Python与项目反应理论：基于EM和MCMC的参数估计算法


代霸天

项目反应理论的开端

早在上世纪初，智力测验的发明者比奈（也可能是西蒙）便发现了一条神奇的曲线，这条曲线的x轴是智力水平，y轴是试题正确率，而这是项目反应理论（以下简称IRT）的最初雏形。上世界五六十年代，ETS的统计学家Lord经过一系列的工作，正式开创了IRT理论。

为什么在经典测验理论（以下简称CTT）存在的情况下，还要继续IRT理论呢？因为CTT的统计模型毛病超级多。首先，CTT的核心概念——信度，无法计算，这是最致命的；其次，CTT统计模型过于简单，像猜测度、失误、时间等等参数都很难纳入模型。所以抛弃了信度概念的IRT应运而生。同时IRT可以解决CTT完全应付不了的问题，比如组卷，IRT可以很好的控制误差和等值，CTT想正儿八经的控制误差和等值难于登天，比如理想点反应机制，IRT有理想点模型处理，CTT无法处理，比如排序数据（CTT假设真分数和误差独立，但在排序数据中，真分数和误差明显不是独立的，而IRT的瑟斯顿模型可以解决这个问题）。

IRT的函数

对于最初的IRT来说，有三条基本假设，一是单维性，二是局部独立性，三是项目反应函数假设，但其实前两条假设可有可无（例如mirt理论打破了单维性假设，题组反应理论打破了局部独立性假设），核心的是第三个假设。

对于单维度IRT来说，最常用的两个函数，一个是正态肩形曲线函数，一个是logistic函数（跟茴香豆的N种叫法一样，你也可以叫它sigmoid函数）。我们这里主要说logistic函数。logisitc的推导没什么好说的，无非是假设试题作答正确概率为 P ， 则答错的概率为 $Q=1-P$ ，取自然对数比，则 $log\left[\frac{P}{Q}\right]=log(P)-log(Q)=z$ ，而 z 是潜在特质 θ 的线性函数 $z=a\theta+b$ ，于是我们可以得到函数 $P=\frac{e^{a\theta+b}}{1+e^{a\theta+b}}$ （这个推导其实是搞笑的，正儿八经的推导是基于正态肩曲线的），其中 a 可以称为区分度（或斜率）， b 可以称为阈值（或截距），而 $-\frac{b}{a}$ 则是难度（或通俗度）。注意，有时候这个logistic函数上会有个特殊参数D，例如 $P=\frac{e^{D(a\theta+b)}}{1+e^{D(a\theta+b)}}$ ，D通常等于1.702，这是为了让logistic函数更接近正态肩形曲线，可以证明 $|\Phi(z)-\Psi(1.702z)|<0.01$ ，但是加D不加D意义不大。

上面的项目反应函数中，如果区分度 a 恒等于1，那么这个函数也称为Rasch函数，是由丹麦家Rasch独立提出的一种统计模型，其中 θ 呈现随机效应，而 b 呈现固定效应，学过线性模型的人能看出这是一个混合模型。

双参数二级计分模型的参数估计

我们先对较为简单的双参数二级计分模型进行参数估计，模型即为之前的 $P = \frac{e^{a\theta+b}}{1+e^{a\theta+b}}$ 。

先定义一个基础类

```
from __future__ import print_function, division
import numpy as np
import warnings

class BaseIrt(object):

    def __init__(self, scores=None):
        self.scores = scores

    @staticmethod
    def p(z):
        # 回答正确的概率函数
        e = np.exp(z)
        p = e / (1.0 + e)
        return p
```

然后定义一个Irt2PL继承这个类，并加上了计算z函数的静态方法，对于z函数的值，我们加了一些限制，这是为了防止数值溢出。

```
class Irt2PL(BaseIrt):
    ''
    @staticmethod
    def z(slop, threshold, theta):
        # z函数
        _z = slop * theta + threshold
        _z[_z > 35] = 35
        _z[_z < -35] = -35
        return _z
```

然法求解 a 和 b 的值 。由此，Irt2PL类的构造函数如下

```
class Irt2PL(BaseIrt):
    # EM算法求解
    def __init__(self, init_slop=None, init_threshold=None, max_iter=10000, tol=1e-5,
                 m_step_method='newton', *args, **kwargs):
        """
        :param init_slop: 斜率初值
        :param init_threshold: 阈值初值
        :param max_iter: EM算法最大迭代次数
        :param tol: 精度
        :param gp_size: Gauss-Hermite积分点数
        """
        super(Irt2PL, self).__init__(*args, **kwargs)
        # 斜率初值
        if init_slop is not None:
            self._init_slop = init_slop
        else:
            self._init_slop = np.ones(self.scores.shape[1])
        # 阈值初值
        if init_threshold is not None:
            self._init_threshold = init_threshold
        else:
            self._init_threshold = np.zeros(self.scores.shape[1])
        self._max_iter = max_iter
        self._tol = tol
        self._m_step_method = '_{0}'.format(m_step_method)
        self.x_nodes, self.x_weights = self.get_gh_point(gp_size)
```

E步的求解

从直觉的角度，依据贝叶斯法则， $P(\theta_i)=\frac{L(\theta_i)g(\theta_i)}{\int_\theta L(\theta_i)g(\theta_i)}$ ，所以 θ 下样本量分布（人数分布）为 $\sum_u \frac{L(\theta_i|u_j)g(\theta_i)}{\int_\theta L(\theta_i)g(\theta_i)}$ ，其中 $g(\theta_i)$ 是概率密度函数， $L(\theta_i|u_j)$ 是试题作答模式 u_j 下的似然函数，而 θ 下答对试题的样本量分布（人数分布）为 $\sum_u \frac{u_{kj}L(\theta_i|u_j)g(\theta_i)}{\int_\theta L(\theta_i)g(\theta_i)}$ ， u_{kj} 代表的是第 j 个作答模式下第 k 道题的答题情况。很明显，前面的公式为连续变量，很难求解，需要用数值积分的方法，考虑到 $g(\theta_i)$ 通常假设为正态分布的概率密度函数，所以我们可以用最简单的Gauss–Hermite积分求解（当然，最好的方法是自适应积分）。

Gauss–Hermite积分形式如下

$$\int e^{-x^2} f(x)dx \approx \sum w_i f(x_i)$$

我们假设 $\theta \sim N(0,1)$ ，于是我们的Gauss–Hermite积分形式为 $\sum \frac{w_i}{\sqrt{\pi}} f(\sqrt{2}x_i)$ ，我们在Irt2PL类上添加一个静态方法，处理Gauss–Hermite积分

```
class Irt2PL(BaseIrt):
    ''
    @staticmethod
    def get_gh_point(gp_size):
        x_nodes, x_weights = np.polynomial.hermite.hermgauss(gp_size)
        x_nodes = x_nodes * 2 ** 0.5
        x_nodes.shape = x_nodes.shape[0], 1
        x_weights = x_weights / np.pi ** 0.5
        x_weights.shape = x_weights.shape[0], 1
        return x_nodes, x_weights
```

似然函数和均值计算

双参数IRT似然函数和E步的代码具体如下

```
class BaseIrt(object):
    ''
    def _lik(self, p_val):
        # 似然函数
        scores = self.scores
        loglik_val = np.dot(np.log(p_val + 1e-200), scores.transpose()) + \
                     np.dot(np.log(1 - p_val + 1e-200), (1 - scores).transpose())
        return np.exp(loglik_val)

    def _e_step(self, p_val, weights):
        # EM算法E步
        # 计算theta的分布人数
        scores = self.scores
        lik_wt = self._lik(p_val) * weights
        # 归一化
        lik_wt_sum = np.sum(lik_wt, axis=0)
        _temp = lik_wt / lik_wt_sum
        # theta的人数分布
```

```
        right_dis = np.dot(_temp, scores)
        full_dis.shape = full_dis.shape[0], 1
        # 对数似然值
        print(np.sum(np.log(lik_wt_sum)))
        return full_dis, right_dis
```

上面的似然函数代码部分用了一点小花招，比如用对数将乘法变成加法以及1e-200这个极小数，这些都是为了避免数值溢出。

M步的求解

M步的求解算法很多，我们这里主要写两种，收敛速度很快的牛顿迭代以及以稳健见长的迭代加权最小二乘法（irls）。

迭代加权最小二乘法（irls）

irls是一种收敛速度很快，也很稳健，同时易于实现编程的非线性方程求解算法，代码如下

```
class Irt2PL(BaseIrt):
    ''
    def _irls(self, p_val, full_dis, right_dis, slop, threshold, theta):
        # 所有题目误差列表
        e_list = (right_dis - full_dis * p_val) / full_dis * (p_val * (1 - p_val))
        # 所有题目权重列表
        _w_list = full_dis * p_val * (1 - p_val)
        # z函数列表
        z_list = self.z(slop, threshold, theta)
        # 加上了阈值哑变量的数据
        x_list = np.vstack((threshold, slop))
        # 精度
        delta_list = np.zeros((len(slop), 2))
        for i in range(len(slop)):
            e = e_list[:, i]
            _w = _w_list[:, i]
            w = np.diag(_w ** 0.5)
            wa = np.dot(w, np.hstack((np.ones((self.x_nodes.shape[0], 1)), theta)))
            temp1 = np.dot(wa.transpose(), w)
            temp2 = np.linalg.inv(np.dot(wa.transpose(), wa))
            x0_temp = np.dot(np.dot(temp2, temp1), (z_list[:, i] + e))
            delta_list[i] = x_list[:, i] - x0_temp
            slop[i], threshold[i] = x0_temp[1], x0_temp[0]
        return slop, threshold, delta_list
```

牛顿迭代

牛顿迭代也是一种收敛速度很快的算法，但缺点是必须要计算步长，否则可能会不收敛，我们假设不需要计算步长，事实上步长恒为1的收敛效果还不错。代码如下

```
class Irt2PL(BaseIrt):
    ''
    def _newton(self, p_val, full_dis, right_dis, slop, threshold, theta):
        # 一阶导数
        dp = right_dis - full_dis * p_val
        # 二阶导数
        ddp = full_dis * p_val * (1 - p_val)
        # jac矩阵和hess矩阵
        jac1 = np.sum(dp, axis=0)
        jac2 = np.sum(dp * theta, axis=0)
        hess11 = -1 * np.sum(ddp, axis=0)
        hess12 = hess21 = -1 * np.sum(ddp * theta, axis=0)
        hess22 = -1 * np.sum(ddp * theta ** 2, axis=0)
        delta_list = np.zeros((len(slop), 2))
        # 把求稀疏矩阵的逆转化成求每个题目的小矩阵的逆
        for i in range(len(slop)):
            jac = np.array([jac1[i], jac2[i]])
            hess = np.array(
                [[hess11[i], hess12[i]],
                 [hess21[i], hess22[i]]]
            )
            delta = np.linalg.solve(hess, jac)
            slop[i], threshold[i] = slop[i] - delta[1], threshold[i] - delta[0]
            delta_list[i] = delta
        return slop, threshold, delta_list
```

上面的牛顿迭代，并没有计算所有参数形成的雅克比矩阵和黑塞矩阵，因为求稀疏矩阵的逆近乎于求每个小矩阵的逆，事实也是如此，参数估计效果一致。

上面还可以看到，无论是irls还是牛顿迭代，都是只迭代一次就返回值，这是EM算法的一种取巧，减少计算量，同时取得精确结果，当然，迭代多次收敛后返回值肯定更好（不确定）。

接下来就是收工的工作

```
class Irt2PL(BaseIrt):
    ''
```

```
        return self._m_step(p_val, full_dis, right_dis, slop, threshold, theta)

    def _m_step(self, p_val, full_dis, right_dis, slop, threshold, theta):
        # EM算法M步
        m_step_method = getattr(self, self._m_step_method)
        return m_step_method(p_val, full_dis, right_dis, slop, threshold, theta)

    def em(self):
        max_iter = self._max_iter
        tol = self._tol
        slop = self._init_slop
        threshold = self._init_threshold
        for i in range(max_iter):
            z = self.z(slop, threshold, self.x_nodes)
            p_val = self.p(z)
            slop, threshold, delta_list = self._est_item_parameter(slop, threshold, se
            if np.max(np.abs(delta_list)) < tol:
                print(i)
                return slop, threshold
        warnings.warn("no convergence")
        return slop, threshold
```

Irt2PL的测试

我们测试一下Irt2PL的结果，测试数据是LSAT，测试代码为

```
f = file('lsat.csv')
score = np.loadtxt(f, delimiter=",")
res = Irt2PL(scores=score, m_step_method='newton').em()
print(res)
```

结果如下，前面是 a 值，后面是b值

```
array([ 0.8256459 ,  0.72277713,  0.89078346,  0.68838961,  0.65689704]), array([ 2.77
```

这个结果与R包ltm和R包mirt的结果一致

特质参数（ θ ）估计

posteriori）算法是唯一不需要迭代的算法，所以它的计算速度是最快的，常用于在线测验的估计，其理论依据是贝叶斯法则。EAP的公式为 $E(\theta_i)=\theta_i=\frac{\int \theta_i g(\theta)L(\theta_i)d\theta}{\int g(\theta)L(\theta_i)d\theta}$ ， $g(\theta)$ 是概率密度函数，常假设服从正态分布，所以上式的积分可以用最简单的Gauss–Hermite积分处理，代码如下

```
class EAPIrt2PLModel(object):

    def __init__(self, score, slop, threshold, model=Irt2PL):
        self.x_nodes, self.x_weights = model.get_gh_point(21)
        z = model.z(slop, threshold, self.x_nodes)
        p = model.p(z)
        self.lik_values = np.prod(p**score*(1.0 - p)**(1-score), axis=1)

    @property
    def g(self):
        x = self.x_nodes[:, 0]
        weight = self.x_weights[:, 0]
        return np.sum(x * weight * self.lik_values)

    @property
    def h(self):
        weight = self.x_weights[:, 0]
        return np.sum(weight * self.lik_values)

    @property
    def res(self):
        return round(self.g / self.h, 3)
```

测试我们的代码

```
# 模拟参数
a = np.random.uniform(1, 3, 1000)
b = np.random.normal(0, 1, size=1000)
z = Irt2PL.z(a, b, 1)
p = Irt2PL.p(z)
score = np.random.binomial(1, p, 1000)
# 计算并打印潜在特质估计值
eap = EAPIrt2PLModel(score, a, b)
print(eap.res)
```

IRT抛弃了CTT测验的信度概念，是因为IRT有了更好的信息函数。从直觉的角度，同一测验，同的特质理当拥有不同的误差，CTT无法做到衡量每一个特质的误差，IRT的信息函数可以。IRT的信息函数表达式为 $\sum \frac{(P_i'(\theta))^2}{P_i(\theta)Q_i(\theta)}$ ，这个公式可由似然函数的二阶导数取均值和相反数导出来。

GRM（等级反应模型）

双参数IRT模型只是GRM（grade response theory）模型的二级计分特殊形式，算法和流程与二级计分大致一样。统计模型用的是ordered logistic（也称为proportional odds）。模型形式为 $P_k(\theta) = P_{k-1}^*(\theta) - P_k^*(\theta)$ ，其中 $P_k^*(\theta)$ 是logistic函数。

等级得分的数据

```
[4, 4, 3, 4, 2, 1]
```

通常会转化如下形式处理为

```
[[1, 1, 0, 1, 0, 0],
[0, 0, 1, 0, 0, 0],
[0, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 0, 1]]
```

本质上还是01计分问题

由于用的还是EM算法，且计算过程和二级计分大同小异，就不详加解释了，原理还是计算每个 θ 的分布，以及在各个选项上的样本分布，然后求项目参数的极大

```
class Grm(object):

    def __init__(self, scores=None, init_slop=None, init_threshold=None, max_iter=1000
        # 试题最大反应计算
        max_score = int(np.max(scores))
        min_score = int(np.min(scores))
        self._rep_len = max_score - min_score + 1
        self.scores = {}
        for i in range(scores.shape[1]):
            temp_scores = np.zeros((scores.shape[0], self._rep_len))
            for j in range(self._rep_len):
                temp_scores[:, j][scores[:, i] == min_score + j] = 1
```

```
        self.item_size = scores.shape[1]
        if init_slop is not None:
            self._init_slop = init_slop
        else:
            self._init_slop = np.ones(scores.shape[1])
        if init_threshold is not None:
            self._init_thresholds = init_threshold
        else:
            self._init_thresholds = np.zeros((scores.shape[1], self._rep_len - 1))
            for i in range(scores.shape[1]):
                self._init_thresholds[i] = np.arange(self._rep_len / 2 - 1, -self._rep
        self._max_iter = max_iter
        self._tol = tol
        self.x_nodes, self.x_weights = self.get_gh_point(gp_size)

    @staticmethod
    def get_gh_point(gp_size):
        x_nodes, x_weights = np.polynomial.hermite.hermgauss(gp_size)
        x_nodes = x_nodes * 2 ** 0.5
        x_nodes.shape = x_nodes.shape[0], 1
        x_weights = x_weights / np.pi ** 0.5
        x_weights.shape = x_weights.shape[0], 1
        return x_nodes, x_weights

    @staticmethod
    def p(z):
        # 回答为某一反应的概率函数
        p_val_dt = {}
        for key in z.keys():
            e = np.exp(z[key])
            p = e / (1.0 + e)
            p_val_dt[key] = p
        return p_val_dt

    @staticmethod
    def z(slop, thresholds, theta):
        # z函数
        z_val = {}
        temp = slop * theta
        for i, threshold in enumerate(thresholds):
            z_val[i] = temp[:, i][:, np.newaxis] + threshold
        return z_val
```

```
        rep_len = self._rep_len
        scores = self.scores
        for i in range(self.item_size):
            for j in range(rep_len):
                p_pre = 1 if j == 0 else p_val_dt[i][:, j - 1]
                p = 0 if j == rep_len - 1 else p_val_dt[i][:, j]
                loglik_val += np.dot(np.log(p_pre - p + 1e-200)[:, np.newaxis], scores
        return np.exp(loglik_val)

    def _e_step(self, p_val_dt, weights):
        # E步计算theta的分布人数
        scores = self.scores
        lik_wt = self._lik(p_val_dt) * weights
        # 归一化
        lik_wt_sum = np.sum(lik_wt, axis=0)
        _temp = lik_wt / lik_wt_sum
        # theta的人数分布
        full_dis = np.sum(_temp, axis=1)
        # theta下回答的人数分布
        right_dis_dt = {}
        for i in range(self.item_size):
            right_dis_dt[i] = np.dot(_temp, scores[i])
        # full_dis.shape = full_dis.shape[0], 1
        # 对数似然值
        print(np.sum(np.log(lik_wt_sum)))
        return full_dis, right_dis_dt

    def _pq(self, p_val):
        return p_val * (1 - p_val)

    @staticmethod
    def _item_jac(p_val, pq_val, right_dis, len_threshold, rep_len, theta):
        # 雅克比矩阵
        dloglik_val = np.zeros(len_threshold + 1)
        _theta = theta[:, 0]
        for i in range(rep_len):
            p_pre, pq_pre = (1, 0) if i == 0 else (p_val[:, i - 1], pq_val[:, i - 1])
            p, pq = (0, 0) if i == rep_len - 1 else (p_val[:, i], pq_val[:, i])
            temp1 = _theta * right_dis[:, i] * (1 - p_pre - p)
            dloglik_val[-1] += np.sum(temp1)
            if i < rep_len - 1:
                temp2 = right_dis[:, i] * pq / (p - p_pre + 1e-200)
                dloglik_val[i] += np.sum(temp2)
```

```
                dloglik_val[i - 1] += np.sum(temp3)
        return dloglik_val

    @staticmethod
    def _item_hess(p_val, pq_val, full_dis, len_threshold, rep_len, theta):
        # 黑塞矩阵
        ddloglik_val = np.zeros((len_threshold + 1, len_threshold + 1))
        _theta = theta[:, 0]
        for i in range(rep_len):
            p_pre, dp_pre = (1, 0) if i == 0 else (p_val[:, i - 1], pq_val[:, i - 1])
            p, dp = (0, 0) if i == rep_len - 1 else (p_val[:, i], pq_val[:, i])
            if i < rep_len - 1:
                temp1 = full_dis * _theta * dp * (dp_pre - dp) / (p_pre - p + 1e-200)
                ddloglik_val[len_threshold:, i] += np.sum(temp1)
                temp2 = full_dis * dp ** 2 / (p_pre - p + 1e-200)
                ddloglik_val[i, i] += -np.sum(temp2)
            if i > 0:
                temp3 = full_dis * _theta * dp_pre * (dp - dp_pre) / (p_pre - p + 1e-2
                ddloglik_val[len_threshold:, i - 1] += np.sum(temp3, axis=0)
                temp4 = full_dis * dp_pre ** 2 / (p_pre - p + 1e-200)
                ddloglik_val[i - 1, i - 1] += -np.sum(temp4)
            if 0 < i < rep_len - 1:
                ddloglik_val[i, i - 1] = np.sum(full_dis * dp * dp_pre / (p_pre - p +
            temp5 = full_dis * _theta ** 2 * (dp_pre - dp) ** 2 / (p - p_pre)
            ddloglik_val[-1, -1] += np.sum(temp5, axis=0)
        ddloglik_val += ddloglik_val.transpose() - np.diag(ddloglik_val.diagonal())
        return ddloglik_val

    def _m_step(self, p_val_dt, full_dis, right_dis_dt, slop, thresholds, theta):
        # M步，牛顿迭代
        rep_len = self._rep_len
        len_threshold = thresholds.shape[1]
        delta_list = np.zeros((self.item_size, len_threshold + 1))
        for i in range(self.item_size):
            p_val = p_val_dt[i]
            pq_val = self._pq(p_val)
            right_dis = right_dis_dt[i]
            jac = self._item_jac(p_val, pq_val, right_dis, len_threshold, rep_len, the
            hess = self._item_hess(p_val, pq_val, full_dis, len_threshold, rep_len, th
            delta = np.linalg.solve(hess, jac)
            slop[i], thresholds[i] = slop[i] - delta[-1], thresholds[i] - delta[:-1]
            delta_list[i] = delta
        return slop, thresholds, delta_list
```

```
            full_dis, right_dis_dt = self._e_step(p_val, self.x_weights)
            return self._m_step(p_val, full_dis, right_dis_dt, slop, threshold, theta)

        def em(self):
            max_iter = self._max_iter
            tol = self._tol
            slop = self._init_slop
            thresholds = self._init_thresholds
            for i in range(max_iter):
                z = self.z(slop, thresholds, self.x_nodes)
                p_val = self.p(z)
                slop, thresholds, delta_list = self._est_item_parameter(slop, thresholds,
                if np.max(np.abs(delta_list)) < tol:
                    print(i)
                    return slop, thresholds
            warnings.warn("no convergence")
            return slop, thresholds
```

我们验证一下上面的代码，数据来源是R包ltm和R包mirt的Science数据

```
scores = np.loadtxt('science.csv', delimiter=',')
grm = Grm(scores=scores)
print(grm.em())
```

打印出来的结果如下，第一个array是区分度，第二array是难度，由于是4级计分，所以每道题目有3个阈值

```
(array([ 1.04263937,  1.22484017,  2.27436428,  1.09570092]),
array([[ 4.86705787,  2.64203124, -1.46577716],
       [ 2.92549801,  0.90209949, -2.26544434],
       [ 5.22055883,  2.20725305, -1.95463406],
       [ 3.35070545,  0.99301087, -1.68806261]]))
```

以上结果与R包ltm和R包mirt结果一致

MCMC

最后我们从贝叶斯的角度来求解IRT参数，即MCMC算法。MCMC算法优点是实现简单，容易编程，对初值不敏感，可以同时估计项目参数和潜在变量，缺点是耗时。我们这次对三参数IRT模型

进行参数估计，三参数IRT模型公式为 $c+(1-c)\frac{e^{a\theta+b}}{1+e^{a\theta+b}}$ ，与双参数模型相比，三参数多了个 c 参数，这个 c 参数通常称为猜测参数。我们采用的MCMC算法是最简单的gibbs抽样。

依赖的库

除了numpy外，由于mcmc很耗时，我们还需要引入progressbar2这个库

```
from __future__ import print_function, division
import numpy as np
import progressbar
```

参数分布

我们假设 $\theta \sim N(0,1)$ ， $a \sim lognormal(0,1)$ （对数正态分布）， $b \sim N(0,1)$ ， $c \sim beta(5,17)$

则上述对数概率密度函数的python代码为

```
def _log_lognormal(param):
    # 对数正态分布的概率密度分布的对数
    return np.log(1.0 / param) + _log_normal(np.log(param))


def _log_normal(param):
    # 正态分布的概率密度分布的对数
    return param ** 2 * -0.5


def _param_den(slop, threshold, guess):
    # 项目参数联合概率密度
    return _log_normal(threshold) + _log_lognormal(slop) + 4 * np.log(guess) + 16 * np
```

对数似然函数

三参数对数似然函数与双参数几乎一模一样

```
def logistic(slop, threshold, guess, theta):
    # logistic函数
    return guess + (1 - guess) / (1.0 + np.exp(-1 * (slop * theta + threshold)))
```

```
def loglik(slop, threshold, guess, theta, scores, axis=1):
    # 对数似然函数
    p = logistic(slop, threshold, guess, theta)
    p[p <= 0] = 1e-10
    p[p >= 1] = 1 - 1e-10
    return np.sum(scores * np.log(p) + (1 - scores) * np.log(1 - p), axis=axis)
```

转移函数

有了对数似然函数和对数概率密度函数，我们就能构造转移函数

```
def _tran_theta(slop, threshold, guess, theta, next_theta, scores):
    # 特质的转移函数
    pi = (loglik(slop, threshold, guess, next_theta, scores) + _log_normal(next_theta)
        loglik(slop, threshold, guess, theta, scores) + _log_normal(theta)[:, 0])
    pi = np.exp(pi)
    # 下步可省略
    pi[pi > 1] = 1
    return pi

def _tran_item_para(slop, threshold, guess, next_slop, next_threshold, next_guess, the
    # 项目参数的转移函数
    nxt = loglik(next_slop, next_threshold, next_guess, theta, scores, 0) + _param_den
    now = loglik(slop, threshold, guess, theta, scores, 0) + _param_den(slop, threshol
    pi = nxt - now
    pi.shape = pi.shape[1]
    pi = np.exp(pi)
    # 下步可省略
    pi[pi > 1] = 1
    return pi
```

抽样

我们定的抽样分布是 $\theta_t \sim N(\theta_{t-1}, 1)$ ， $a_t \sim N(a_{t-1}, 0.3)$ ， $b_t \sim N(b_{t-1}, 0.3)$ ， $c_t \sim unif(c_{t-1}, 0.03)$ （均匀分布），整个MCMC抽样的具体代码如下

```
def mcmc(chain_size, scores):
    # 样本量
    person_size = scores.shape[0]
    # 项目量
```

```
    theta = np.zeros((person_size, 1))
    # 斜率初值
    slop = np.ones((1, item_size))
    # 阈值初值
    threshold = np.zeros((1, item_size))
    # 猜测参数初值
    guess = np.zeros((1, item_size)) + 0.1
    # 参数储存记录
    theta_list = np.zeros((chain_size, len(theta)))
    slop_list = np.zeros((chain_size, item_size))
    threshold_list = np.zeros((chain_size, item_size))
    guess_list = np.zeros((chain_size, item_size))
    bar = progressbar.ProgressBar()
    for i in bar(range(chain_size)):
        next_theta = np.random.normal(theta, 1)
        theta_pi = _tran_theta(slop, threshold, guess, theta, next_theta, scores)
        theta_r = np.random.uniform(0, 1, len(theta))
        theta[theta_r <= theta_pi] = next_theta[theta_r <= theta_pi]
        theta_list[i] = theta[:, 0]
        next_slop = np.random.normal(slop, 0.3)
        # 防止数值溢出
        next_slop[next_slop < 0] = 1e-10
        next_threshold = np.random.normal(threshold, 0.3)
        next_guess = np.random.uniform(guess - 0.03, guess + 0.03)
        # 防止数值溢出
        next_guess[next_guess <= 0] = 1e-10
        next_guess[next_guess >= 1] = 1 - 1e-10
        param_pi = _tran_item_para(slop, threshold, guess, next_slop, next_threshold,
        param_r = np.random.uniform(0, 1, item_size)
        slop[0][param_r <= param_pi] = next_slop[0][param_r <= param_pi]
        threshold[0][param_r <= param_pi] = next_threshold[0][param_r <= param_pi]
        guess[0][param_r <= param_pi] = next_guess[0][param_r <= param_pi]
        slop_list[i] = slop[0]
        threshold_list[i] = threshold[0]
        guess_list[i] = guess[0]
    return theta_list, slop_list, threshold_list, guess_list
```

我们测试一下代码，只用一个链，长度7000，燃烧最初的3000次转移

```
# 样本量和题量
PERSON_SIZE = 1000
ITEM_SIZE = 60
```

```
a = np.random.lognormal(0, 1, (1, ITEM_SIZE))
a[a > 4] = 4
b = np.random.normal(0, 1, (1, ITEM_SIZE))
b[b > 4] = 4
b[b < -4] = -4
c = np.random.beta(5, 17, (1, ITEM_SIZE))
c[c < 0] = 0
c[c > 0.2] = 0.2
true_theta = np.random.normal(0, 1, (PERSON_SIZE, 1))
p_val = logistic(a, b, c, true_theta)
scores = np.random.binomial(1, p_val)
# MCMC参数估计
thetas, slops, thresholds, guesses = mcmc(7000, scores=scores)
est_theta = np.mean(thetas[3000:], axis=0)
est_slop = np.mean(slops[3000:], axis=0)
est_threshold = np.mean(thresholds[3000:], axis=0)
est_guess = np.mean(guesses[3000:], axis=0)
# 打印误差
print(np.mean(np.abs(est_slop - a[0])))
print(np.mean(np.abs(est_threshold - b[0])))
print(np.mean(np.abs(est_guess - c[:, 0])))
print(np.mean(np.abs(est_theta - true_theta[:, 0])))
```

结果如下，还不赖

```
0.152234956351
0.179545516817
0.07689038666
0.24119026894
```

总结

IRT可以说是心理测量界的一次革命，也正是因为IRT理论的存在，SAT、ACT、雅思、托业等考试才能做到一年多次考试（其中的玄机在于IRT等值和基于IRT的自适应测验），同时，运用IRT的非认知测验（例如人格等），也在处理自比数据和抵抗作假等方面成果卓越。本文介绍的是最简单的IRT模型，IRT模型成千上万，下一章将会介绍多维IRT模型，也即全息项目因子分析。

编辑于 2018-04-17

心理统计

文章被以下专栏收录

心理测量与自适应学习
统计和机器学习在心理和教育的应用

关注专栏

推荐阅读

用Python编写结构方程模型参数估计程序（上篇）

结构方程模型是一种很low的心理测量方法，唯一的例外是它的测量模型部分——验证性因子分析，因为验证性因子分析本质上可以从CTT和IRT推导出来（无论是CTT和IRT，均有严格的公理假设），而...

代霸天

处理潜变量或随机效应的高维积分时，我们做些什么之MCEM

本文面向对象是心理和教育等社科统计的初入门者。 广义线性潜变量模型（Generalized Linear Latent Variable Models，在心理学和教育学常用的是连接函数为probit或logit的项目反应模型或因...

代霸天

Python与经典测量理论

什么是经典测量理论？经典测量理论（Classical Test Theory，简CTT）发端于100年前， 其优点计算简单不烧脑（不算结构方程一套），缺点是理论假设存在重bug（后面会讲），所以上世纪...

代霸天

28 条评论

切换为时间排序

写下你的评论...

Lucky 2017-12-09

我想用MULTILOG计算，1000个考生在20道多项计分题上面的反应情况，可是出现的output我看不太懂，网上也没有什么资料可以寻找，最主要的是从高中，本科，我都是文科生……求帮忙……

赞

代霸天 (作者) 回复 Lucky 2017-12-28

没用过这个软件，爱莫能助

赞

赞

展开其他 1 条回复

哇咔咔 2018-01-15

想问一下，源码有上传吗?

1

HuangXiao 2018-02-07

笔者您其他系列文章还会更新吗?

赞

代霸天 (作者) 回复 HuangXiao 2018-02-07

工作繁忙，慢更

赞

代霸天 (作者) 回复 HuangXiao 2018-02-07

工作繁忙，慢更

赞

小橙子 2018-02-24

你好，gp_size会对结果有啥影响啊。

赞

代霸天 (作者) 回复 小橙子 2018-02-24

高斯厄米特积分点的数量，影响积分计算的精度

1

小橙子 回复 代霸天 (作者) 2018-02-24

这个是越大越好是吗，数量小了有时候跑不出来结果。

赞

展开其他 2 条回复

流光江影 2018-04-08

请问您是在哪个公司工作呢?

赞

已赞同 58 ▼ 28 条评论 分享 喜欢 收藏 申请转载 ···

赞

oiwuliang 2018-06-06

请问用的是什么python包（需要python install什么吧）？数据来源是R包ltm和R包mirt的Science数据，在哪？

赞

代霸天 (作者) 回复 oiwuliang 2018-06-06

自己写的参数估计程序，只引用了线性代数库numy，数据在ltm和mirt库里面

赞

aliciayang 2019-01-17

Science数据维度是怎样的，这个R包数据没有找到，正在尝试用其他的题目分数数据来代替进行实验

赞

zarelone Lee 2019-04-06

有一个小疑问：
BaseIRT里面计算的是p，但是后面IRT2PL里面的都是p_val，所以就想问问各中原因是什么呢？

赞

代霸天 (作者) 回复 zarelone Lee 2019-04-09

p_val代表p的值

赞

小白 2019-04-26

作者您好，您代码与我是用R语言mirt包跑出的结果不一致，我是用coef(model, simplify=TRUE, IRTpars = TRUE) 获取的项目参数如下：

a b1 b2 b3
Comfort 1.042 -4.669 -2.534 1.407
Work 1.226 -2.385 -0.735 1.849
Future 2.293 -2.282 -0.965 0.856
Benefit 1.095 -3.058 -0.906 1.542

如果是我R语言用错了，请您指导一下，谢谢🙏

赞

因子旋转算法不一样吧

赞

zarelone Lee 回复 小白 2019-05-24

小白too来掺和一下，作者的写代码时的b应该可以说是易度参数（加个负号就是难度参数），然后总体上看参数还是比较一致的

赞

shaun 2019-04-28

我关注您的时候我看您写的是酒店经理来着哈哈哈，今天来重温一下代码，突然发现换成体育老师了，有风格！

赞

张学奶 2019-06-13

太厉害了，感谢，学习到了

赞

林献寒 06-03

有没有多级多维度的irt代码呀？

赞

神林 07-11

您好，方便提供一下您程序中的原始数据吗。谢谢。

赞

山川 08-31

作者您好，有python求三参数估计的源码吗？[拜托][拜托]有偿求源代码

赞