学　　期　2021-2022（2）

北京航空航天大學
BEIHANG UNIVERSITY

深度学习与自然语言处理第二次大作业

EM 算法

院（系）名称	自动化科学与电气工程学院
专业名称	模式识别
学生姓名	殷健凯
学号	SY2103130
指导老师	秦曾昌

2022 年　4 月

1 问题描述

一个袋子中三种硬币的混合比例为：s1, s2 与 1-s1-s2 (0<=si<=1), 三种硬币掷出正面的概率分别为：p, q, r。指定系数 s1, s2, p, q, r，生成 N 个投掷硬币的结果（由 01 构成的序列，其中 1 为正面，0 为反面），利用 EM 算法来对参数进行估计并与预先假定的参数进行比较。

2 EM 算法

EM 算法是一种迭代优化策略，由于它的计算方法中每一次迭代都分两步，其中一个为期望步（E 步），另一个为极大步（M 步），所以算法被称为 EM 算法（Expectation-Maximization Algorithm）。EM 算法受到缺失思想影响，最初是为了解决数据缺失情况下的参数估计问题，其算法基础和收敛有效性等问题在 Dempster、Laird 和 Rubin 三人于 1977 年所做的文章《Maximum likelihood from incomplete data via the EM algorithm》中给出了详细的阐述。其基本思想是：首先根据已经给出的观测数据，估计出模型参数的值；然后再依据上一步估计出的参数值估计缺失数据的值，再根据估计出的缺失数据加上之前已经观测到的数据重新再对参数值进行估计，然后反复迭代，直至最后收敛，迭代结束。

3 算法流程

输入：观察到的数据 x=（x1,x2...xn），联合分布 p(x,z;Θ)，条件分布 p(x|z,Θ) ，最大迭代次数 J。

(1)随机初始化模型参数Θ的初值，Θ_0

(2)j=1,2,...,J 开始 EM 算法迭代：

E 步：计算联合分布的条件概率期望：

$$Q_i(z_i|x_i,\theta_j)$$

$$l(\theta,\theta_j)=\sum_{i=1}^{n}\sum_{z_i}Q_i(z_i)\log\frac{p(x_i,z_i;\theta)}{Q_i(z_i)}$$

M 步：极大化 $l(\theta,\theta_j)$，得到θ_{j+1}

若θ_{j+1}已经收敛，则算法结束，否则继续迭代 E 步和 M 步。

输出：模型参数Θ

4 公式推导

设初始参数构成为$\theta=[s_1,s_2,p,q,r]$，共 M 枚硬币（由这三类硬币构成），每枚硬币投掷 N 次，则投掷硬币的结果可表示为一个 M 行 N 列的矩阵 result，可表示为：

$$result=[[x_1^1,x_1^2\dots x_1^N],\dots[x_M^1,x_M^2\dots,x_M^N]]$$

根据已知参数，可以推断出：

各个种类的硬币掷出的条件分布概率为：

$$P(x_m^n|kind1)=x_m^n p+(1-x_m^n)(1-p)$$
$$P(x_m^n|kind2)=x_m^n q+(1-x_m^n)(1-q)$$
$$P(x_m^n|kind2)=x_m^n r+(1-x_m^n)(1-r)$$

联合三类硬币出现的概率，可以得到投掷硬币结果的联合分布概率：

$$\begin{aligned}P(x_m^n|\theta)&=s_1\cdot P(x_m^n|kind1)+s_2\cdot P(x_m^n|kind2)+(1-s_1-s_2)\cdot P(x_m^n|kind3)\\&=s_1\cdot[x_m^n p+(1-x_m^n)(1-p)]+s_2\cdot[x_m^n q+(1-x_m^n)(1-q)]+\\&\quad(1-s_1-s_2)\cdot[x_m^n r+(1-x_m^n)(1-r)]\end{aligned}$$

由此，可以估算出三种硬币的概率为：

$$P(kind1)=\frac{P(x_m^n|kind1)\cdot s_1}{P(x_m^n|\theta)}$$
$$P(kind2)=\frac{P(x_m^n|kind2)\cdot s_2}{P(x_m^n|\theta)}$$

$$P(kind3) = \frac{P(x_m^n|kind3)\cdot(1-s_1-s_2)}{P(x_m^n|\theta)}$$

再对结果进行加权处理并进行数学运算，可得：

$$s_1 = \frac{\sum_{i=1}^{n} P(kind1)}{n}$$

$$s_2 = \frac{\sum_{i=1}^{n} P(kind2)}{n}$$

那么，每类硬币是正面的概率分布为：

$$p = \frac{\sum_{i=1}^{n} x_m^n P(kind1)}{\sum_{i=1}^{n} P(kind1)}$$

$$q = \frac{\sum_{i=1}^{n} x_m^n P(kind2)}{\sum_{i=1}^{n} P(kind2)}$$

$$r = \frac{\sum_{i=1}^{n} x_m^n P(kind3)}{\sum_{i=1}^{n} P(kind3)}$$

5 实验过程

5.1 生成投掷数据

投掷数据由 M 行 N 列构成，即共有 M 个硬币，每个硬币投掷 N 次，并按照给定概率进行数据生成。生成数据的参数如下表所示：

参数	s1	s2	p	q	r
数值	0.4	0.4	0.3	0.3	0.8

5.2 数据初始化

对数据迭代进行初始化操作。

参数	s1	s2	p	q	r	Epochs
数值	0.1	0.1	0.5	0.5	0.5	50

```
num_prob = [0.4, 0.4, 0.2]
pos_pron = [0.3, 0.3, 0.8]
N = 1000  # 硬币的个数
M = 100  # 每个硬币投掷的次数
results = []  # 记录抛掷的结果，M列N行
epochs = 50  # 迭代轮数
init_parameters = [0.1, 0.1, 0.5, 0.5, 0.5]  # 种类一二比例、种类一二三为正概率
process_parameters = [init_parameters]
for coin in range(N):
    #  第n个硬币
    result = []
    rand_coin = random.randint(0, 99)
    coin_kind = judge_coin_kind(rand_coin)  # 返回当前硬币的种类
    for throw in range(M):
        #  第n个硬币投掷M次
        rand_result = random.randint(0, 99)
        pos_or_neg = judge_coin_pos_or_neg(coin_kind, rand_result)  # 返回硬币的正反面
        result.append(pos_or_neg)
    results.append(result)
```

5.3 E 步

首先计算联合分布概率概率：

```
den = math.pow(init_parameters[2], np.sum(results, 1)) * math.pow(1 - init_parameters[2],
                                                                  M - np.sum(results, 1)) * init_parameters[0] \
      + math.pow(init_parameters[3], np.sum(results, 1)) * math.pow(1 - init_parameters[3],
                                                                    M - np.sum(results, 1)) * init_parameters[1] \
      + math.pow(init_parameters[4], np.sum(results, 1)) * math.pow(1 - init_parameters[4],
                                                                    M - np.sum(results, 1)) * (
                1 - init_parameters[0] - init_parameters[1])
```

再计算条件分布概率：

```
u1_num = math.pow(init_parameters[2], np.sum(results, 1)) * math.pow(1 - init_parameters[2],
                                                                     M - np.sum(results, 1)) * init_parameters[0]
```

```
u2_num = math.pow(init_parameters[3], np.sum(results, 1)) * math.pow(1 - init_parameters[3],
                                                                     M - np.sum(results, 1)) * init_parameters[1]
```

最终计算出三类硬币的分布概率为：

```
u1 = u1_num / den
u2 = u2_num / den
u3 = 1 - u1 - u2
```

5.4 M 步

在 M 步中对参数进行极大似然估计，实现对参数的更新。

6 实验结果

最终结果如下表所示：

参数	s1	s2	p	q	r
真实数值	0.4	0.4	0.3	0.3	0.8
预测数值	0.405	0.405	0.29993827	0.29993827	0.80142105

收敛效果较好，但 EM 算法高度依赖初值的选择，当将初值选择为：

参数	s1	s2	p	q	r
数值	0.1	0.7	0.1	0.1	0.1

最终结果如下表：

参数	s1	s2	p	q	r
真实数值	0.4	0.4	0.3	0.3	0.8
预测数值	0.195	0.45954488	0.7974359	0.30135424	0.29738798

可以看出，EM 算法十分依赖初值的选择，当初值选择的较好时，经过几步就可以迭代完成，但是当初值选择与真值相差较远时，可能陷入局部最优解。

7 代码

```
import random
import numpy as np
import matplotlib.pyplot as plt


def judge_coin_kind(rand):
    if 0 <= rand < num_prob[0] * 100:
        return 0
    elif num_prob[0] * 100 <= rand < (num_prob[1] + num_prob[0]) * 100:
        return 1
    else:
```

```
        return 2


def judge_coin_pos_or_neg(kind, rand):
    if 0 <= rand < pos_pron[kind] * 100:
        return 1
    else:
        return 0


num_prob = [0.4, 0.4, 0.2]
pos_pron = [0.3, 0.3, 0.8]
N = 1000  # 硬币的个数
M = 100  # 每个硬币投掷的次数
results = []  # 记录抛掷的结果，M 列 N 行
epochs = 50  # 迭代轮数
init_parameters = [0.1, 0.7, 0.1, 0.1, 0.1]  # 种类一二比例、种类一二三为正概率
process_parameters = [init_parameters]
for coin in range(N):
    #  第 n 个硬币
    result = []
    rand_coin = random.randint(0, 99)
    coin_kind = judge_coin_kind(rand_coin)  # 返回当前硬币的种类
    for throw in range(M):
        #  第 n 个硬币投掷 M 次
        rand_result = random.randint(0, 99)
        pos_or_neg = judge_coin_pos_or_neg(coin_kind, rand_result)  # 返回硬币的正反面
        result.append(pos_or_neg)
    results.append(result)
# 开始迭代
for epoch in range(epochs):
    u1_num=np.power(init_parameters[2],np.sum(results,1))* np.power(1 - init_parameters[2],M - np.sum(results, 1)) * init_parameters[0]
    den = np.power(init_parameters[2], np.sum(results, 1)) * np.power(1 - init_parameters[2],M - np.sum(results, 1)) * init_parameters[0] + np.power(init_parameters[3], np.sum(results, 1)) * np.power(1-init_parameters[3],M-np.sum(results,1))*init_parameters[1]+np.power(init_parameters[4], np.sum(results, 1)) * np.power(1 - init_parameters[4], M - np.sum(results, 1)) * (1 - init_parameters[0] - init_parameters[1])
    u2_num=np.power(init_parameters[3], np.sum(results, 1)) * np.power(1 - init_parameters[3], M - np.sum(results, 1)) * init_parameters[1]
    u1 = u1_num / den
    u2 = u2_num / den
    u3 = 1 - u1 - u2
```

```
        init_parameters[0] = sum(u1) / N
        init_parameters[1] = sum(u2) / N
        init_parameters[2] = sum(u1 * np.sum(results, 1)) / sum(u1 * M)
        init_parameters[3] = sum(u2 * np.sum(results, 1)) / sum(u2 * M)
        init_parameters[4] = sum(u3 * np.sum(results, 1)) / sum(u3 * M)
        process_parameters.append(init_parameters)
x = np.linspace(0, epochs, epochs + 1)
process_parameters = np.array(process_parameters).T
print(process_parameters)
```