第3章 逻辑回归方法

3.1 基本概念

3.1.1 回归概述

回归指研究一组随机变量($Y_1, Y_2, \cdots, Y_i$)和另一组($X_1, X_2, \cdots, X_k$)变量之间关系的统计分析方法,又称多重回归分析。通常前者是因变量,后者是自变量。当因变量和自变量为线性关系时,它是一种特殊的线性模型。最简单的情形是一元线性回归,由大体上有线性关系的一个自变量和一个因变量组成;模型是 $Y=a+bX+\varepsilon$(X 是自变量,Y 是因变量,ε 是随机误差)。若进一步假定随机误差遵从正态分布,就叫作正态线性模型。

一般地,若有 k 个自变量和 1 个因变量,则因变量的值分为两部分:一部分由自变量影响,即表示为它的函数,函数形式已知且含有未知参数;另一部分由其他未考虑因素和随机性影响,即随机误差。当函数为参数未知的线性函数时,称为线性回归分析模型;当函数为参数未知的非线性函数时,称为非线性回归分析模型。当自变量个数大于 1 时称为多元回归,当因变量个数大于 1 时称为多重回归。回归的主要种类有线性回归、曲线回归、二分类逻辑回归及多分类逻辑回归。

3.1.2 线性回归简介

线性回归(Linear Regression)是利用称为线性回归方程的最小平方函数对一个或多个自变量和因变量之间关系进行建模的一种回归分析。这种函数是一个或多个称为回归系数的模型参数的线性组合。回归分析中,如果只包括一个自变量和一个因变量,且二者的关系可用一条直线近似表示,则这种回归分析称为一元线性回归分析。如果回归分析中包括两个或两个以上的自变量,且因变量和自变量之间是线性关系,则称为多元线性回归分析。

【例 3.1】 下面举例说明线性回归的含义,假设有一个房屋销售的数据如表 3-1 所示。

表 3-1 房屋销售数据

面积/m^2	售价/万元	面积/m^2	售价/万元
123	250	102	220
150	320	…	…
87	160		

对于一个全新面积的房屋，应该如何预测其售价呢？可以用一条曲线去尽量准确地拟合表 3-1 中的数据，如果有新的面积输入，采用拟合的函数输出其售价。为了描述方便，下面给出几个常见的概念。

(1) 房屋销售记录表(表 3-1)：即训练数据集(Training Data Set)。房屋面积是模型的输入数据，为自变量 x。

(2) 房屋销售价格：输出数据，因变量 y。

(3) 拟合函数(又称为假设或模型)：$y=h(x)$。

(4) 训练数据的条目：一条训练数据由一对输入数据和输出数据组成，维度 n 称为特征个数(如表 3-1 维度为 1)。

线性回归假设特征和结果满足线性关系，每个特征对结果的影响强弱由特征前面的参数体现。如果每个特征变量先映射到一个函数，然后再参与线性计算，这样就可以表达特征与结果之间的非线性关系。设 $x_1,x_2,\cdots,x_n$ 表示 n 个特征，则拟合函数

$$h(x)=h_\theta(x)=\theta_0+\theta_1x_1+\theta_2x_2+\cdots+\theta_nx_n \tag{3-1}$$

令 $x_0=1$，则上式可以写成：

$$h_\theta(x)=\theta^{\mathrm{T}}X \tag{3-2}$$

其中，θ 是参数，用来调整每个特征 $x_i(1\leqslant i\leqslant n)$对结果的影响力。我们需要对 $h(x)$函数进行评估，这个评估函数又称为损失函数(Loss Function)或者错误函数(Error Function)，称为 J 函数。

$$J(\theta)=\frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)})-y^{(i)})^2 \tag{3-3}$$

$$\min_{\theta\in R^{n+1}}J_\theta$$

其中，$x^{(i)},y^{(i)}$是第 i 次训练的输入数据和输出数。把 $x^{(i)}$的估计值 $h_\theta(x^{(i)})$与实际值 $y^{(i)}$的差的平方和作为错误估计函数，系数$\frac{1}{2}$是为了求导方便。下面从概率的角度解释为什么误差函数要采用误差平方和的形式。假设预测值 $h_\theta(x)$和实际值 $y^{(i)}$有误差 $\varepsilon^{(i)}$，即

$$y^{(i)}=h_\theta(x)+\varepsilon^{(i)} \tag{3-4}$$

一般来说，误差满足平均值为 0 的正态分布，即已知 x 时，y 的条件概率为

$$p(y^{(i)}\mid x^{(i)};\theta)=\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y^{(i)}-\theta^{\mathrm{T}}x^{(i)})^2}{2\sigma^2}\right) \tag{3-5}$$

这样就估计了一条样本的结果概率，然而我们期待的是模型能够在全部样本上预测最准，即概率密度函数积最大，这个概率积成为最大似然估计，即希望在最大似然估计得到最大值的时候确定参数θ，需要对最大似然估计公式求导，θ 的最大似然估计$\hat{\theta}$ 满足：

$$J(\hat{\theta})=\lim_{\theta\in R^{n+1}}\frac{1}{2}\sum_{i=1}^{m}(y^{(j)}-\theta^{\mathrm{T}}x^{(i)})^2 \tag{3-6}$$

这个解释大概说明了为什么误差函数需要使用平方和，解释过程中做了一些符合客观规律的假设。如何调整θ 使得 $J(\theta)$取最小值有很多方法，最常用的是最小二乘法和梯度下降法，下面以梯度下降法为例介绍求解过程。

在选定线性回归模型后需要确定参数θ 。采用梯度下降法的流程如下。

(1) 对θ 赋初值，初值可以是随机值，也可以是全零向量。

(2) 改变θ的值,使得$J(\theta)$按梯度下降的方向进行减少。梯度方向由$J(\theta)$对θ的偏导数确定,由于是求极小值,因此梯度下降方向是偏导数的反方向,即

$$\theta_j := \theta_j + \alpha(y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)}$$

迭代更新的方法有两种,一种是批梯度下降,也就是对全部的训练数据求得误差后再对θ进行更新;另一种是增量梯度下降,即每扫描一步都要对θ进行更新。前一种方法能够不断收敛,后一种方法可能不断在收敛处徘徊。

如果将训练特征表示为$\boldsymbol{X}$矩阵,结果表示成$\boldsymbol{y}$向量,仍然采用线性回归模型,误差函数不变,则可采用最小二乘法求解θ,方法如下。

$$\theta = (\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y}$$

采用最小二乘法要求特征矩阵$\boldsymbol{X}$是列满秩的。

3.2 逻辑回归

3.2.1 二分类逻辑回归

逻辑回归(logistic regression)本质上是线性回归,只是在特征到结果的映射中加入了一层函数映射,即先把特征线性求和,然后使用逻辑回归函数$g(z)$作为最终假设函数预测,$g(z)$函数可以将连续值映射到0和1上。线性回归的假设函数只是$\theta^{\mathrm{T}}\boldsymbol{X}$,逻辑回归的假设函数如下。

$$\begin{aligned} h_\theta(x) &= g(\theta^{\mathrm{T}}\boldsymbol{X}) = \frac{1}{1+\mathrm{e}^{-\theta^{\mathrm{T}}X}} \\ g(z) &= \frac{1}{1+\mathrm{e}^{-z}} \end{aligned} \tag{3-7}$$

逻辑回归用来处理0/1问题,即预测结果属于0或1的二值分类问题。这里假设二值满足伯努利分布(假设满足泊松分布或指数分布都可以,会涉及下面的一般线性模型),即

$$\begin{aligned} P(y=1 \mid x;\theta) &= h_\theta(x) \\ P(y=0 \mid x;\theta) &= 1 - h_\theta(x) \end{aligned} \tag{3-8}$$

同样地,求最大似然估计并求导,得到的迭代公式结果为

$$\theta_j := \theta_j + \alpha(y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)} \tag{3-9}$$

可以看出,与线性回归类似,只是这里的$h_\theta(x^{(i)})$经过了$g(z)$映射。需要说明的是,逻辑回归的函数$g(z)$采用$g(z)=\frac{1}{1+\mathrm{e}^{-z}}$的形式有一套理论作为支撑,即一般线性模型理论。如果一个概率分布可以表示为

$$p(y;\eta) = b(y)\exp(\eta^{\mathrm{T}}T(y) - \alpha(\eta)) \tag{3-10}$$

这个概率分布称为指数分布。伯努利分布、正态分布及泊松分布等都属于指数分布。在逻辑回归时采用的是伯努利分布,伯努利分布的概率可以表示如下。

$$\begin{aligned} p(y;\phi) &= \phi^y(1-\phi)^{1-y} \\ &= \exp(y\log\phi + (1-y)\log(1-\phi)) \\ &= \exp\left(\left(\log\left(\frac{\phi}{1-\phi}\right)\right)y + \log(1-\phi)\right) \end{aligned} \tag{3-11}$$

其中 $\eta=\log(\phi/(1-\phi))$，因此 $\phi=\dfrac{1}{1+e^{\eta}}$。这就解释了逻辑回归的 $g(z)$为什么要采用这种形式。

3.2.2 多分类逻辑回归

上一小节主要介绍二分类逻辑回归问题，但在实际问题中，因变量 y 不一定只有两种情况，可能有多种取值，于是就有了多分类逻辑回归模型[1]。设 y 有 c 个取值，从 0 到 $c-1$，并且 $y=0$ 是一个参照组，自变量 $x=(x_1,x_2,\cdots,x_p)$，则 y 的条件概率为

$$P(y=k\mid x)=\frac{e^{g_k(x)}}{1+\sum_{i=1}^{c-1}e^{g_j(x)}} \tag{3-12}$$

其中 $k=0,1,2,\cdots,c-1$。由此可以得出相应的逻辑回归模型：

$$g_k(x)=\ln\left[\frac{P(y=k\mid x)}{P(y=0\mid x)}\right]=\beta_{k0}+\beta_{k1}x_1+\cdots+\beta_{kp}x_p \tag{3-13}$$

显然，$g_0(x)=0$。关于参数估计的详细内容可参考文献[1]。

3.2.3 逻辑回归应用举例

【例 3.2】 用逻辑回归分析顾客是否购买人造黄油与人造黄油的可涂抹性 X_1 与保质期 X_2 的关系。并依据所得模型，判定性质为 $X_1=3$，$X_2=1$ 的人造黄油是否为顾客受欢迎的黄油。该数据如表 3-2 所示。

表 3-2 人造黄油数据表

顾　　客	可涂抹性 X_1	保质期 X_2	是否购买黄油 y
1	2	3	1
2	3	4	1
3	6	5	1
4	4	4	1
5	3	2	1
6	4	7	1
7	3	5	1
8	2	4	1
9	5	6	1
10	3	6	1
11	3	3	1
12	4	5	1
13	5	4	0
14	4	3	0
15	7	5	0
16	3	3	0
17	4	4	0
18	5	2	0
19	4	2	0

续表

顾　　客	可涂抹性 X_1	保质期 X_2	是否购买黄油 y
20	5	5	0
21	6	7	0
22	5	3	0
23	6	4	0
24	6	6	0

解：设逻辑回归模型为

$$\begin{cases} z = b_0 + b_1 x_1 + b_2 x_2 \\ p(y=1) = \dfrac{e^z}{1+e^z} \end{cases}$$

所以，似然函数为

$$L = \prod_{k=1}^{24} \left(\frac{e^{z_k}}{1+e^{z_k}} \right)^{y_k} \left(1 - \frac{e^{z_k}}{-1+e^{z_k}} \right)^{1-y_k}$$

其中，y_k，$x_1(k)$，$x_2(k)$分别表示第 k 个顾客对应的可涂抹性 X_1，保质期 X_2，是否购买人造黄油，$z_k = b_0 + b_1 x_1(k) + b_2 x_2(k)$。

求解 $\max\limits_{b_0, b_1, b_2} L$，可得 $\begin{cases} b_0 = 3.528 \\ b_1 = 1.943 \\ b_2 = 1.119 \end{cases}$

则逻辑回归模型为

$$\begin{cases} z = 3.528 - 1.943 x_1 + 1.119 x_2 \\ p(y=1) = \dfrac{e^z}{1+e^z} \end{cases}$$

该模型用于预测的混淆矩阵如表 3-3 所示。

表 3-3　已知人造黄油的混淆矩阵表

实　　际	预　　测	
	预 测 购 买	预测不购买
实际购买	10	2
实际不购买	2	10

正确率为 0.883。

当 $X_1 = 3$，$X_2 = 1$ 时，$p(y=1) = \dfrac{e^z}{1+e^z} = \dfrac{e^{3.528-1.943\times3+1.119\times1}}{1+e^{3.528-1.943\times3+1.119\times1}} = 0.7653 > 0.5$

则可认为该人造黄油是顾客喜欢的黄油。

3.2.4　逻辑回归方法的特点

逻辑回归方法有下列优点。

(1) 预测结果是介于 0 和 1 之间的概率。

(2) 可以适用于连续性和类别性自变量。

(3) 容易使用和解释。

逻辑回归的缺点如下。

(1) 对模型中自变量多重共线性较为敏感。例如,两个高度相关自变量同时放入模型,可能导致较弱的一个自变量回归符号不符合预期,符号被扭转。需要利用因子分析或者变量聚类分析等手段选择代表性的自变量,以减少候选变量之间的相关性。

(2) 预测结果呈S形,因此从log(odds)向概率转化的过程是非线性的,在两端随着log(odds)值的变化,概率变化很小,边际值太小,斜率太小,而中间概率的变化很大,很敏感。导致很多区间的变量变化对目标概率的影响没有区分度,无法确定阈值。

3.2.5 逻辑回归方法的应用

逻辑回归方法在日常生活中应用非常广泛,下面举例说明该方法的应用情况。

1. 利用逻辑回归方法进行微博用户可信度的建模

文献[2]针对微博虚假用户问题,以新浪微博为研究平台对微博用户的行为进行分析,从在线时长、发帖时间、互动程度等方面,提取用于区分用户类别的特征变量,运用逻辑回归算法,提出一个基于逻辑回归的微博用户可信度评价模型。实验结果表明,该模型能够对传统的虚假用户“僵尸粉”进行识别,对新型虚假用户有较高的识别率,可以根据置信度对用户进行大致分类,实用性较强。

2. 利用逻辑回归方法识别水军

随着诸如twitter和微博等新媒体的发展,由于网络公关与营销等原因,网络水军也出现并呈现出急剧增加的态势。造成大量的网络资源和普通用户的时间遭到侵占,同时也对舆情真实性产生了重要影响。文献[3]建立了一种基于逻辑回归算法的水军识别模型,利用累计分布函数(CDF)对新浪微博用户行为属性及账号属性进行分析和选取,将合适的属性包括好友数、粉丝数、文本相似度、URL率等作为输入参数,用以训练基于逻辑回归算法的分类模型,得到相应系数,从而完成对网络水军识别模型的构建。实验结果证明了模型的准确性和有效性。

3. 利用逻辑回归方法进行垃圾邮件过滤

垃圾邮件过滤是网络信息处理中的重要问题,基于机器学习方法的垃圾邮件过滤技术是目前的研究热点。文献[4]将垃圾邮件过滤问题转化成排序问题进行建模,提出了在线排序逻辑回归学习算法,解决了在线学习中的邮件得分偏移问题;综合应用TONE算法和重采样技术,提出参数权重更新算法,解决模型学习中在线调整模型参数时的处理速度问题,满足垃圾邮件实时过滤的要求。

3.3 逻辑回归源代码结果分析

3.3.1 线性回归

线性回归的源代码如下,相关程序如下所示。

```
package logicregression.linearRegressionAnalysis;
public class LinearRegression {
    private static int chlk(double[] a, int n, int m, double[] d) {
        int u, v;
        if ((a[0] + 1.0 == 1.0) || (a[0] < 0.0)) {
              System.out.println("失败了……");
              return (-2);
        }
        a[0] = Math.sqrt(a[0]);
        for (int j = 1; j <= n - 1; j++){
             a[j] = a[j] / a[0];
        }
        for (int i = 1; i <= n - 1; i++) {
            u = i * n + i;
            for (int j = 1; j <= i; j++) {
                v = (j - 1) * n + i;
                a[u] = a[u] - a[v] * a[v];
            }
            if ((a[u] + 1.0 == 1.0) || (a[u] < 0.0)) {
                  System.out.println("失败了!!!");
                  return (-2);
            }
            a[u] = Math.sqrt(a[u]);
            if (i != (n - 1)) {
                  for (int j = i + 1; j <= n - 1; j++) {
                      v = i * n + j;
                      for (int k = 1; k <= i; k++){
                            a[v] = a[v] - a[(k - 1) * n + i] * a[(k - 1) * n + j];
                      }
                      a[v] = a[v] / a[u];
                  }
            }
        }
        for (int j = 0; j <= m - 1; j++) {
            d[j] = d[j] / a[0];
            for (int i = 1; i <= n - 1; i++) {
                u = i * n + i;
                v = i * m + j;
                for (int k = 1; k <= i; k++)
                    d[v] = d[v] - a[(k - 1) * n + i] * d[(k - 1) * m + j];
                d[v] = d[v] / a[u];
            }
        }
        for (int j = 0; j <= m - 1; j++) {
            u = (n - 1) * m + j;
            d[u] = d[u] / a[n * n - 1];
            for (int k = n - 1; k >= 1; k--) {
                u = (k - 1) * m + j;
                for (int i = k; i <= n - 1; i++) {
                    v = (k - 1) * n + i;
                    d[u] = d[u] - a[v] * d[i * m + j];
```

```
            }
            v = (k - 1) * n + k - 1;
            d[u] = d[u] / a[v];
        }
    }
    return (2);
}
/* 多元线性分析
 * @param x[m][n]    每一列存放 m 个自变量的观察值
 * @param y[n]       存放随机变量 y 的 n 个观察值
 * @param dt[4]      dt[0]:偏差平方和 q,dt[1]:平均标准偏差 s,dt[2]:返回复相关系数 r,
                     dt[3]:返回回归平方和 r
 * @param v[]        返回 m 个自变量的偏相关系数
 * @param a[]        返回回归系数 a1,a2,…,an
 * @param m          自变量的个数
 * @param n          观察数据的组数
 */
public void sqt(double[][] x, double[] y, double[] dt, double[] v, double[] a, int m, int n) {
    int mm;
    double q, e, u, p, yy, s, r, pp;
    double[] b = new double[(m + 1) * (m + 1)];
    mm = m + 1;
    b[mm * mm - 1] = n;
    for (int j = 0; j < m - 1; j++) {
        p = 0.0;
        for (int i = 0; i <= n - 1; i++) {
            p = p + x[j][i];
        }
        b[m * mm + j] = p;
        b[j * mm + m] = p;
    }
    for (int i = 0; i < m - 1; i++) {
        for (int j = i; j <= m - 1; j++) {
            p = 0.0;
            for (int k = 0; k <= n - 1; k++) {
                p = p + x[i][k] * x[j][k];
            }
            b[j * mm + i] = p;
            b[i * mm + j] = p;
        }
    }
    a[m] = 0.0;
    for (int i = 0; i < n - 1; i++) {
        a[m] = a[m] + y[i];
    }
    for (int i = 0; i < m - 1; i++) {
        a[i] = 0.0;
        for (int j = 0; j <= n - 1; j++) {
            a[i] = a[i] + x[i][j] * y[j];
        }
    }
```

```
        chlk(b, mm, 1, a);
        yy = 0.0;
        for (int i = 0; i < n - 1; i++) {
            yy = yy + y[i] / n;
        }
        q = 0.0;
        e = 0.0;
        u = 0.0;
        for (int i = 0; i <= n - 1; i++) {
            p = a[m];
            for (int j = 0; j <= m - 1; j++)
                p = p + a[j] * x[j][i];
            q = q + (y[i] - p) * (y[i] - p);
            e = e + (y[i] - yy) * (y[i] - yy);
            u = u + (yy - p) * (yy - p);
        }
        s = Math.sqrt(q / n);
        r = Math.sqrt(1.0 - q / e);
        for (int j = 0; j <= m - 1; j++) {
            p = 0.0;
            for (int i = 0; i <= n - 1; i++) {
                pp = a[m];
                for (int k = 0; k <= m - 1; k++)
                    if (k != j)
                        pp = pp + a[k] * x[k][i];
                p = p + (y[i] - pp) * (y[i] - pp);
            }
            v[j] = Math.sqrt(1.0 - q / p);
        }
        dt[0] = q;
        dt[1] = s;
        dt[2] = r;
        dt[3] = u;
    }
    /* @param args
     */
     public static void main(String[] args) {
    /* 多元回归
     */
    double[] a = new double[4];
    double[] v = new double[3];
    double[] dt = new double[4];

    double[][] x = { { 1.1, 1.0, 1.2, 1.1, 0.9 },
                     { 2.0, 2.0, 1.8, 1.9, 2.1 },
                     { 3.2, 3.2, 3.0, 2.9, 2.9 } };
    double[] y = { 10.1, 10.2, 10.0, 10.1, 10.0 };
    LinearRegression lr = new LinearRegression();
    lr.sqt(x, y, dt, v, a, 3, 5);
    for (int i = 0; i <= 3; i++){
        System.out.println("a(" + i + ") = " + a[i]);
```

```
        }
        System.out.println("q = " + dt[0] + "  s = " + dt[1] + "  r = " + dt[2]);
        for (int i = 0; i <= 2; i++){
            System.out.println("v(" + i + ") = " + v[i]);
        }
        System.out.println("u = " + dt[3]);
    }
}
```

上述代码的运行结果如图 3-1 所示。

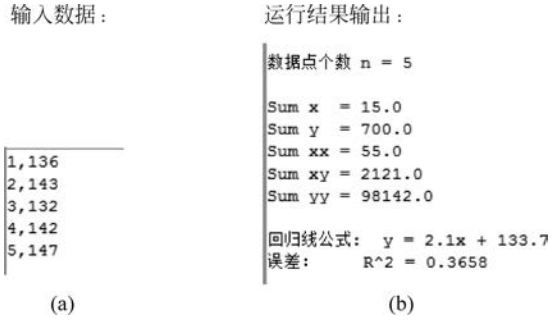

图 3-1　线性回归程序运行结果

对于 4 对自变量和因变量(前面的是自变量,后面的是因变量)通过一元线性回归可以得到当前的数据点的数目为 5,所有自变量的和为 15,所有因变量的和为 700,自变量的平方的值为 55,自变量和因变量的乘积为 2121,因变量的平方值为 98 142,统计得出了回归线的公式以及最后得到的误差值。

3.3.2　多分类逻辑回归

多分类逻辑回归源代码包括 3 个文件,即 DataPoint. java、LinearRegression. java 和 RegressionLine. java。

1. DataPoint. java

```
package logicregression.multipleLinearRegressionAnalysis;
/*定义一个 DataPoint 类,对坐标 x 和 y 点进行封装
 */
public class DataPoint {

    public float x;
    public float y;
    public DataPoint(float x,float y){
        this.x = x;
        this.y = y;
    }
    }
```

2. LinearRegression. java

```
package logicregression.multipleLinearRegressionAnalysis;
```

```
public class LinearRegression {
    private static final int MAX_POINTS = 10;
    private double E;
    public static void main(String[] args) {
        RegressionLine line = new RegressionLine();
        line.addDataPoint(new DataPoint(1, 136));
        line.addDataPoint(new DataPoint(2, 143));
        line.addDataPoint(new DataPoint(3, 132));
        line.addDataPoint(new DataPoint(4, 142));
        line.addDataPoint(new DataPoint(5, 147));
        printSums(line);
        printLine(line);
    }
    /* 打印和
     * @param line 回归线
     */
    private static void printSums(RegressionLine line) {
        System.out.println("\n数据点个数 n = " + line.getDataPointCount());
        System.out.println("\nSum x  = " + line.getSumX());
        System.out.println("Sum y  = " + line.getSumY());
        System.out.println("Sum xx = " + line.getSumXX());
        System.out.println("Sum xy = " + line.getSumXY());
        System.out.println("Sum yy = " + line.getSumYY());
    }
    /* 打印回归线功能
     * @param line    回归线
     */
    private static void printLine(RegressionLine line) {
        System.out.println("\n回归线公式:  y = " + line.getA1() + "x + "
                + line.getA0());
        System.out.println("误差:      R^2 = " + line.getR());
    }
}
```

3. RegressionLine.java

```
package logicregression.multipleLinearRegressionAnalysis;
import java.math.BigDecimal;
import java.util.ArrayList;
public class RegressionLine {
/* x的数量 */
private double sumX;
/* y的数量 */
private double sumY;
/* x的平方值 */
private double sumXX;
/* x乘以y的值 */
private double sumXY;
/* y的平方的值 */
private double sumYY;
/* sumDeltaY的平方的值 */
```

```
private double sumDeltaY2;
/* 误差 */
private double sse;
private double sst;
private double E;
private String[] xy;
private ArrayList listX;
private ArrayList listY;
private int XMin, XMax, YMin, YMax;
/* 线系数 a0 */
private float a0;
/* 线系数 a1 */
private float a1;
/* 数据点数 */
private int pn;
/* 记录系数是否有效 */
private boolean coefsValid;
//构造函数
public RegressionLine() {
    XMax = 0;
    YMax = 0;
    pn = 0;
    xy = new String[2];
    listX = new ArrayList();
    listY = new ArrayList();
}
/* 返回当前数据点的数目
 * @return 当前数据点的数目
 */
public int getDataPointCount() {
    return pn;
}
/* 返回线系数 a0
 */
public float getA0() {
    validateCoefficients();
    return a0;
}
/* 返回线系数 a1
 */
public float getA1() {
    validateCoefficients();
    return a1;
}
public double getSumX() {
    return sumX;
}
public double getSumY() {
    return sumY;
}
public double getSumXX() {
```

```
        return sumXX;
    }
    public double getSumXY() {
        return sumXY;
    }
    public double getSumYY() {
        return sumYY;
    }
    public int getXMin() {
        return XMin;
    }
    public int getXMax() {
        return XMax;
    }
    public int getYMin() {
        return YMin;
    }
    public int getYMax() {
        return YMax;
    }
    /* 计算方程系数 y = ax + b 中的 a
     */
    private void validateCoefficients() {
        if (coefsValid) {
            return;
        }
        if (pn >= 2) {
            float xBar = (float) (sumX / pn);
            float yBar = (float) (sumY / pn);
            a1 = (float) ((pn * sumXY - sumX * sumY) / (pn * sumXX - sumX * sumX));
            a0 = (float) (yBar - a1 * xBar);
        } else {
            a0 = a1 = Float.NaN;
        }
        coefsValid = true;
    }
    /* 添加一个新的数据点: 改变总量
     */
    public void addDataPoint(DataPoint dataPoint) {
        sumX += dataPoint.x;
        sumY += dataPoint.y;
        sumXX += dataPoint.x * dataPoint.x;
        sumXY += dataPoint.x * dataPoint.y;
        sumYY += dataPoint.y * dataPoint.y;
        if (dataPoint.x >= XMax) {
            XMax = (int) dataPoint.x;
        }
        if (dataPoint.y >= YMax) {
            YMax = (int) dataPoint.y;
        }
        //把每个点的具体坐标存入 ArrayList 中,备用
```

```
        xy[0] = (int) dataPoint.x + "";
        xy[1] = (int) dataPoint.y + "";
        if (dataPoint.x != 0 && dataPoint.y != 0) {
            System.out.print(xy[0] + ",");
            System.out.println(xy[1]);
            try {
                listX.add(pn, xy[0]);
                listY.add(pn, xy[1]);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        ++pn;
        coefsValid = false;
    }
    /*返回 x 的回归线函数的值
     */
    public float at(int x) {
        if (pn < 2)
            return Float.NaN;
        validateCoefficients();
        return a0 + a1 * x;
    }
    public void reset() {
        pn = 0;
        sumX = sumY = sumXX = sumXY = 0;
        coefsValid = false;
    }
    /*返回误差
     */
    public double getR() {
        //遍历这个 list 并计算分母
        for (int i = 0; i < pn - 1; i++) {
            float Yi = (float) Integer.parseInt(listY.get(i).toString());
            float Y = at(Integer.parseInt(listX.get(i).toString()));
            float deltaY = Yi - Y;
            float deltaY2 = deltaY * deltaY;
            sumDeltaY2 += deltaY2;
        }
        sst = sumYY - (sumY * sumY) / pn;
        E = 1 - sumDeltaY2 / sst;
        return round(E, 4);
    }
    //用于实现精确的四舍五入
    public double round(double v, int scale) {
        if (scale < 0) {
            throw new IllegalArgumentException(
                    "这个比例必须是一个正整数或零");
        }
        BigDecimal b = new BigDecimal(Double.toString(v));
        BigDecimal one = new BigDecimal("1");
```

```
        return b.divide(one, scale, BigDecimal.ROUND_HALF_UP).doubleValue();
                //向"最接近的"数字舍入,如果与两个相邻数字的距离相等,则为向上舍入的舍入模式
        }
        public float round(float v, int scale) {
            if (scale < 0) {
                throw new IllegalArgumentException(
                        "这个比例必须是一个正整数或零");
            }
            BigDecimal b = new BigDecimal(Double.toString(v));
            BigDecimal one = new BigDecimal("1");
            return b.divide(one, scale, BigDecimal.ROUND_HALF_UP).floatValue();
        }
    }
```

程序运行界面如图 3-2 所示。

输入数据：

```
{ { 1.1, 1.0, 1.2, 1.1, 0.9 }
  { 2.0, 2.0, 1.8, 1.9, 2.1 }
  { 3.2, 3.2, 3.0, 2.9, 2.9 }
10.1, 10.2, 10.0, 10.1, 10.0
```

(a)

运行结果输出：

```
a(0)=53.42
a(1)=98.78999999999999
a(2)=0.0
a(3)=40.4
q=393756.25257799996  s=280.6265320948823  r=NaN
v(0)=NaN
v(1)=NaN
v(2)=0.0
u=399388.454698
```

(b)

图 3-2　多分类逻辑回归程序运行界面

从运行结果可以看出回归系数 a 的值依次是 53.42、98.789 999、0.0、40.4；偏差的平方和 q 的值为 393 756.2525；标准平方差的值 s 为 280.626；复相关系数 r 的值是 NaN；自变量的偏相关系数分别是 NaN、NaN 和 0.0；最后得出回归平方和 u 的值是 399 388.4546。

3.4　基于阿里云数加平台的逻辑回归实例

3.4.1　二分类逻辑回归应用实例

对于表 bank_data 进行逻辑回归二分类的操作，分类算法总体目的是通过训练有标签数据，来给无标签数据赋标签，二分类只是标签只有两个的特殊情况。而表 bank_data 只是一个信息表，可以把某一列视为标签列，通过拆分组件的功能，一部分用来训练，一部分用来预测。其实，正常情况下，需要预测的数据是没有标签的，这时便可以使用训练后的训练数据给预测签数据赋标签。而这里，拆分过后的预测数据相当于也是有标签的，便可以预测其标签，然后与真实标签进行对比，得出该分类算法的准确率等相关信息。阿里云机器学习平台使用分类算法的目的大多是以上两种思路，后面的分类算法大多如此。其中，二分类逻辑回归便属于后者。

表 bank_data 的详细信息第 2 章已经提及。流程图如图 3-3 所示，由于逻辑回归本身就属于计算概率来进行分类，因此，最好将数据归一化，将归一化后的数据进行拆分，一部分

用来进行训练，一部分进行预测，从图中可以看到，拆分组件下面有两根线，左边的线代表将拆分后的训练数据放入逻辑回归二分类算法组件中进行训练，右边的线与逻辑回归二分类组件的线一起连入预测组件则表示用训练过后的训练数据给拆分过后的预测数据赋标签。最后的结果用混淆矩阵呈现出来。

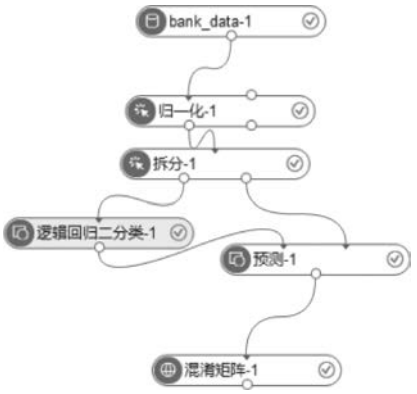

图 3-3　逻辑回归二分类流程图

逻辑回归二分类的字段信息与参数设置如图 3-4 所示，假设列 y 为标签列，因为该列表示“是否有定期存款”，只有两种不同的数据值，正好用来做二分类，因此目标列选择 y。训练特征列只能选择 double 和 int 类型字段，这里选择的是所有的符合条件的归一化后的字段。参数设置选择的是默认参数，根据需要也可调整，其中，正则项有 None，L1，L2 三种，是不同的防止过拟合的方法。

图 3-4　逻辑回归二分类方法的设置界面

拆分的参数设置如图 3-5 所示，为训练数据与预测数据的比例，这里选择的是 0.7。

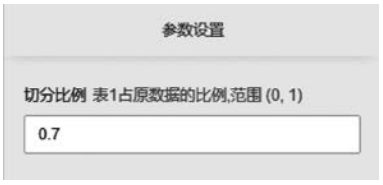

图 3-5　拆分方法的设置界面

预测的字段设置如图 3-6 所示，特征列的选择与图 3-4 中特征列选择一致，原样输出列选择的是标签列 y，最后 3 项是默认的结果输出列名，也可自行命名。

混淆矩阵的字段设置如图 3-7 所示，它是一个可视化工具。原数据的标签列列名选择的是 y，预测结果的“标签列”与“详细列”这两个参数不能共存，如指定阈值，则应自己命名预测结果的详细列列名，并指定正样本的标签值。

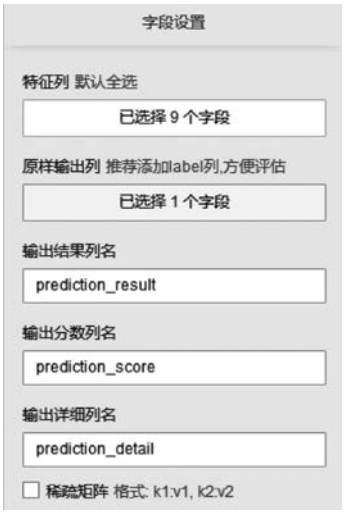

图 3-6　预测方法的设置界面

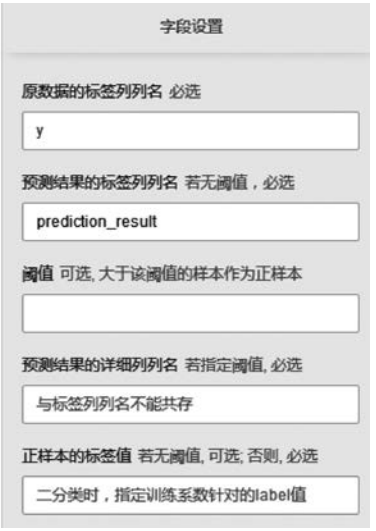

图 3-7　混淆矩阵方法的设置界面

预测组件的实验结果如图 3-8 所示，预测结果展示的是预测集进行预测得到不同标签的概率。如第一行，样本真实标签为“0”，预测标签为“0”的概率为 0.964，为“1”的概率为 0.036；因此，取 0.964，预测该样本标签为“0”。

y	prediction_result	prediction_score	prediction_detail
0	0	0.9637856160283...	{"0": 0.9637856160283249, "1": 0.0362143839716751}
1	1	0.6943437149809...	{"0": 0.3056562850190917, "1": 0.6943437149809083}
0	0	0.9699556975638...	{"0": 0.9699556975638698, "1": 0.0300
0	0	0.8764930218471...	{"0": 0.8764930218471138, "1": 0.1235
0	0	0.947803218268248	{"0": 0.947803218268248, "1": 0.0521
0	0	0.8669427586128...	{"0": 0.8669427586128203, "1": 0.1330572413871796}
0	0	0.9598956920041...	{"0": 0.9598956920041128, "1": 0.04010430799588717}
0	0	0.9625913556977...	{"0": 0.9625913556977236, "1": 0.03740864430227647}
0	0	0.8556344729114...	{"0": 0.8556344729114088, "1": 0.1443655270885912}
0	0	0.9655674393430...	{"0": 0.9655674393430479, "1": 0.03443256065695215}
0	0	0.9441553485010...	{"0": 0.9441553485010012, "1": 0.05584465149899878}
0	0	0.8505428738122...	{"0": 0.8505428738122986, "1": 0.1494571261877014}
0	0	0.8539415929646...	{"0": 0.8539415929646736, "1": 0.1460584070353264}
1	0	0.7422351581784...	{"0": 0.7422351581784865, "1": 0.2577648418215134}
0	0	0.9740755981787...	{"0": 0.9740755981787624, "1": 0.02592440182123758}
0	0	0.9344448140553...	{"0": 0.9344448140553643, "1": 0.06555518594463569}
0	0	0.964882730605	{"0": 0.964882730605

{"0": 0.305656285
0.6943437149805

图 3-8　预测组件的实验结果

从混淆矩阵的统计信息里可以评估模型的准确率等参数，如图 3-9 所示。

模型	正确数	错误数	总计	准确率	准确率	召回率	F1指标
0	10943	1147	12090	89.942%	90.513%	99.041%	94.585%
1	262	106	368	89.942%	71.196%	18.595%	29.488%

图 3-9　混淆矩阵组件的实验结果

从模型中可以查看训练好的逻辑回归二分类模型，如图 3-10 所示。

图 3-10　逻辑回归输出

3.4.2　多分类逻辑回归应用实例

逻辑回归多分类流程图如图 3-11 所示，该组件也可进行二分类操作，当该组件做多分类操作时，需要选择有多值的列作为标签列，previous 列满足此条件，因此设其为标签列。

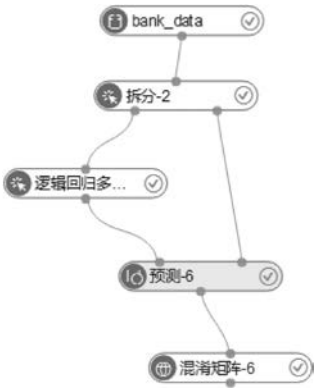

图 3-11　逻辑回归多分类流程图

其中，逻辑回归多分类的字段设置和参数设置如图 3-12 所示，除了将目标列改为 previous 列，其他与逻辑回归二分类的设置一致，这里不再赘述。

预测组件得到的结果如图 3-13 所示。

混淆矩阵中的标签统计信息如图 3-14 所示。

从模型中查看训练好的逻辑回归多分类模型如图 3-15 所示。

图 3-12　逻辑回归多分类方法设置界面

previous	prediction_result	prediction_score	prediction_detail
0	0	0.9171087304043...	{"0": 0.9171087304043734, "1": 0.09286609678976802, "2": 0.0002199318288866513, "3": 4.3833475889864...
0	0	0.9910072201144...	{"0": 0.9910072201144109, "1": 0.009837398180320799, "2": 3.844547690417482e-05, "3": 5.357969131260...
0	0	0.6648318524656...	{"0": 0.6648318524656754, "1": 0.3148744699894324, "2": 0.01811324358109297, "3": 0.0006787398220893...
0	0	0.6663645708905...	{"0": 0.6663645708905497, "1": 0.3136754565804028, "2": 0.01866720437501295, "3": 0.0006875116557745...
0	0	0.9841247004561...	{"0": 0.9841247004561563, "1": 0.01287883536494262, "2": 6.864833346084681e-05, "3": 1.2908095210568...
0	0	0.742104102390657	{"0": 0.742104102390657, "1": 0.2447465487541765, "2": 0.016458084511222, "3": 0.0007967852103548327...
0	0	0.8167450119999...	{"0": 0.8167450119999072, "1": 0.1880132896068601, "2": 0.01469640483947962, "3": 0.0005611346884559...
0	0	0.9833168862832...	{"0": 0.9833168862832321, "1": 0.01352766077123254, "2": 6.989988380640126e-05, "3": 1.3365953042230...
0	0	0.9879039562723...	{"0": 0.9879039562723315, "1": 0.01305371221036549, "2": 4.362560246415237e-05, "3": 5.1895150697041...
0	0	0.9837707465253...	{"0": 0.9837707465253062, "1": 0.01720241517277895, "2": 5.096473259483943e-05, "3": 4.4567944605742...
0	0	0.9860753081886...	{"0": 0.9860753081886684, "1": 0.01486959494314224, "2": 4.777245356607437e-05, "3": 4.5741530472507...
0	0	0.5499373671320...	{"0": 0.5499373671320485, "1": 0.3293304897824758, "2": 0.06289005357318661, "3": 0.0099634116579731...
0	0	0.6742362994956...	{"0": 0.6742362994956456, "1": 0.3057744321680063, "2": 0.01811757749821807, "3": 0.0006466466554541...
0	0	0.6804477009964...	{"0": 0.6804477009964867, "1": 0.3015917324763912, "2": 0.0182267361494518, "3": 0.0007693079078498 4...
0	0	0.984521607947193	{"0": 0.984521607947193, "1": 0.01649086316211188, "2": 4.904805504506203e-05, "3": 4.84780425030735...
0	0	0.9844097588517...	{"0": 0.9844097588517742, "1": 0.0165249881980249, "2": 4.445457288351488e-05, "3": 3.19563827833238...
1	0	0.6878051223439...	{"0": 0.6878051223439711, "1": 0.2928659963697695, "2": 0.01656872203023752, "3": 0.0006368377558753...

图 3-13　预测组件的实验结果

模型	正确数	错误数	总计	准确率	准确率	召回率	F1指标
0	10635	1195	11830	89.891%	89.899%	99.495%	94.454%
1	255	248	503	89.276%	50.696%	19.144%	27.793%
2	11	10	21	98.057%	52.381%	4.564%	8.397%
3	0	1	1	99.482%	0%	0%	0%
4	0	0	0	99.806%	0%	0%	0%
5	0	0	0	99.968%	0%	0%	0%
6	0	0	0	99.984%	0%	0%	0%

图 3-14　混淆矩阵组件的实验结果

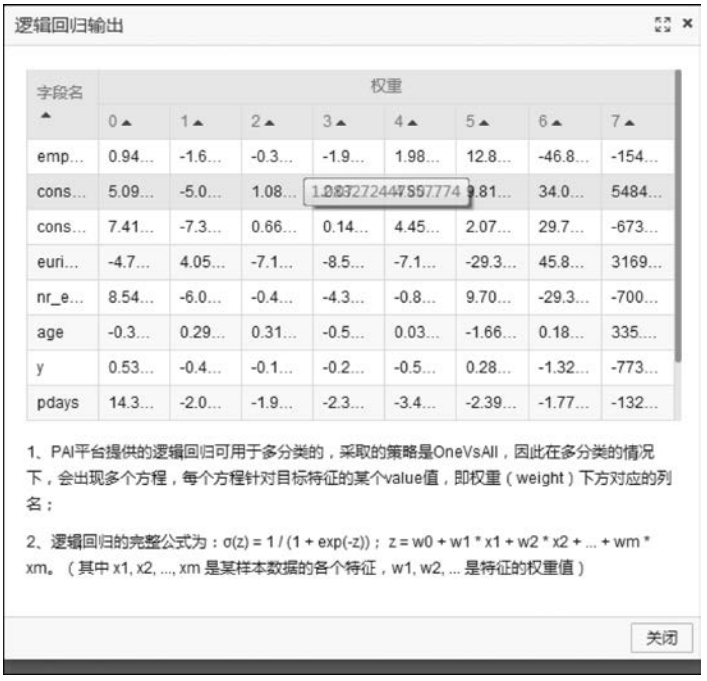

图 3-15　逻辑回归输出

3.5　小　　结

回归是研究一组随机变量(y_1,y_2,…,y_i)和另一组变量(x_1,x_2,…,x_k)之间关系的统计分析方法。

线性回归是利用称为线性回归方程的最小平方函数对一个或多个自变量和因变量之间关系进行建模的一种回归分析。

逻辑回归本质上是线性回归，只是在特征到结果的映射中加入了一层函数映射，即先把特征线性求和，然后使用函数 $g(z)$作为最终假设函数来预测，它又分为二分类逻辑回归和多分类逻辑回归。

思　考　题

1. 什么是回归、线性回归和逻辑回归？
2. 简述逻辑回归的特点。
3. 基于阿里云数加平台进行二分类和多分类逻辑回归操作。