机器学习概论 实验报告

Lab1: LR

2020 年 12 月 4 日

目录

1 实验简介

本实验为 Logistics Regression 模型实现实验, 我们的目标是根据 Horse-colic 数据集中的部分样本作回归, 以梯度上升法为基础, 并用测试集测试精确度.

2 理论基础

2.1 多元回归方程形式

多元回归方程形式:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p$$

写成矩阵形式为:

$$\boldsymbol{Y} = \boldsymbol{X\beta}$$

2.2 Logistics Regression 模型

Logistics Regression 模型中, 利用了 sigmoid 函数来估计概率

$$P(\boldsymbol{Y}=1) = \frac{1}{1+e^{\boldsymbol{X\beta}}}$$

为了估计出参数 $\boldsymbol{\beta}$, 课本采用了 最大似然估计. 以二分类问题为例, 我们有:

$$P(y|x,\beta) = P(y=1|x,\beta)^y[1-P(y=1|x,\beta)]^{1-y}$$

由此可以写出似然函数:

$$\mathcal{L}(\beta) = \prod_{i=1}^{n} P(y_i|x_i,\beta) = \prod_{i=1}^{n} \left(\frac{1}{1+e^{-x_i\beta}}\right)^{y_i} \left(1-\frac{1}{1+e^{-x_i\beta}}\right)^{1-y_i}$$

对其取对数即得到对数似然函数:

$$\log \mathcal{L}(\beta) = \sum_{i=1}^{n} \left[y_i \log\left(\frac{1}{1+e^{-x_i\beta}}\right) + (1-y_i)\log\left(1-\frac{1}{1+e^{-x_i\beta}}\right)\right]$$

我们可以将它的相反数当做损失函数:

$$J(\beta) = -\log \mathcal{L}(\beta) = -\sum_{i=1}^{n} \left[y_i \log\left(P(y_i)\right) + (1-y_i)\log\left(1-P(y_i)\right)\right]$$

2.3 优化方法

为了用梯度法优化参数, 应当将损失函数对参数 β 求导:

$$\frac{\partial J(\beta)}{\partial \beta_j} = -\sum_{i=1}^{n}(y_i - f(x_i,\beta)) \cdot x_{ij} = \sum_{i=1}^{n}\left(\frac{1}{1+e^{-x_i\beta}} - y_i\right) \cdot x_{ij}$$

然后根据梯度下降法的原理:

$$\beta_{t+1} \leftarrow \beta_t - \alpha \nabla J(\beta)$$

即可进行迭代优化. 当然也可以使用 SGD 或 Newton法进行优化.

3 优化算法

3.1 Gradient Descent 算法(GD)

Algorithm 1 GD

Require: 训练的 epochs T; 初始化 $\beta = (w, b)$, 学习率 α

for 每个 epoch **do**
  $d\beta = 0$
  for 每个训练样本 x_i **do**
    $d\beta = d\beta + \sum_{i=1}^{n} \left(\frac{1}{1+e^{-x_i\beta}} - y_i \right) \cdot x_{ij}$
  $\beta = \beta - \alpha * d\beta$

3.2 Stochastic Gradient Descent 算法(SGD)

Algorithm 2 SGD

Require: 训练的 epochs T; 初始化 $\beta = (w, b)$, 学习率 α

for 每个 epoch **do**
  $d\beta = 0$
  随机选定样本序号 i
  $d\beta = d\beta + -\sum_{i=1}^{n} (y_i - f(x_i, \beta)) \cdot x_{ij} = \sum_{i=1}^{n} \left(\frac{1}{1+e^{-x_i\beta}} - y_i \right) \cdot x_{ij}$
  $\beta = \beta - \alpha * d\beta$

3.3 Newton法

Algorithm 3 Newton

Require: 训练的 epochs T; 初始化 $\beta = (w, b)$, 学习率 α

for 每个 epoch **do**
  $d\beta = 0$
  $dd\beta = 0$
  for 每个训练样本 x_i **do**
    $d\beta = d\beta + \sum_{i=1}^{n} \left(\frac{1}{1+e^{-x_i\beta}} - y_i \right) \cdot x_{ij}$
    $dd\beta = dd\beta + \left(\frac{1}{1+e^{-x_i\beta}} \right) \left(1 - \frac{1}{1+e^{-x_i\beta}} \right)$
  $\beta = \beta - \alpha * d\beta / dd\beta$

4 实验结果

4.1 总体对比

模型/算法	训练集准确度	测试集准确度	迭代次数
GD	0.94	1.0	300
SGD	0.97	0.93	5000
Newton	0.97	0.93	200
sklearn	0.96	0.93	300

可以看到就这个简单的数据集而言，GD算法已经能够达到非常好的表现了，而 SGD 算法则表现不那么佳，可能是样本太少造成的. 至于 Newton法 结果则与与 SGD算法 相似. 最后一行 sklearn 的结果只是给出一个 baseline.

4.2 GD算法实验结果

- 首先是损失率-迭代次数的图:

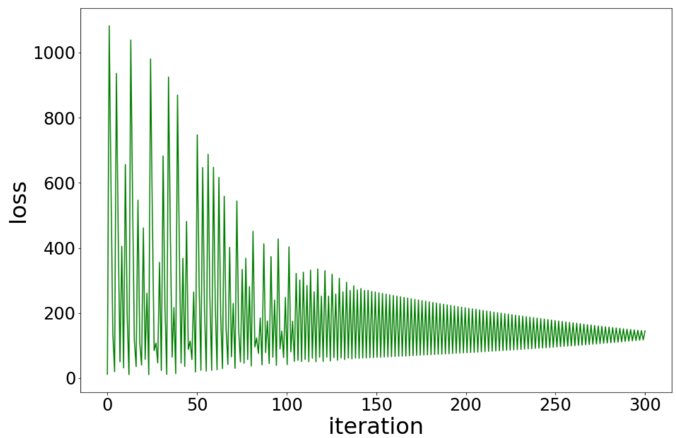

图 1: GD算法 损失率-迭代次数图(学习率=0.5)

可以看到损失率在波动范围内不断下降. 但显然这个波动太大了，这是由于**学习率过高引起的**，因此降低学习率到0.1，得到以下图:

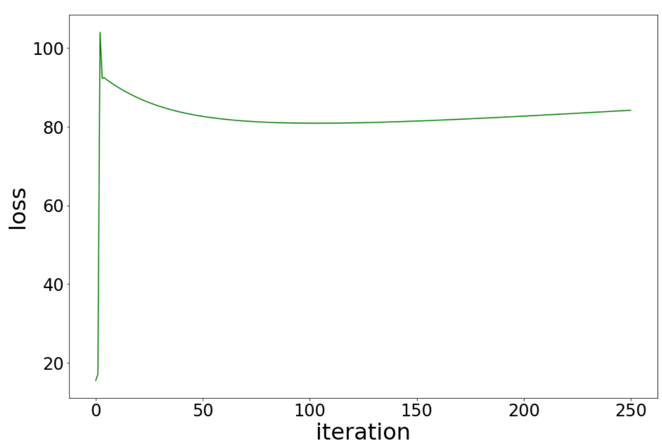

图 2: GD算法 损失率-迭代次数图(学习率=0.1)

- 经过调整参数, 可以得到在**学习率为 0.1, epoch为300时**, 能够在**训练集上达到0.94的准确率**, 在**测试集上达到 1.0 的准确率**.
- 训练结果可视化. 从图中可以看出来, 测试集上的绿色线将两类样本都划分开来了.

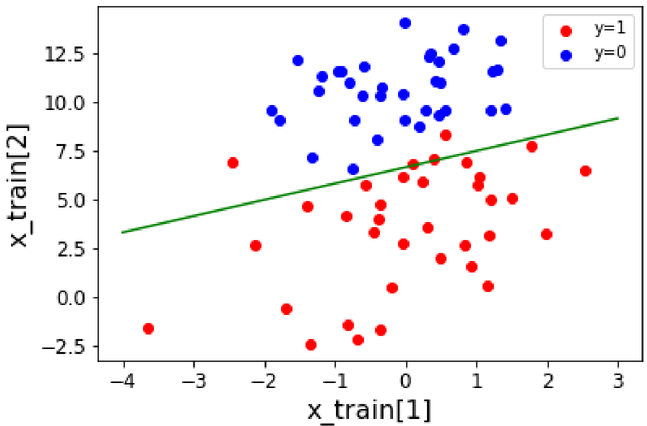

图 3: GD算法在训练集上的表现

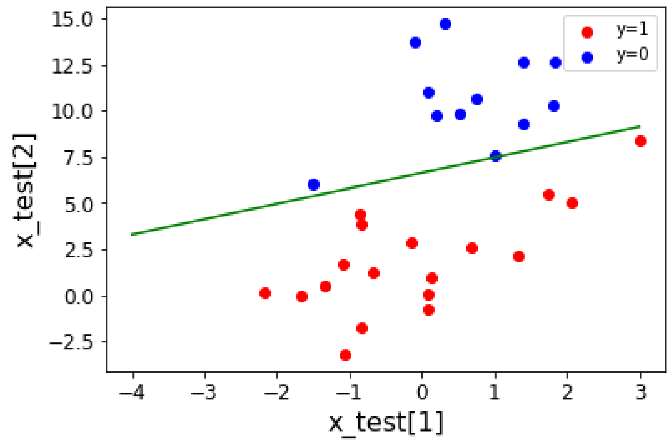

图 4: GD算法在测试集上的表现

4.3 SGD算法实验结果

- 经过调整参数, 可以得到在**学习率为 0.1, epoch为 5000 时**, 能够在**训练集上达到0.97的准确率**, 在**测试集上达到 0.93 的准确率**.
- 训练结果可视化:

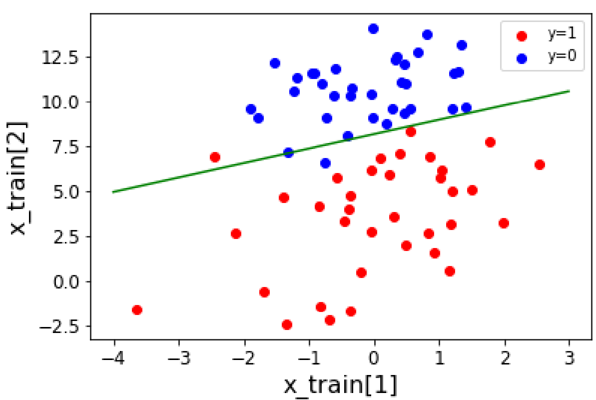

图 5: SGD算法在训练集上的表现

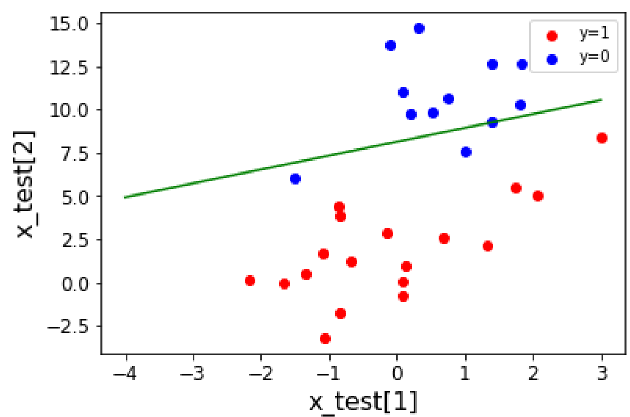

图 6: SGD算法在测试集上的表现

4.4 Newton法实验结果

- 经过调整参数, 可以得到在**学习率为 0.1, epoch为200时**, 能够在**训练集上达到0.94的准确率**, 在**测试集上达到 1.0 的准确率**.
- 训练结果可视化. 从图中可以看出来, 测试集上的绿色线将两类样本都划分开来了.

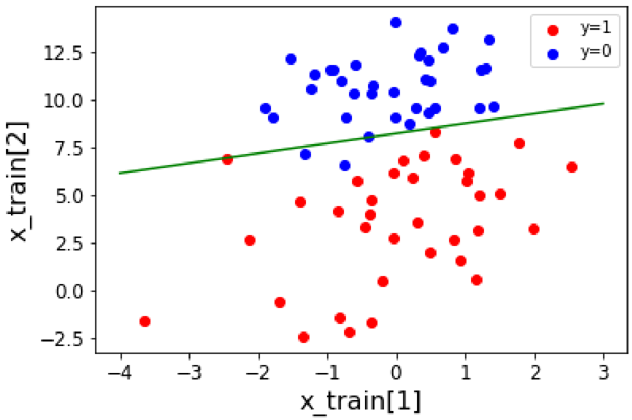

图 7: Newton法在训练集上的表现

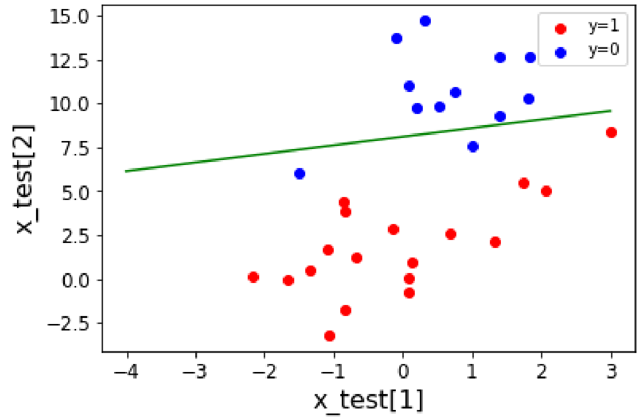

图 8: Newton法在测试集上的表现

4.5 sklearn 实验结果

直接调用sklearn能够得到在训练集上 0.957, 在测试集上 0.933 的效果

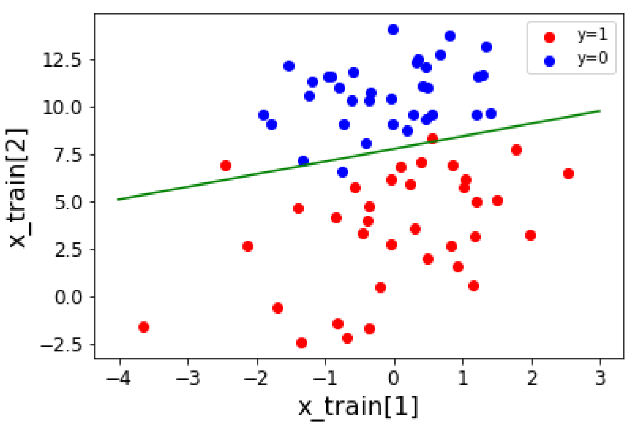

图 9: 调用sklearn在训练集上的表现

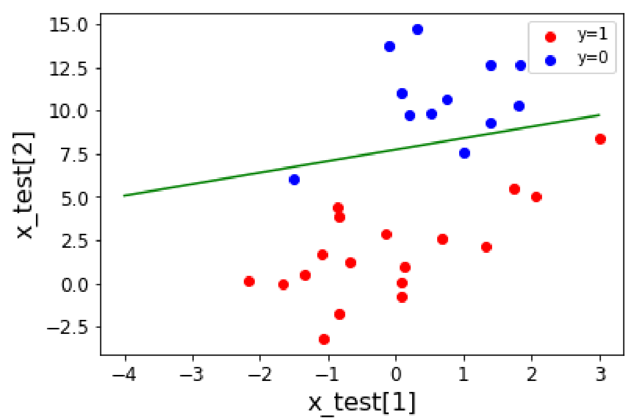

图 10: 调用sklearn在测试集上的表现

5 附录: 实验代码

5.1 对率回归算法代码(包括GD,SGD,Newton)

```python
import numpy as np

x_train = np.load("./data/LR/train_data.npy")
y_train = np.load("./data/LR/train_target.npy")
x_test = np.load("./data/LR/test_data.npy")
y_test = np.load("./data/LR/test_target.npy")

class LogitRegression:
    def __init__(self, x_train, y_train, alpha=0.2, epoch=-1, tol=1e-2, model='GD'):
        self.x_train = x_train
        self.y_train = y_train
        self.beta = np.ones(x_train.shape[1])
        self.beta = np.random.uniform(low=0, high=1, size=x_train.shape[1])
        self.alpha = alpha
        self.epoch = epoch
        self.tol = tol
        self.model = model

    def probability(self, x):
        return 1/(1+np.exp(-x @ self.beta))

    def loss_J(self):
        p = self.probability(self.x_train)
        return -np.sum([self.y_train[i] * np.log(p[i]) + (1-self.y_train[i]) * np.log(p[i])
                        for i in range(len(self.y_train))])

    @property
    def m(self):
        return len(self.y_train)

    def train(self):
        # stop by tol
        loss = []
        if self.model == 'GD':
            if self.epoch == -1:
                while True:
                    grad = np.zeros(self.x_train.shape)
                    for i in range(self.m):
                        grad[i] = (1/(1+np.exp(-self.x_train[i] @ self.beta)) - self.y_train[i]) * self.x_train[i]
                    delta = self.alpha * grad.mean(axis=0)
                    self.beta -= delta
                    if np.abs(delta).mean() < self.tol:
                        break
                return

            # stop by epoch
            for _ in range(self.epoch):
                grad = np.zeros(self.x_train.shape)
                loss.append(self.loss_J())
                for i in range(self.m):
                    grad[i] = (1/(1+np.exp(-self.x_train[i] @ self.beta)) - self.y_train[i]) * self.x_train[i]
                self.beta -= self.alpha * grad.mean(axis=0)
            return loss
        elif self.model == 'SGD':
            # stop by epoch
            for _ in range(self.epoch):
                loss.append(self.loss_J())
                i = int(np.random.uniform(low=0, high=self.m-1, size=1))
                grad = (1/(1+np.exp(-self.x_train[i] @ self.beta)) - self.y_train[i]) * self.x_train[i]
                self.beta -= self.alpha * grad
            return loss
        elif self.model == 'newton':
            # stop by epoch
            for _ in range(self.epoch):
                grad = np.zeros(self.x_train.shape)
                gradd = np.zeros(1)
                loss.append(self.loss_J())
                for i in range(self.m):
                    expxbeta = np.exp(-self.x_train[i] @ self.beta)
                    grad[i] = (1/(1+expxbeta) - self.y_train[i]) * self.x_train[i]
                    gradd += (1/(1+expxbeta)) * (1-1/(1+expxbeta))
                self.beta -= self.alpha * grad.mean(axis=0) / gradd
            return loss

```

```python

    def test(self, x_test, y_test):
        x_test = np.array(x_test)
        p = self.probability(x_test)
        acc = 1-np.sum(np.abs([float(round(i)) for i in lr.probability(x_test)] - y_test)) / len(y_test)
        return p, acc

    def get_xxyy(self, span):
        xx = np.linspace(*span, (span[1]-span[0])*100)
        yy = (-self.beta[0]-self.beta[1]*xx) / self.beta[2]
        return xx, yy

lr = LogitRegression(x_train, y_train, alpha=0.5, epoch=300, model='GD')
loss = lr.train()

print(lr.test(x_train, y_train)[1])
print(lr.test(x_test, y_test)[1])
```

Listing 1: 对率回归算法代码

5.2 绘图代码

```python
import matplotlib.pyplot as plt

xx, yy = lr.get_xxyy((-4, 3))
plt.xlabel('x_train[1]', size=16)
plt.ylabel('x_train[2]', size=16)
plt.xticks(size=12)
plt.yticks(size=12)
red = np.where(y_train == 1)
blue = np.where(y_train == 0)
plt.scatter(x_train[red][:, 1], x_train[red][:, 2], c='red', label='y=1')
plt.scatter(x_train[blue][:, 1], x_train[blue][:,2], c='blue', label='y=0')
plt.legend()
plt.plot(xx, yy, c='green')
```

Listing 2: 绘图代码