Central University of Finance and Economics

中央财经大学

数据挖掘课程

代码文档

吴宇翀

2017310836

WUYUCHONG.COM

指导老师：马景义

2020 年 5 月 23 日

目录

1 简介

此文档为两个算法的代码文档，包括了 **Logisitic 回归**和**神经网络**两个模型算法。

1. model 文件夹：存放两个算法模型的类
2. source 文件夹：存放依赖函数
3. main 文件：测试用例 demo

2 Logistic 回归算法

我们使用**梯度下降**的优化方法构建 logit 模型。

2.1 模型求解步骤

如果 p 是一个事件的概率，这个事件的发生比率就是 p/(1-p)。逻辑回归就是建立模型预测这一比率的对数:

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \cdots + \beta_P x_P$$

即：

$$p = \frac{1}{1 + \exp\left[-\left(\beta_0 + \beta_1 x_1 + \cdots + \beta_P x_P\right)\right]}$$

假设我们有 n 个独立的训练样本 {(x1, y1) ,(x2, y2),..., (xn, yn)}，y={0, 1}。那每一个观察到的样本 (xi, yi) 出现的概率是：

$$P\left(\mathrm{y}_i, \mathrm{x}_i\right) = P\left(\mathrm{y}_i = 1|\mathrm{x}_i\right)^{y_i} \left(1 - P\left(\mathrm{y}_i = 1|\mathrm{x}_i\right)\right)^{1-y_i}$$

那我们的整个样本集，也就是 n 个独立的样本出现的似然函数为:

$$L(\theta) = \prod P\left(\mathrm{y}_i = 1|\mathrm{x}_i\right)^{y_i} \left(1 - P\left(\mathrm{y}_i = 1|\mathrm{x}_i\right)\right)^{1-y_i}$$

那么，损失函数（cost function）就是最大似然函数**取对数**。[1]

用 $L(\theta)$ 对 θ 求导，得到：

[1] 最大似然法就是求模型中使得似然函数最大的系数取值 *

$$\frac{\partial L(\theta)}{\partial \theta} = \sum_{i=1}^{n} y_i x_i - \sum_{i}^{n} \frac{e^{\theta^T x_i}}{1 + e^{\theta_{x_i}}} x_i = \sum_{i=1}^{n} \left(y_i - \sigma\left(\theta^T x_i\right)\right) x_i$$

令该导数为 0，无法解析求解。使用梯度下降法 @ 汪宝彬 2011 随机梯度下降法的一些性质，那么：

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial L(\theta)}{\partial \theta} = \theta^t - \alpha \sum_{i=1}^{n} \left(y_i - \sigma\left(\theta^T x_i\right)\right) x_j$$

在进行训练之前，我们对各个变量的数据进行标准化处理。然后，我们让学习率逐步递减进行训练。@LiFeng

2.2 代码实现 - 逐步讲解（Step by Step）

2.2.1 包和数据导入

```
import numpy as np
import math
from sklearn import datasets
```

我们读取经典的 iris 数据集。[2]

```
iris = datasets.load_iris()
X = iris['data']
y = iris['target']
X = X[y!=2]
y = y[y!=2]
X[0:5]
```

```
## array([[5.1, 3.5, 1.4, 0.2],
##        [4.9, 3. , 1.4, 0.2],
##        [4.7, 3.2, 1.3, 0.2],
##        [4.6, 3.1, 1.5, 0.2],
##        [5. , 3.6, 1.4, 0.2]])
```

```
y[0:5]
```

```
## array([0, 0, 0, 0, 0])
```

[2] 只展示前 5 行

2.2.2 定义 sigmoid 函数

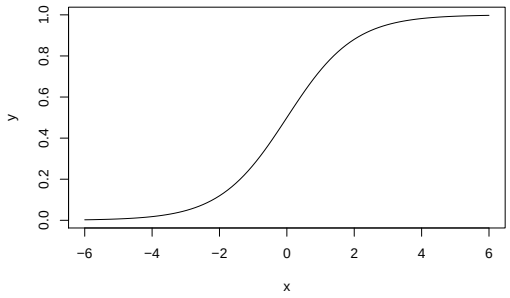

图 1: sigmoid function

适用于向量的 sigmoid 函数

```
def sigmoidVector(Xi,thetas):
    params = - np.sum(Xi * thetas)
    outcome = 1 /(1 + math.exp(params))
    return outcome
```

适用于矩阵的 sigmoid 函数

```
def sigmoidMatrix(Xb,thetas):
    params = - Xb.dot(thetas)
    outcome = np.zeros(params.shape[0])
    for i in range(len(outcome)):
        outcome[i] = 1 /(1 + math.exp(params[i]))
    return outcome
```

带阈值判别的 sigmoid 函数

- 阈值（threshold）：用于给出概率后进行分类，默认为 50%

```
def sigmoidThreshold(Xb,thetas):
    params = - Xb.dot(thetas)
    outcome = np.zeros(params.shape[0])
    for i in range(len(outcome)):
        outcome[i] = 1 /(1 + math.exp(params[i]))
        if outcome[i] >= 0.5:
            outcome[i] = 1
        else:
            outcome[i] = 0
    return outcome
```

2.2.3 定义损失函数

损失函数（cost function）就是最大似然函数**取对数**

```
def costFunc(Xb,y):
    sum = 0.0
    for i in range(m):
        yPre = sigmoidVector(Xb[i,], thetas)
        if yPre == 1 or yPre == 0:
            return float(-2**31)
        sum += y[i] * math.log(yPre) + (1 - y[i]) * math.log(1-yPre)
    return -1/m * sum
```

2.2.4 用梯度下降法进行训练

初始化：

- 学习率（alpha）：用于调整每次迭代的对损失函数的影响大小
- 准确度（accuracy）：作为终止迭代的评判指标

```
thetas = None
m = 0
alpha = 0.01
accuracy = 0.001
```

在第一列插入 1，构成 Xb 矩阵

```
thetas = np.full(X.shape[1]+1,0.5)
m = X.shape[0]
a = np.full((m, 1), 1)
Xb = np.column_stack((a, X))
n = X.shape[1] + 1
```

使用梯度下降法进行迭代：

```
count = 1
while True:
    before = costFunc(Xb, y)
    c = sigmoidMatrix(Xb, thetas)-y
    for j in range(n):
        thetas[j] = thetas[j] -alpha * np.sum(c * Xb[:,j])
    after = costFunc(Xb, y)
    if after == before or math.fabs(after - before) < accuracy:
```

```
        print(" 迭代完成, 损失函数值:",after)
        break
    print(" 迭代次数:",count)
    print(" 损失函数变化",(after - before))
    count += 1
```

```
## 迭代次数: 1
## 损失函数变化 5.025121118513411
## 迭代次数: 2
## 损失函数变化 -3.292032434097819
## 迭代次数: 3
## 损失函数变化 -0.9713865369569756
## 迭代次数: 4
## 损失函数变化 2.649715136143148
## 迭代次数: 5
## 损失函数变化 -6.0552442422478245
## 迭代次数: 6
## 损失函数变化 -0.08147412308098556
## 迭代次数: 7
## 损失函数变化 -0.05224409412488061
## 迭代完成, 损失函数值: 0.00953683385119681
```

迭代完成后得到的参数:

```
thetas
```

```
## array([ 0.05586408, -0.68733466, -1.75042736,  2.95078531,  1.58964498])
```

2.2.5 预测

```
a = np.full((len(X),1),1)
Xb = np.column_stack((a,X))
sigmoidThreshold(Xb, thetas)
```

```
## array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
##        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
##        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
##        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
##        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
##        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

2.3 代码实现 - 类封装

可以调整的参数包括：

- 学习率（alpha）：用于调整每次迭代的对损失函数的影响大小
- 准确度（accuracy）：作为终止迭代的评判指标
- 阈值（threshold）：用于给出概率后进行分类，默认为 50%

```
# ----------------- 导入基本模块 -----------------
import numpy as np
import math

# ------------ 导入 source 中定义的函数 ------------
from source.sigmoidVector import sigmoidVector
from source.sigmoidMatrix import sigmoidMatrix
from source.sigmoidThreshold import sigmoidThreshold

# ----------------- 定义 base 类 -----------------
class Regression(object):
    def __init__(self, X, y, threshold = 0.5):
        self.thetas = None
        self.X = X
        self.y = y

# ---------------- 定义逻辑回归类 ----------------
class LogsiticRegression(Regression):
    def __init__(self, X, y, threshold = 0.5):
        Regression.__init__(self, X, y, threshold = 0.5) # 继承 Regression 类
        self.m = 0
        self.threshold = threshold
        self.epoch = 1

    def fit(self, alpha = 0.01, accuracy = 0.001):
        self.thetas = np.full(self.X.shape[1] + 1,0.5)
        self.m = self.X.shape[0]
        a = np.full((self.m, 1), 1)
        Xb = np.column_stack((a,self.X))
        n  = self.X.shape[1]+1

        while True:
            before = self.costFunc(Xb, y)
```

```
            c = sigmoidMatrix(Xb, self.thetas) - y
            for j in range(n):
                self.thetas[j] = self.thetas[j] - alpha * np.sum(c * Xb[:,j])
            after = self.costFunc(Xb, y)
            if after == before or math.fabs(after - before) < accuracy:
                break
            self.epoch += 1

    def costFunc(self, Xb, y):
        sum = 0.0
        for i in range(self.m):
            yPre = sigmoidVector(Xb[i,], self.thetas)
            if yPre == 1 or yPre == 0:
                return float(-2**31)
            sum += y[i] * math.log(yPre) + (1 - y[i]) * math.log(1 - yPre)
        return -1/self.m * sum

    def predict(self):
        a = np.full((len(X), 1), 1)
        Xb = np.column_stack((a, X))
        return sigmoidThreshold(Xb, self.thetas, self.threshold)
```

2.4 测试用例

我们使用经典的 iris 数据集作为测试用例。

```
iris = datasets.load_iris()
X = iris['data']
y = iris['target']
X = X[y!=2]
y = y[y!=2]

Logstic = LogsiticRegression(X, y)
Logstic.fit()
print("epoch:",Logstic.epoch)
```

```
## epoch: 8
```

```
print("theta:",Logstic.thetas)
```

```
## theta: [ 0.05586408 -0.68733466 -1.75042736  2.95078531  1.58964498]
```

```
y_predict = Logstic.predict()
y_predict
```

```
## array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
##        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
##        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
##        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
##        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
##        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

3 神经网络算法

我们使用**随机梯度下降**的优化方法构建一个较为简单的神经网络模型。[3]

3.1 模型求解步骤

3.1.1 神经元

神经元是神经网络的基本单元。@Chua1988Cellular 神经元先获得输入，然后执行某些数学运算后，再产生一个输出。@ 庄镇泉 1990 神经网络与神经计算机: 第二讲

对于一个二输入神经元，输出步骤为：先将两个输入乘以权重，把两个结果想加，再加上一个偏置，最后将它们经过激活函数处理得到输出。[4]

$$y = f(x1 \times w1 + x2 \times w2 + b)$$

我们选择 sigmoid 函数作为神经网络的激活函数。

$$S(x) = \frac{1}{1 + e^{-x}}$$

$$S'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = S(x)(1 - S(x))$$

[3]考虑到神经网络是较为复杂的一类模型，在此我们只构造它的初始版本，限制较多，不具备广泛实用性。

[4]神经元（Neurons）权重（weight）偏置（bias）激活函数（activation function）

3.1.2 神经网络

我们搭建一个具有 2 个输入、一个包含 2 个神经元的隐藏层（h1 和 h2）、包含 1 个神经元的输出层（o1）的简单神经网络。[5] @ 阎平凡 2005 人工神经网络与模拟进化计算

3.1.3 前馈

把神经元的输入向前传递获得输出的过程称为前馈 [6]

3.1.4 损失

在训练神经网络之前，我们需要有一个标准定义，以便我们进行改进。我们采用均方误差（MSE）来定义损失（loss）：

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})$$

我们在训练中每次只取一个样本，那么损失函数为：

$$\begin{aligned}\text{MSE} &= \frac{1}{1}\sum_{i=1}^{1}\left(y_{true} - y_{pred}\right)^2 \\ &= \left(y_{true} - y_{pred}\right)^2 \\ &= \left(1 - y_{pred}\right)^2\end{aligned}$$

3.1.5 训练神经网络

训练神经网络就是将损失最小化，预测结果越好，损失就越低。

由于预测值是由一系列网络权重和偏置计算出来的，所以损失函数实际上是包含多个权重、偏置的多元函数：

$$L\left(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3\right)$$

由链式求导法则（以 w1 为例）：

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

[5] A neural network with: - 2 inputs - a hidden layer with 2 neurons (h1, h2) - an output layer with 1 neuron (o1)

[6] 前馈（feedforward）

这种向后计算偏导数的系统称为反向传导。[7]

3.1.6 随机梯度下降优化方法

我们使用随机梯度下降（SGD）的优化算法 @ 王功鹏 2018 基于卷积神经网络的随机梯度下降算法来逐步改变网络的权重 w 和偏置 b，使损失函数会缓慢降低，从而改进我们的神经网络。以 w1 为例：

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

3.1.7 权重的初始化

神经网络中结点的各权重 (weight) 和偏置 (bias) 的初始化均服从标准正态分布

$$Weight_i, Bias_i \sim N(0,1)$$

3.2 代码实现 - 逐步讲解（Step by Step）

3.2.1 包和数据导入

```
import numpy as np
import math
```

我们读取经典的 iris 数据集，取其前两个自变量。[8]

```
iris = datasets.load_iris()
X = iris['data']
y = iris['target']
X = X[y!=2][:,0:2]
y = y[y!=2]
X[0:5]
```

```
## array([[5.1, 3.5],
##        [4.9, 3. ],
##        [4.7, 3.2],
##        [4.6, 3.1],
##        [5. , 3.6]])
```

[7] 反向传导（backpropagation）

[8] 只展示前 5 行

```
y[0:5]
```

```
## array([0, 0, 0, 0, 0])
```

3.2.2 定义 sigmoid 函数

$$f(x) = \frac{1}{1 + e^{-x}}$$

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

3.2.3 定义 sigmoid 的导函数

$$\frac{\mathrm{d}f}{\mathrm{d}x} = f(x) * (1 - f(x))$$

```
def deriv_sigmoid(x):
    fx = sigmoid(x)
    return fx * (1 - fx)
```

3.2.4 定义均方误差损失函数：

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})$$

```
def mse_loss(y_true, y_pred):
    return ((y_true - y_pred) ** 2).mean()
```

3.2.5 权重和截距的初始化

$$L\left(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3\right)$$

```
w1 = np.random.normal()
w2 = np.random.normal()
w3 = np.random.normal()
w4 = np.random.normal()
```

```
w5 = np.random.normal()
w6 = np.random.normal()

b1 = np.random.normal()
b2 = np.random.normal()
b3 = np.random.normal()
```

3.2.6 前馈

将输入向前传递获得输出。

```
def feedforward(x):
    h1 = sigmoid(w1 * x[0] + w2 * x[1] + b1)
    h2 = sigmoid(w3 * x[0] + w4 * x[1] + b2)
    o1 = sigmoid(w5 * h1 + w6 * h2 + b3)
    return o1
```

3.2.7 设置学习率和迭代次数

```
learn_rate = 0.1
epochs = 1000
record = np.array((None, None))
```

3.2.8 训练神经网络

```
record = np.array([None, None])
for epoch in range(epochs):
    for x, y_true in zip(X, y):
        # --- Do a feedforward (we'll need these values later)
        sum_h1 = w1 * x[0] + w2 * x[1] + b1
        h1 = sigmoid(sum_h1)

        sum_h2 = w3 * x[0] + w4 * x[1] + b2
        h2 = sigmoid(sum_h2)

        sum_o1 = w5 * h1 + w6 * h2 + b3
        o1 = sigmoid(sum_o1)
        y_pred = o1
```

```
        # --- Calculate partial derivatives.
        # --- Naming: d_L_d_w1 represents "partial L / partial w1"
        d_L_d_ypred = -2 * (y_true - y_pred)

        # Neuron o1
        d_ypred_d_w5 = h1 * deriv_sigmoid(sum_o1)
        d_ypred_d_w6 = h2 * deriv_sigmoid(sum_o1)
        d_ypred_d_b3 = deriv_sigmoid(sum_o1)

        d_ypred_d_h1 = w5 * deriv_sigmoid(sum_o1)
        d_ypred_d_h2 = w6 * deriv_sigmoid(sum_o1)

        # Neuron h1
        d_h1_d_w1 = x[0] * deriv_sigmoid(sum_h1)
        d_h1_d_w2 = x[1] * deriv_sigmoid(sum_h1)
        d_h1_d_b1 = deriv_sigmoid(sum_h1)

        # Neuron h2
        d_h2_d_w3 = x[0] * deriv_sigmoid(sum_h2)
        d_h2_d_w4 = x[1] * deriv_sigmoid(sum_h2)
        d_h2_d_b2 = deriv_sigmoid(sum_h2)

        # --- Update weights and biases
        # Neuron h1
        w1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w1
        w2 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_w2
        b1 -= learn_rate * d_L_d_ypred * d_ypred_d_h1 * d_h1_d_b1

        # Neuron h2
        w3 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w3
        w4 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_w4
        b2 -= learn_rate * d_L_d_ypred * d_ypred_d_h2 * d_h2_d_b2

        # Neuron o1
        w5 -= learn_rate * d_L_d_ypred * d_ypred_d_w5
        w6 -= learn_rate * d_L_d_ypred * d_ypred_d_w6
        b3 -= learn_rate * d_L_d_ypred * d_ypred_d_b3
```

```
    # --- 计算
    y_preds = np.apply_along_axis(feedforward, 1, X)
    loss = mse_loss(y, y_preds)
    new = np.array([epoch, loss])
    record = np.vstack([record,new])
```

4 查看损失函数的变化

```
record[1:10:,1]
```

```
## array([0.3331110607724303, 0.3444369130606642, 0.3461342540766672,
##        0.3464988720143262, 0.3457255311281579, 0.34328664914464,
##        0.33899018716039564, 0.3330072110848336, 0.32543469654464147],
##       dtype=object)
```

我们画出损失函数的变化图像，可以看出，随着迭代次数的增加，均方误差先快速减小，后趋向于稳定。

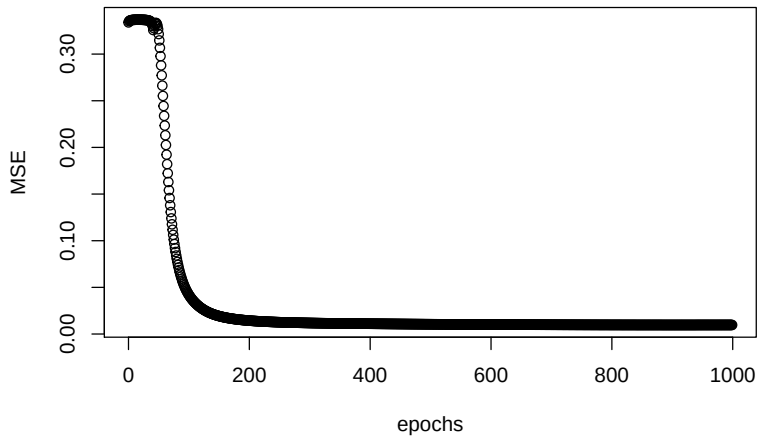

图 2: 损失函数的变化图像

4.1 代码实现 - 类封装

可以调整的参数包括：

- 学习率（alpha）：用于调整每次迭代的对损失函数的影响大小
- 准确度（accuracy）：作为终止迭代的评判指标
- 阈值（threshold）：用于给出概率后进行分类，默认为 50%

5 参考文献