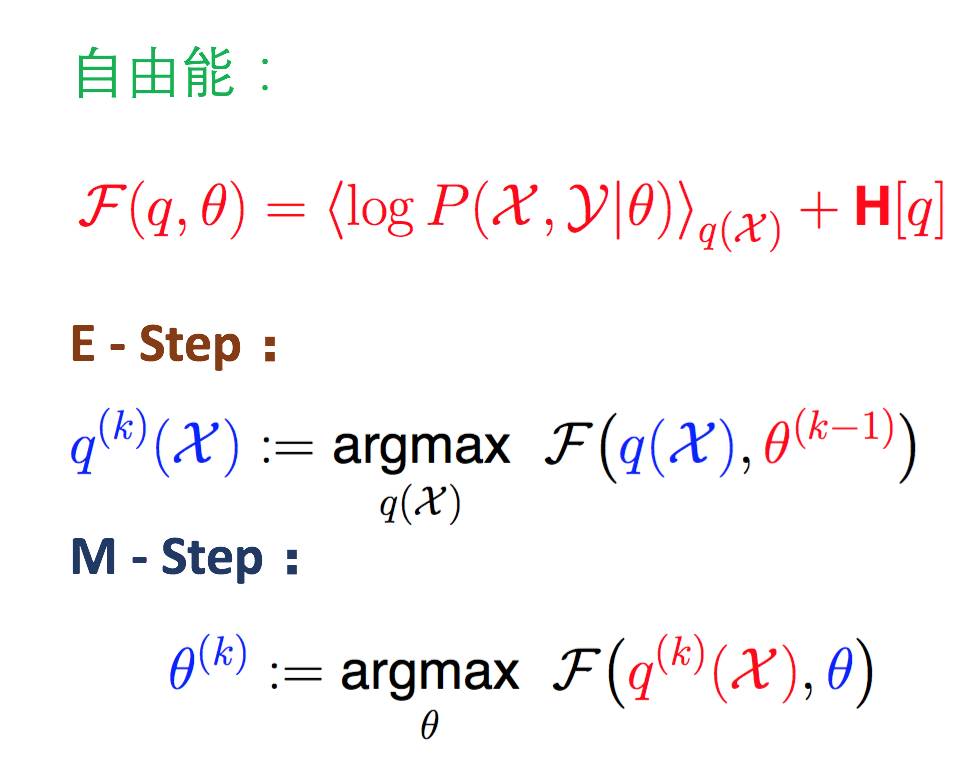
3）K-均值方法是一种Hard EM算法。所谓hard 就是要么是，要么不是0-1抉择。 而Soft是0.7比例是c1，0.3比例是c2的情况。



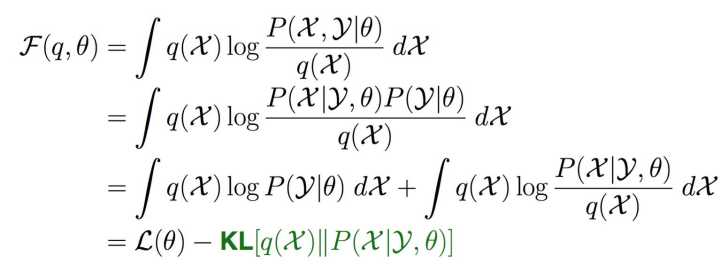
那么充分理解了k-均值和EM算法本身的演化和差异有什么帮助呢？**让你进一步理解到隐变量是存在一种分布的**。



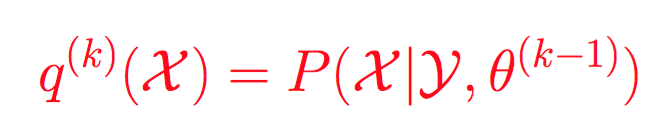
4）EM是广义EM的特例。我们把Jensen不等式右边的部分定义为自由能（那么**E步骤是固定参数优化隐分布， M步骤是固定隐分布优化参数，这就是广义EM算法了**。



有了广义EM算法之后， 我们对自由能深入挖掘， 发现自由能和似然度和KL距离之间的关系：



所以固定参数的情况下， 那么只能最优化KL距离了， 那么隐分布只能取如下分布：



而这个在EM算法里面是直接给出的。 所以EM算法是广义EM算法的天然最优的隐分布情况。 但是很多时候隐分布不是那么容易计算的！

1. （必填）别人提出的问题的理解（选择几个问题罗列，并给出理解）：
2. 问题3：P178页步骤三，每次迭代使似然函数增大或达到局部极值是怎么证明的 ？

自己的理解：

要证收敛，只需证对数似然函数的值一直在增大（单调有界可以推收敛，联合概率必小于等于1，对数似然函数必小于等于零），即

由

令

两式相减得

上式分别取和，相减得

现在证明上式非负，由于使极大，因此

而

|  |  |  |
| --- | --- | --- |
|  |  | (3) |
|  |  | (4) |
|  |  | (5) |

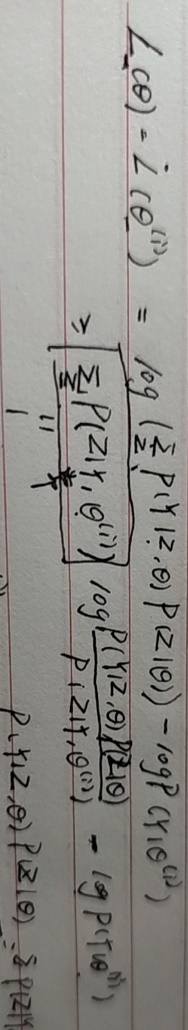
其中（4）式是Jenson不等式的逆用，（5）式是概率之和为1。

综上，

得证，参数收敛于某一局部最优解，如果是凸的，则可以收敛到全局最优

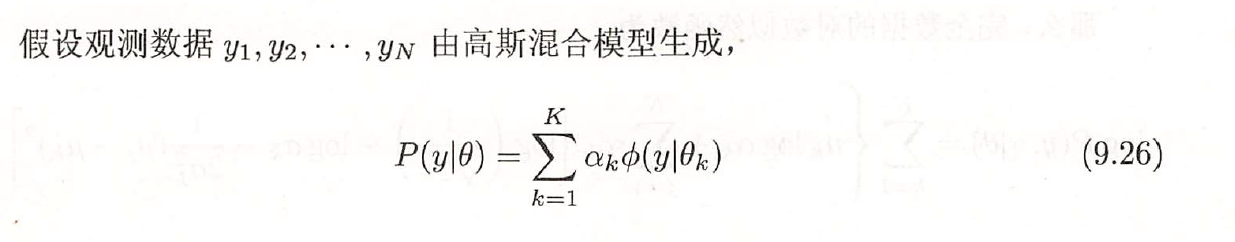
问题4：P179式（9.13）上面的式子怎么推出来的。

自己的理解：框部分是1，因为后面是常数直接给后面加上这个1的部分也不影响



问题5：P185,中间的推导最后一步是怎么出来的。

自己的理解：由高斯混合模型易得



三、（必填）读书计划

1、本周完成的内容章节：李航书第9章

四、（选做）读书摘要及理解或伪代码的具体实现（读书摘要、伪代码的具体实现代码等可以写到这个部分）

1、读书摘要及理解（选做）

#### 1 极大似然估计(Maximum Likelihood Estimation, MLE)

MLE是一种估算模型参数的方法[2]，在已知样本分布和随机样本的条件下计算极大似然函数求极值，从而得到参数的估计值。极大似然函数公式如下

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

其中P(xi;θ)代表在未知参数θ的情况下样本xi(i=1,2,…,n)均服从该分布的联合概率，一般来说我们对极大似然函数先取对数再求极值

#### 2 Jenson不等式

对任意x∈R，f’’(x)≥0（f(x)是凸函数），满足

当且仅当x是常量，不等式取等号。

同理，对于凹函数f(x),满足

2.代码实现

import numpy as np

import math

pro\_A, pro\_B, por\_C = 0.5, 0.5, 0.5

def pmf(i, pro\_A, pro\_B, pro\_C):

pro\_1 = pro\_A \* math.pow(pro\_B, data[i]) \* math.pow((1-pro\_B), 1-data[i])

pro\_2 = pro\_A \* math.pow(pro\_C, data[i]) \* math.pow((1-pro\_C), 1-data[i])

return pro\_1 / (pro\_1 + pro\_2)

class EM:

def \_\_init\_\_(self, prob):

self.pro\_A, self.pro\_B, self.pro\_C = prob

# e\_step

def pmf(self, i):

pro\_1 = self.pro\_A \* math.pow(self.pro\_B, data[i]) \* math.pow((1 - self.pro\_B), 1 - data[i])

pro\_2 = (1 - self.pro\_A) \* math.pow(self.pro\_C, data[i]) \* math.pow((1 - self.pro\_C), 1 - data[i])

return pro\_1 / (pro\_1 + pro\_2)

# m\_step

def fit(self, data):

count = len(data)

print('init prob:{}, {}, {}'.format(self.pro\_A, self.pro\_B, self.pro\_C))

for d in range(count):

\_ = yield

\_pmf = [self.pmf(k) for k in range(count)]

pro\_A = 1 / count \* sum(\_pmf)

pro\_B = sum([\_pmf[k] \* data[k] for k in range(count)]) / sum([\_pmf[k] for k in range(count)])

pro\_C = sum([(1 - \_pmf[k]) \* data[k] for k in range(count)]) / sum([(1 - \_pmf[k]) for k in range(count)])

print('{}/{} pro\_a:{:.3f}, pro\_b:{:.3f}, pro\_c:{:.3f}'.format(d + 1, count, pro\_A, pro\_B, pro\_C))

self.pro\_A = pro\_A

self.pro\_B = pro\_B

self.pro\_C = pro\_C

data=[1,1,0,1,0,0,1,0,1,1]

# em = EM(prob=[0.5, 0.5, 0.5])

# #https://www.ibm.com/developerworks/cn/opensource/os-cn-python-yield/index.html

# #https://blog.csdn.net/qq\_39521554/article/details/79864889

# f = em.fit(data)

# next(f)

# # 第一次迭代

# f.send(1)

# # 第二次

# f.send(2)

em = EM(prob=[0.46, 0.55, 0.67])

f2 = em.fit(data)

next(f2)

f2.send(1)

f2.send(2)