Expectation–maximization 算法的深入理解

大数据机器学习课程第七次作业

叶璨铭 (2024214500) ycm24@mails.tsinghua.edu.cn

2024年12月1日星期日

目录

from scholarly\_infrastructure import logger, print  
from thu\_big\_data\_ml.help import plt, pio

|  |
| --- |
| 重要 |
| 本文档具有一定的交互性，建议使用浏览器打开html文件，这样比pdf文件阅读体验更佳。 |

## 1 第一题——一个例子理解 EM 算法

例9.1（三硬币模型）假设有3枚硬币，分别记作A，B，C.这些硬币正面出现的概率分别是，p和q.进行如下掷硬币试验：先掷硬币A，根据其结果选出硬币B或硬币C；正面选硬币B，反面选硬币C；然后掷选出的硬币，掷硬币的结果，出现正面记作1，出现反面记作0；独立地重复n次试验(这里，n=10)，观测结果如下:

1,1,0,1,0,0,1,0,1,1

假设只能观测到掷硬币的结果，不能观测掷硬币的过程.问如何估计三硬币正面出现的概率，即三硬币模型的参数。

题目:如例9.1的三硬币模型。假设观测数据不变，试选择不同的初值，例如，,求模型参数的极大似然估计。

表1：应聘人员情况数据表

|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 身体 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 业务能力 | 1 | 3 | 2 | 1 | 2 | 1 | 1 | 1 | 3 | 2 |
| 发展潜力 | 3 | 1 | 2 | 3 | 3 | 2 | 2 | 1 | 1 | 1 |
| 分类 | -1 | -1 | -1 | -1 | -1 | -1 | 1 | 1 | -1 | -1 |

### 1.1 审题

我们首先复习一下李航书上的内容和课件上的内容。

* 抽象来说，整个问题的框架是什么？
* 为什么不能直接求解？如果使用梯度下降方法来求解，为什么没有 EM 算法好？
* EM 算法的思想是什么？
* EM 算法 和其他交替 (alternating) 优化算法有什么区别？
* Jensen 不等式是什么？
* EM 算法的强大之处在哪里？
  + 注意
* EM 算法 和 迭代尺度法有没有关系？

复习之后，我们来看看题目询问的问题。本题需要我们尝试和李航书例题解答不同的初值来尝试求解。

首先 数据 如下

import numpy as np  
import plotly.express as px

X = np.array([1,1,0,1,0,0,1,0,1,1])  
px.histogram(x=X, nbins=2)

Unable to display output for mime type(s): text/html

pi, p, q = 0.46, 0.55, 0.67  
px.bar(x=['π', 'p', 'q'], y=[pi, p, q], title='Bernoulli Distribution Parameters', labels={'x':'硬币A (决定抛B还是C), 硬币B, 硬币C', 'y': 'P(X=1)'})

Unable to display output for mime type(s): text/html

接下来尝试使用 EM 算法来迭代求解这个“混合伯努利分布”极大似然估计。

### 1.2 解题

#### 1.2.1 解法一：手工推导法（考试写解答题）

##### 1.2.1.1 第一轮迭代，我们有初始的参数值

假如这个参数值当做真的参数值，我们可以算什么呢？

* 不完全数据的对数似然函数 是可以计算的，虽然我们不能得到解析解，但是这个似然的值至少是可以计算的，可以知道我们现在这个参数估计的好不好。
* 如果我们有Z的具体值 (也就是每一个样本 是来自硬币B还是C)，完全数据的 就能算出一个值，不过我们自然是没有 Z 的具体值，否则 可以直接极大似然估计了。
* 但是我们有, 而且我们有 的计算公式，所以我们是可以估算出来，每一个样本 既然等于1或者0，那么在我的 下 它来自Z的概率是多少。

所以我们首先来计算此概率。Z 无非 取值 1, 0 (硬币B,硬币C)， 所以根据贝叶斯公式

同理

这个式子和 李航《统计学习方法》 9.5 式一致，我们使用了 表示归一化参数, 也就是贝叶斯公式的分母。其中注意 无非就是 伯努利分布的概率密度函数，实际上是把 和 的情况合并在一起表示。

我们代入数值计算一下，按照上面列出的公式

def get\_p\_z\_given\_x\_theta(X, pi, p, q):  
 p\_z\_eq1\_given\_x\_theta = (p\*\*X) \* ((1-p)\*\*(1-X)) \* pi  
 p\_z\_eq0\_given\_x\_theta = (q\*\*X) \* ((1-q)\*\*(1-X)) \* (1-pi)  
 eta = 1/ (p\_z\_eq1\_given\_x\_theta + p\_z\_eq0\_given\_x\_theta)  
 p\_z\_eq1\_given\_x = eta \* p\_z\_eq1\_given\_x\_theta  
 p\_z\_eq0\_given\_x = eta \* p\_z\_eq0\_given\_x\_theta  
 return p\_z\_eq1\_given\_x, p\_z\_eq0\_given\_x  
p\_z\_eq1\_given\_x, p\_z\_eq0\_given\_x = get\_p\_z\_given\_x\_theta(X, pi, p, q)  
p\_z\_eq1\_given\_x

array([0.41151594, 0.41151594, 0.53738318, 0.41151594, 0.53738318,  
 0.53738318, 0.41151594, 0.53738318, 0.41151594, 0.41151594])

上面的值意味着给定参数时，每一个样本是被 硬币 B 得到的概率。 考试的时候，我们不需要每一个样本都算一次，因为 无非是0或1，我们只需要计算 和

李航书上说这就完成了 **Expectation Step**，对于硬币这个混合伯努利分布模型确实到这一步可以算下一步 Maximization 了，但是我们这里把概念搞清楚一点。EM算法所求解的是Q函数，也就是一个期望，不只是 Z 的概率分布。

具体这个期望是谁的期望呢？要说明一个期望，第一需要知道采样这个期望的概率分布是什么分布，第二要知道期望想要知道的函数值是什么。 我们简单复习一下概率论的知识 本来期望的定义是对一个随机变量而言的，但是因为有这个基本公式，只要是另外一个随机变量是原本随机变量的函数关系，就可以用原本的随机变量的分布来采样函数值，得到目标随机变量的期望。

对于EM算法而言，经过Jensen不等式的推导，我们感兴趣的期望是 也就是 完全数据的对数使然函数，然后因为我们没有Z的确切值但是有Z的估计分布，所以我们可以用这个分布去采样，也就是关注的重点在于 。

注意这里这个式子的概率论符号我认为是很有必要进行解释的，否则我们都无法知道我们到底在讨论什么。

* 首先 的意思是右边这个式子中间内部使用了一些变量，但是这些变量最后都和最后的值无关，比如说被求和和积分“积掉了（reduce了）”，最后只有 （上一轮的参数估计值或者第一轮的先验）和变量决定右边式子的值，据此下一步 Maximization 才能够优化 。
* , 下标代表用来估计期望的密度函数，李航书上写的是 ，这个是指用来采样用的那个随机变量是Z，我们表达的更加精确，我们说的是密度函数是我们上面求出的 。 括号内的 竖线表示 **条件期望**，也是决定用来估计期望的密度函数的一个表示方法。

那为什么李航书上我们没有继续去求 , 而是在 戛然而止了呢？

这是因为我们在**作业1** 已经详细推导过了 伯努利分布模型的 极大似然估计和贝叶斯估计，我们知道结论是什么，我们不需要再次推导一遍（还是稍有区别，因为Z不是固定值而是分布）。

我们已经知道 有了 的情况下，极大 估计的结果，所以我们可以直接进入到 **Maximization Step**

首先极大似然估计 这个硬币A伯努利分布的参数，如果Z是固定值，极大似然估计无非就是看看有多少个1多少个0，计算个频率即可，贝叶斯估计无非就是增加一个Beta分布的先验，先验的多几个1和0的统计结果罢了。现在Z是 ，经过李航书上省略的推导，要最大化Expectation Step的期望，本质上无非就是计算频率的时候用Z的概率值来加权计算。所以

pi = p\_z\_eq1\_given\_x\_theta.mean()  
pi

0.23460000000000006

而p也是伯努利分布，表示B硬币的正面概率，如果你告诉我X数据里面哪些值是B抛出了，极大似然估计无非就是这个X数据的子集里面，0有多少个，1有多少个。但是我当然不知道Z的具体分配，我只有Z的估计的分布。李航书上再次省略了推导，但是很容易理解，如果我们要找到一个p最大化期望，本质上无非就是整个X我都统计0和1，不过每一个x\_i的频率贡献是根据Z的概率值贡献加权的。所以

同理

p = sum(p\_z\_eq1\_given\_x\_theta\* X)/ sum(p\_z\_eq1\_given\_x\_theta)  
q = sum(p\_z\_eq0\_given\_x\_theta\* X)/ sum(p\_z\_eq0\_given\_x\_theta)  
p, q

(0.6470588235294118, 0.7528089887640449)

##### 1.2.1.2 第二轮迭代

同理

p\_z\_eq1\_given\_x, p\_z\_eq0\_given\_x = get\_p\_z\_given\_x\_theta(X, pi, p, q)  
pi = p\_z\_eq1\_given\_x\_theta.mean()  
p = sum(p\_z\_eq1\_given\_x\_theta\* X)/ sum(p\_z\_eq1\_given\_x\_theta)  
q = sum(p\_z\_eq0\_given\_x\_theta\* X)/ sum(p\_z\_eq0\_given\_x\_theta)  
pi, p, q

(0.23460000000000006, 0.6470588235294118, 0.7528089887640449)

我们发现我们已经获得了一个稳定的解，已经稳定，收敛到到了局部最优值，因而EM算法结束。

## 2 第二题——混合高斯分布上使用 EM 算法

题目如下 > 已知观测数据 -67, -48, 6, 8, 14, 16, 23, 24, 28, 29, 41, 49, 56, 60, 75，试估计两个分量的高斯混合模型的5个参数。

### 2.1 审题

import plotly.figure\_factory as ff  
from nbdev.showdoc import show\_doc

# 已知观测数据  
data = np.array([-67, -48, 6, 8, 14, 16, 23, 24, 28, 29, 41, 49, 56, 60, 75])  
fig = px.histogram(x=data,   
 marginal="box", # or violin, rug  
 )  
# hist\_data = [data]  
# group\_labels = ['distplot'] # name of the dataset  
  
# fig = ff.create\_distplot(hist\_data, group\_labels)  
fig

Unable to display output for mime type(s): text/html

### 2.2 解题

#### 2.2.1 解法一：手工推导法（考试写解答题）

上一题我们已经详细解释了EM算法，使用EM算法估计了 混合伯努利分布。

这一题是混合高斯分布，其实混合高斯分布的EM的公式是类似的，这一题两个分量也就是上一题的硬币A，我们也是先计算每一个样本属于哪个分量的概率 ，然后使用概率作为加权，去极大似然估计参数 。估计的公式和上面完全一样，因为是伯努利分布。

而估计两个分量的高斯分布的参数的公式则和上面有所不同。高斯分布的极大似然估计，无非就是计算数据子集的均值和方差。而在EM当中，我们需要推导怎么极大化 Q函数，实际上对于高斯混合分布，和伯努利混合分布类似，得到的结论无非就是 以概率 加权 计算均值和方差。

由于上一题我们已经手算推导过，我们这一题不再重复，而是来探究一下**解法二：使用Python和机器学习库**，我们这一题来探究一下，在现实科研场景下，我们怎么使用EM算法，以混合高斯分布为例。

#### 2.2.2 解法二：使用Python和机器学习库（科研使用）

我们使用sklearn来使用EM算法来对高斯混合模型进行训练。

from sklearn.mixture import GaussianMixture

GaussianMixture?

Init signature:  
GaussianMixture(  
 n\_components=1,  
 \*,  
 covariance\_type='full',  
 tol=0.001,  
 reg\_covar=1e-06,  
 max\_iter=100,  
 n\_init=1,  
 init\_params='kmeans',  
 weights\_init=None,  
 means\_init=None,  
 precisions\_init=None,  
 random\_state=None,  
 warm\_start=False,  
 verbose=0,  
 verbose\_interval=10,  
)  
Docstring:   
Gaussian Mixture.  
  
Representation of a Gaussian mixture model probability distribution.  
This class allows to estimate the parameters of a Gaussian mixture  
distribution.  
  
Read more in the :ref:`User Guide <gmm>`.  
  
.. versionadded:: 0.18  
  
Parameters  
----------  
n\_components : int, default=1  
 The number of mixture components.  
  
covariance\_type : {'full', 'tied', 'diag', 'spherical'}, default='full'  
 String describing the type of covariance parameters to use.  
 Must be one of:  
  
 - 'full': each component has its own general covariance matrix.  
 - 'tied': all components share the same general covariance matrix.  
 - 'diag': each component has its own diagonal covariance matrix.  
 - 'spherical': each component has its own single variance.  
  
tol : float, default=1e-3  
 The convergence threshold. EM iterations will stop when the  
 lower bound average gain is below this threshold.  
  
reg\_covar : float, default=1e-6  
 Non-negative regularization added to the diagonal of covariance.  
 Allows to assure that the covariance matrices are all positive.  
  
max\_iter : int, default=100  
 The number of EM iterations to perform.  
  
n\_init : int, default=1  
 The number of initializations to perform. The best results are kept.  
  
init\_params : {'kmeans', 'k-means++', 'random', 'random\_from\_data'}, default='kmeans'  
 The method used to initialize the weights, the means and the  
 precisions.  
 String must be one of:  
  
 - 'kmeans' : responsibilities are initialized using kmeans.  
 - 'k-means++' : use the k-means++ method to initialize.  
 - 'random' : responsibilities are initialized randomly.  
 - 'random\_from\_data' : initial means are randomly selected data points.  
  
 .. versionchanged:: v1.1  
 `init\_params` now accepts 'random\_from\_data' and 'k-means++' as  
 initialization methods.  
  
weights\_init : array-like of shape (n\_components, ), default=None  
 The user-provided initial weights.  
 If it is None, weights are initialized using the `init\_params` method.  
  
means\_init : array-like of shape (n\_components, n\_features), default=None  
 The user-provided initial means,  
 If it is None, means are initialized using the `init\_params` method.  
  
precisions\_init : array-like, default=None  
 The user-provided initial precisions (inverse of the covariance  
 matrices).  
 If it is None, precisions are initialized using the 'init\_params'  
 method.  
 The shape depends on 'covariance\_type'::  
  
 (n\_components,) if 'spherical',  
 (n\_features, n\_features) if 'tied',  
 (n\_components, n\_features) if 'diag',  
 (n\_components, n\_features, n\_features) if 'full'  
  
random\_state : int, RandomState instance or None, default=None  
 Controls the random seed given to the method chosen to initialize the  
 parameters (see `init\_params`).  
 In addition, it controls the generation of random samples from the  
 fitted distribution (see the method `sample`).  
 Pass an int for reproducible output across multiple function calls.  
 See :term:`Glossary <random\_state>`.  
  
warm\_start : bool, default=False  
 If 'warm\_start' is True, the solution of the last fitting is used as  
 initialization for the next call of fit(). This can speed up  
 convergence when fit is called several times on similar problems.  
 In that case, 'n\_init' is ignored and only a single initialization  
 occurs upon the first call.  
 See :term:`the Glossary <warm\_start>`.  
  
verbose : int, default=0  
 Enable verbose output. If 1 then it prints the current  
 initialization and each iteration step. If greater than 1 then  
 it prints also the log probability and the time needed  
 for each step.  
  
verbose\_interval : int, default=10  
 Number of iteration done before the next print.  
  
Attributes  
----------  
weights\_ : array-like of shape (n\_components,)  
 The weights of each mixture components.  
  
means\_ : array-like of shape (n\_components, n\_features)  
 The mean of each mixture component.  
  
covariances\_ : array-like  
 The covariance of each mixture component.  
 The shape depends on `covariance\_type`::  
  
 (n\_components,) if 'spherical',  
 (n\_features, n\_features) if 'tied',  
 (n\_components, n\_features) if 'diag',  
 (n\_components, n\_features, n\_features) if 'full'  
  
precisions\_ : array-like  
 The precision matrices for each component in the mixture. A precision  
 matrix is the inverse of a covariance matrix. A covariance matrix is  
 symmetric positive definite so the mixture of Gaussian can be  
 equivalently parameterized by the precision matrices. Storing the  
 precision matrices instead of the covariance matrices makes it more  
 efficient to compute the log-likelihood of new samples at test time.  
 The shape depends on `covariance\_type`::  
  
 (n\_components,) if 'spherical',  
 (n\_features, n\_features) if 'tied',  
 (n\_components, n\_features) if 'diag',  
 (n\_components, n\_features, n\_features) if 'full'  
  
precisions\_cholesky\_ : array-like  
 The cholesky decomposition of the precision matrices of each mixture  
 component. A precision matrix is the inverse of a covariance matrix.  
 A covariance matrix is symmetric positive definite so the mixture of  
 Gaussian can be equivalently parameterized by the precision matrices.  
 Storing the precision matrices instead of the covariance matrices makes  
 it more efficient to compute the log-likelihood of new samples at test  
 time. The shape depends on `covariance\_type`::  
  
 (n\_components,) if 'spherical',  
 (n\_features, n\_features) if 'tied',  
 (n\_components, n\_features) if 'diag',  
 (n\_components, n\_features, n\_features) if 'full'  
  
converged\_ : bool  
 True when convergence of the best fit of EM was reached, False otherwise.  
  
n\_iter\_ : int  
 Number of step used by the best fit of EM to reach the convergence.  
  
lower\_bound\_ : float  
 Lower bound value on the log-likelihood (of the training data with  
 respect to the model) of the best fit of EM.  
  
n\_features\_in\_ : int  
 Number of features seen during :term:`fit`.  
  
 .. versionadded:: 0.24  
  
feature\_names\_in\_ : ndarray of shape (`n\_features\_in\_`,)  
 Names of features seen during :term:`fit`. Defined only when `X`  
 has feature names that are all strings.  
  
 .. versionadded:: 1.0  
  
See Also  
--------  
BayesianGaussianMixture : Gaussian mixture model fit with a variational  
 inference.  
  
Examples  
--------  
>>> import numpy as np  
>>> from sklearn.mixture import GaussianMixture  
>>> X = np.array([[1, 2], [1, 4], [1, 0], [10, 2], [10, 4], [10, 0]])  
>>> gm = GaussianMixture(n\_components=2, random\_state=0).fit(X)  
>>> gm.means\_  
array([[10., 2.],  
 [ 1., 2.]])  
>>> gm.predict([[0, 0], [12, 3]])  
array([1, 0])  
File: ~/program\_files/managers/conda/envs/yuequ/lib/python3.10/site-packages/sklearn/mixture/\_gaussian\_mixture.py  
Type: ABCMeta  
Subclasses:

# 将数据转换为二维数组，因为根据上面的文档，GMM需要二维输入  
data\_2d = data.reshape(-1, 1)

# 创建GMM模型，指定两个分量  
gmm = GaussianMixture(n\_components=2,   
 random\_state=0, verbose=2)  
  
# 拟合模型  
gmm.fit(data\_2d)

Initialization 0  
Initialization converged. time lapse 0.21671s lower bound -4.73756.

GaussianMixture(n\_components=2, random\_state=0, verbose=2)

我们看到GMM都还没开始，Initialization的结果就已经让它收敛了，这是因为 GMM 使用了 KMeans的结果来作为初始化。 我们再来试试随机初始化。

gmm = GaussianMixture(n\_components=2, init\_params = "random",   
 random\_state=0, verbose=2, verbose\_interval=1)  
  
# 拟合模型  
gmm.fit(data\_2d)

Initialization 0  
 Iteration 1 time lapse 0.01017s ll change inf  
 Iteration 2 time lapse 0.00344s ll change 0.00766  
 Iteration 3 time lapse 0.00336s ll change 0.02146  
 Iteration 4 time lapse 0.00226s ll change 0.04780  
 Iteration 5 time lapse 0.00037s ll change 0.04391  
 Iteration 6 time lapse 0.00031s ll change 0.01010  
 Iteration 7 time lapse 0.00032s ll change 0.00443  
 Iteration 8 time lapse 0.00135s ll change 0.00365  
 Iteration 9 time lapse 0.00378s ll change 0.00317  
 Iteration 10 time lapse 0.00309s ll change 0.00283  
 Iteration 11 time lapse 0.00342s ll change 0.00261  
 Iteration 12 time lapse 0.00346s ll change 0.00253  
 Iteration 13 time lapse 0.00258s ll change 0.00260  
 Iteration 14 time lapse 0.00032s ll change 0.00287  
 Iteration 15 time lapse 0.00030s ll change 0.00348  
 Iteration 16 time lapse 0.00041s ll change 0.00467  
 Iteration 17 time lapse 0.00313s ll change 0.00709  
 Iteration 18 time lapse 0.00334s ll change 0.01260  
 Iteration 19 time lapse 0.00326s ll change 0.02772  
 Iteration 20 time lapse 0.00318s ll change 0.05395  
 Iteration 21 time lapse 0.00339s ll change 0.01015  
 Iteration 22 time lapse 0.00117s ll change 0.00000  
Initialization converged. time lapse 0.05644s lower bound -4.73756.

GaussianMixture(init\_params='random', n\_components=2, random\_state=0, verbose=2,  
 verbose\_interval=1)

这一次则是使用了22次迭代，EM算法才收敛。

我们来看看模型估计的结果是什么样的。

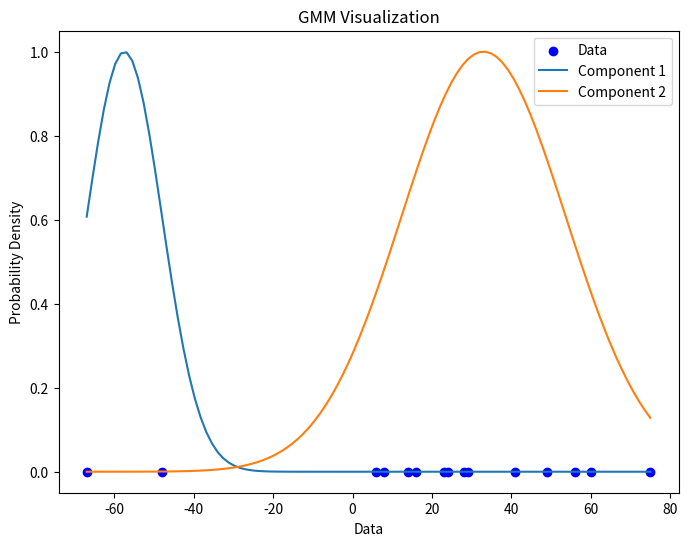
# 提取参数  
weights = gmm.weights\_  
means = gmm.means\_  
covariances = gmm.covariances\_  
weights, means, covariances

(  
 array([0.13317228, 0.86682772]),  
 array([[-57.51107672],  
 [ 32.98488751]]),  
 array([[[ 90.24987868]],  
  
 [[429.45832976]]])  
)

mean和covariances就是估计出来的两个高斯分布分量的均值和协方差矩阵，因为这里是一维的，协方差矩阵只有一个元素就是方差。 weights则表示这两个分量分别占多少，也就是先抛一枚硬币A决定用哪个分量的这个概率。

我们可以可视化一下拟合出来的混合高斯分布，如下所示：

# 可视化数据和GMM模型  
plt.figure(figsize=(8, 6))  
plt.scatter(data, np.zeros\_like(data), c='blue', label='Data')  
  
# 绘制GMM模型的等高线  
x = np.linspace(data.min(), data.max(), 100)  
for i, (mean, cov) in enumerate(zip(means, covariances)):  
 y = np.exp(-0.5 \* ((x - mean[0]) \*\* 2 / cov[0, 0]))  
 plt.plot(x, y, label=f'Component {i+1}')  
  
plt.title('GMM Visualization')  
plt.xlabel('Data')  
plt.ylabel('Probability Density')  
plt.legend()  
plt.show()



其中 y = np.exp(-0.5 \* ((x - mean[0]) \*\* 2 / cov[0, 0])) 是高斯分布的密度函数。

我们还可以计算一下在这个参数下，每一个样本属于分量的概率。

gmm.predict\_proba(data\_2d)

array([[9.99956686e-01, 4.33138991e-05],  
 [9.97627532e-01, 2.37246752e-03],  
 [1.54227685e-10, 1.00000000e+00],  
 [3.27141378e-11, 1.00000000e+00],  
 [2.53066088e-13, 1.00000000e+00],  
 [4.66651997e-14, 1.00000000e+00],  
 [9.53724105e-17, 1.00000000e+00],  
 [3.80184958e-17, 1.00000000e+00],  
 [8.79579073e-19, 1.00000000e+00],  
 [3.35616030e-19, 1.00000000e+00],  
 [1.61505874e-24, 1.00000000e+00],  
 [2.28568088e-28, 1.00000000e+00],  
 [6.18648047e-32, 1.00000000e+00],  
 [4.66887134e-34, 1.00000000e+00],  
 [1.47644926e-42, 1.00000000e+00]])

我们可以看到，前两个样本有较高的概率属于第一个分量，而后面所有样本几乎百分之一百属于第二个分量。

sklearn库只能求解 一些有限的 机器学习模型，有没有通用的用来解决 EM 算法的 隐变量估计问题的算法呢？答案是肯定的。

pip install mixem

from mixem import em  
from mixem.distribution import NormalDistribution

# 我们来定义分布  
distributions = [  
 NormalDistribution(mu=0, sigma=1),  
 NormalDistribution(mu=0, sigma=1)  
]

# 使用 0.5, 0.5 作为两个分量选择的初始值，开始EM算法  
weights, distributions, log\_likelihood = em(data, distributions, initial\_weights=[0.5, 0.5])

iteration 0 (log-likelihood=nan): p(x|Φ) = nan\*Norm[μ=nan, σ=nan] + nan\*Norm[μ=nan, σ=nan]

然而很可惜，这个库似乎按照其官网教程并没有正确工作。我们下次再探索。

### 2.3 题目扩展问题