MLlab1 实验报告

PB19030861 王湘峰

一、 实验要求

本次实验的总体流程是完成逻辑回归的代码实现，并在给定数据集上进行训练和验证/测试。具体来说需要完成以下部分：

· 读取训练数据集(training set)和测试数据集(testing set)

· (如有必要)对数据进行预处理

· 初始化逻辑回归模型

· 实现优化算法(推荐梯度下降或牛顿法，择一即可)

· 在训练数据集上进行模型的参数优化，要求该步骤在有限时间内停止（即具备收敛性）

· 在测试数据集上进行测试，并输出测试集中样本的预测结果（即具备准确性）

· 最后，完成实验报告

二、 实验原理

本实验关注线性模型在分类问题上的一个典型应用。

· 对数几率回归(Logistic Regression)

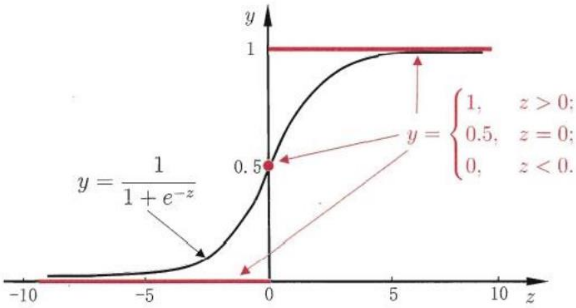
· 基本形式

$f(\boldsymbol{x}) = \frac{1}{1+e^{-(\boldsymbol{w^T}\boldsymbol{x}+b)}} \quad s.t. f(\boldsymbol{x}) \approx y$

其中$y \in \{0,1\}$，即逻辑回归是一个二分类模型

· 是广义线性模型的一个特例

$g(y) = ln\frac{y}{1-y} = \boldsymbol{w^T}\boldsymbol{x} + b$



优化目标

在分类中，我们可用最大似然法，最大化对数似然函数

- $l(\boldsymbol{w}, b) = \sum_{i=1}^{m} y_i \log\, P(y = 1|\boldsymbol{x_i}; \boldsymbol{w}, b) + (1 - y)\log P(y = 0|\boldsymbol{x_i}; \boldsymbol{w}, b)$
- $P(y = 1|\boldsymbol{x}; \boldsymbol{w}, b) = \frac{1}{1+e^{-(\boldsymbol{w}^{\boldsymbol{T}} x + b)}}$
- $P(y = 0|\boldsymbol{x}; \boldsymbol{w}, b) = 1 - P(y = 1|\boldsymbol{x}; \boldsymbol{w}, b)$

由于$l(\boldsymbol{w}, b)$是一个关于$(\boldsymbol{w}, b)$的高阶可导的连续凸函数，(得益于逻辑函数的数学性质)，可用经典的数值优化算法求全局最优解。

优化方法

负对数似然$l(\boldsymbol{w}) = \sum_{i=1}^{m}(-y_i \boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x_i} + \log\left(1 + e^{\boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x}_i}\right))$

一阶导数:

$$\nabla l(\boldsymbol{w}) = \sum_i (-y_i + \frac{1}{1 + e^{-\boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x}_i}}) x_i = -\sum_i (y_i - P(y = 1|\boldsymbol{x_i}; \boldsymbol{w}, b))\boldsymbol{x_i}$$

优化过程:

While 不满足终止条件:

$\boldsymbol{w} = \boldsymbol{w} - \alpha \cdot \nabla l(\boldsymbol{w})$

在本次实验中，我选择的终止条件更新前后目标函数的差值小于阈值。

(详情请见代码)

三、 核心代码讲解

```python
def grad(self, w, x, y):
    return ((y - self.sigmoid(x @ w)).T @ x).T
```

计算梯度的函数，$y,\boldsymbol{w}$为向量，x 为矩阵，返回值为$\boldsymbol{w}$的梯度

```python
def sigmoid(self, x):
    return 1.0 / (1.0 + np.exp(-x))
```

Sigmoid 函数

```python
def fit(self, train_x, train_y):

    w = np.ones((train_x.shape[1] + 1, 1))
    train_y = np.array(train_y).reshape(len(train_y), 1)

    for i in range(train_x.shape[1]):
        self.max.append(train_x.iloc[:, i].max())
        self.min.append(train_x.iloc[:, i].min())
        train_x.iloc[:, i] = (train_x.iloc[:, i] - train_x.iloc[:, i].min()) / (
                train_x.iloc[:, i].max() - train_x.iloc[:, i].min())
    train_x = np.array(train_x)
    train_x = np.c_[train_x, np.ones(train_x.shape[0])]
```

拟合函数 fit 的预处理部分，将训练集的每个属性归一化，同时将每个属性的最大最小值记录下来，在测试的时候对预测数据进行归一化。

p.s.预处理时将偏置b并入$\boldsymbol{w}$中，同时x中加入一列 1 向量。

```python
counter = 0
L1 = 0

for i in range(train_x.shape[0]):
    L1 += np.log2(1 + np.exp((np.dot(train_x[i], w)[0]))) - train_y[i] * (np.dot(train_x[i], w)[0])

while True:
    counter += 1
    dl = self.grad(w, train_x, train_y)
    w = w + self.alpha * dl

    L2 = 0
    for i in range(train_x.shape[0]):
        L2 += np.log2(1 + np.exp((np.dot(train_x[i], w)[0]))) - train_y[i] * (np.dot(train_x[i], w)[0])

    if abs(L2 - L1) < self.epsilon and counter > 50:
        break
    L1 = L2

self.w = w
```

训练部分(接上图)其中 L1、L2 都是目标函数值，分别代表每次梯度下降

前后目标函数的值。类内变量 epsilon 表示训练前后目标函数差的阈值。变量 counter 记录训练次数，在终止条件处有 counter>50 的条件，这是由于观察到有时可能出现某次更新(次数小于 50)前后目标函数变化较小，直接达到了终止条件，但实际上参数还处于欠拟合的状态。因此设置了 counter>50 的限制，经多次测试结果较为良好。

```python
def predict(self, test_x):

    for i in range(test_x.shape[1]):
        test_x.iloc[:, i] = (test_x.iloc[:, i] - self.min[i]) / (self.max[i] - self.min[i])
    test_x = np.array(test_x)
    test_x = np.c_[test_x, np.ones(test_x.shape[0])]

    pre = self.sigmoid(test_x @ self.w).flatten().tolist()
    for i in range(len(pre)):
        if pre[i] > 0.5:
            pre[i] = 1
        else:
            pre[i] = 0
    return pre
```

输出预测部分。同样的对 testX 进行预处理(归一化)，这里采用的是训练集每个属性的最大最小值。原因如下：

1. 训练时使用的数据是被**训练集**的最大最小值归一化的，从逻辑上来说，这里的最大最小值也是训练参数的一部分。对测试集归一化时采用测试集的最大最小值是没有牢固的**理论基础**的。
2. 从实际实现角度来说，使用训练集的最大最小值可以节省预测的时间开销。更重要的是，经过实际测试，在所有参数均不变的情况，使用训练集而非测试集进行归一化的预测准确率平均高了 10~20%!

```python
if __name__ == '__main__':
    df1 = pd.read_csv(sys.argv[1], header=None)
    df1.iloc[:, 1] = df1.iloc[:, 1].apply(lambda x: 1 if x == 'M' else 0)
    trainx = df1.iloc[:, 2:]
    trainy = df1.iloc[:, 1]

    df2 = pd.read_csv(sys.argv[2], header=None)
    testx = df2.iloc[:, 2:]

    LR = Logistic_Regression()
    LR.fit(trainx, trainy)
    pre = LR.predict(testx)

    for i in pre:
        if i == 0:
            print('B')
        else:
            print('M')
```

主函数部分(将标签处理为 0,1 数据)

四、 实验中的困难与解决

困难 1：矩阵运算中的对齐问题

我们知道，矩阵$A_{(a\times b)}$和矩阵$B_{(c\times d)}$可以做乘法 AB 的充要条件是$b=c$，在实际写代码中，很容易搞错矩阵的维数导致程序无法运行(例如：什么时候要转置？这个矩阵是左乘还是右乘？)

解决方案：

俗话说，好记性不如烂笔头。于是本人准备了一张 A4 纸，把代码中所有出现的矩阵的维数给记录了下来，从头到尾把代码改了改，结果一遍下来就给过了！我也因此获得了一个重要的经验。

困难 2：代码的收敛性

从数学上，我们已经证明了梯度下降法的收敛性，但是对于收敛时间却不能保证，如果参数设置的不好，需要训练上万次甚至十万次才能收

敛。

解决方案：起初使用的是梯度的 1 范数作为终止条件，但是不管怎么调参，算法要么需要很长时间收敛(不过准确率很高)，要么较快收敛，但准确率较低。后来考虑到计算机在 0 处附近的相对误差较大（因为理想条件下$\boldsymbol{w}$的梯度为 0 向量），将终止条件改为了目标函数训练前后的差值（本质上是对梯度的粗糙近似）。后来经过运行发现收敛相当的稳定了，一般几百次甚至一百多次就能收敛，再也没有出现过训练几万次的情况。但是此时却发现准确率有些偏低（80~90%），这正是下面要讨论的。

困难 3：在算法正确的前提下准确率偏低

解决方案：起初尝试通过调参来提高准确率，但是收效甚微，此时开始思考是不是算法的细节处理出现了问题。经过思考和查阅资料，发现对测试集的预处理是用的其本身的最大最小值归一化的。我们知道，归一化本质上是将原数据做线性变换；参数$\boldsymbol{w}$是通过训练集做线性变换之后拟合得到的，如果测试集用自己的最大最小值值归一化，实质上是使用了别的线性变换，这样直接使得$\boldsymbol{w}$失去了意义。

五、 实验结果的展示

· 准确率

将数据集 wdbc.data 按照 4:1 划分训练和测试集

（以下截图源自 jupyter notebook）

随机化划分数据集并计算准确率

```
raw = pd.read_csv('wdbc.data',header=None)
df1 = raw.sample(frac=0.8)
df2 = raw[raw.index.isin(df1.index) == False]

trainx = df1.iloc[:,2:]
trainy = df1.iloc[:,1].apply(lambda x:1 if x=='M' else 0)
testx = df2.iloc[:,2:]
testy = df2.iloc[:,1].apply(lambda x:1 if x=='M' else 0)
```

```
LR = Logistic_Regression()
LR.fit(trainx,trainy)
```

```
pre = LR.predict(testx)
out = []
for i in pre:
    if i == 0:
        out.append('B')
    else:
        out.append('M')
print(out)
print('correct rate:',evaluate(pre,testy)*100,'%')
```

```
['M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B',
'M', 'B', 'M', 'B', 'B', 'B', 'M', 'M', 'M', 'B', 'M', 'B', 'B', 'B', 'M', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'M', 'M', 'B',
'M', 'B', 'M', 'B', 'B', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'M', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M', 'B',
'B', 'B', 'B', 'M', 'M', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'B', 'M', 'B', 'M', 'M', 'M', 'B', 'B', 'M', 'B', 'B', 'B',
'M', 'B', 'B', 'B', 'B', 'B']
correct: 113 num: 114
correct rate: 99.12280701754386 %
```

可以看到，某次随机划分训练的准确率为 99%（有一个未分类正确）

随机化划分数据集并计算准确率

```
raw = pd.read_csv('wdbc.data',header=None)
df1 = raw.sample(frac=0.8)
df2 = raw[raw.index.isin(df1.index) == False]

trainx = df1.iloc[:,2:]
trainy = df1.iloc[:,1].apply(lambda x:1 if x=='M' else 0)
testx = df2.iloc[:,2:]
testy = df2.iloc[:,1].apply(lambda x:1 if x=='M' else 0)
```

```
LR = Logistic_Regression()
LR.fit(trainx,trainy)
```

```
pre = LR.predict(testx)
out = []
for i in pre:
    if i == 0:
        out.append('B')
    else:
        out.append('M')
print(out)
print('correct rate:',evaluate(pre,testy)*100,'%')
```

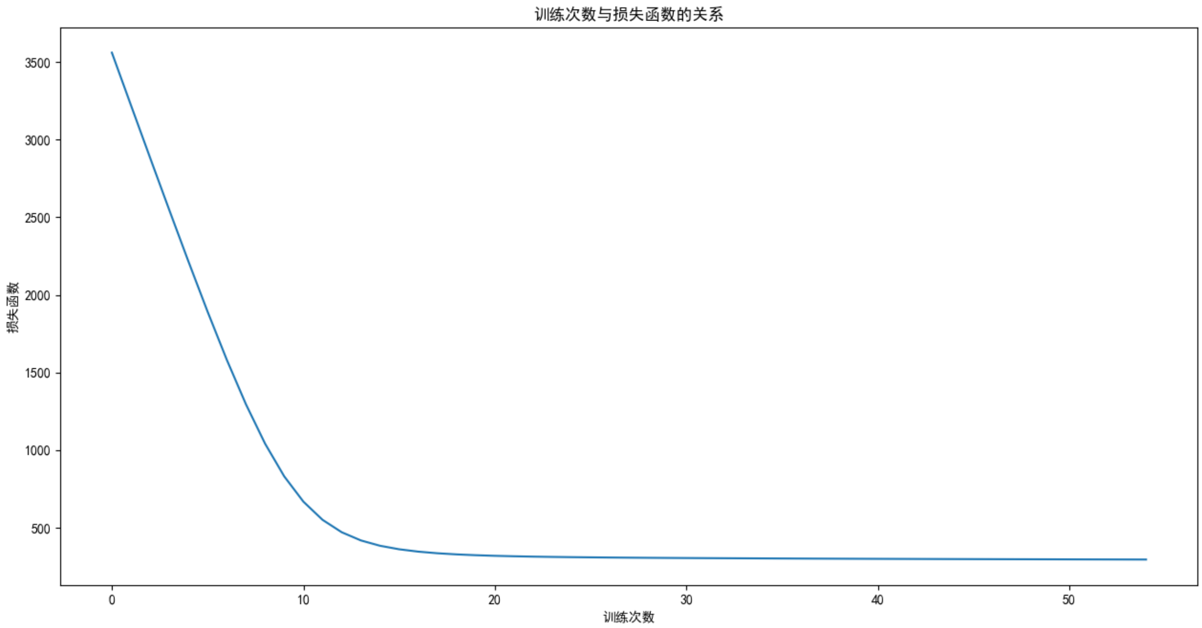
```
['M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'B', 'M', 'M', 'B', 'B', 'B', 'M', 'B', 'M', 'B', 'B', 'M', 'B', 'B', 'B', 'M', 'M', 'B', 'B',
'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'M', 'M', 'B', 'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M', 'M', 'M', 'M',
'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'M', 'B', 'M', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'M', 'M',
'B', 'M', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'B', 'M', 'B', 'M', 'M', 'B', 'B', 'B', 'B', 'B', 'M', 'B', 'M', 'B', 'B', 'B', 'B', 'B',
'B', 'B', 'B', 'B', 'M', 'M']
correct: 110 num: 114
correct rate: 96.49122807017544 %
```

另一次的随机划分训练的准确率为 96%

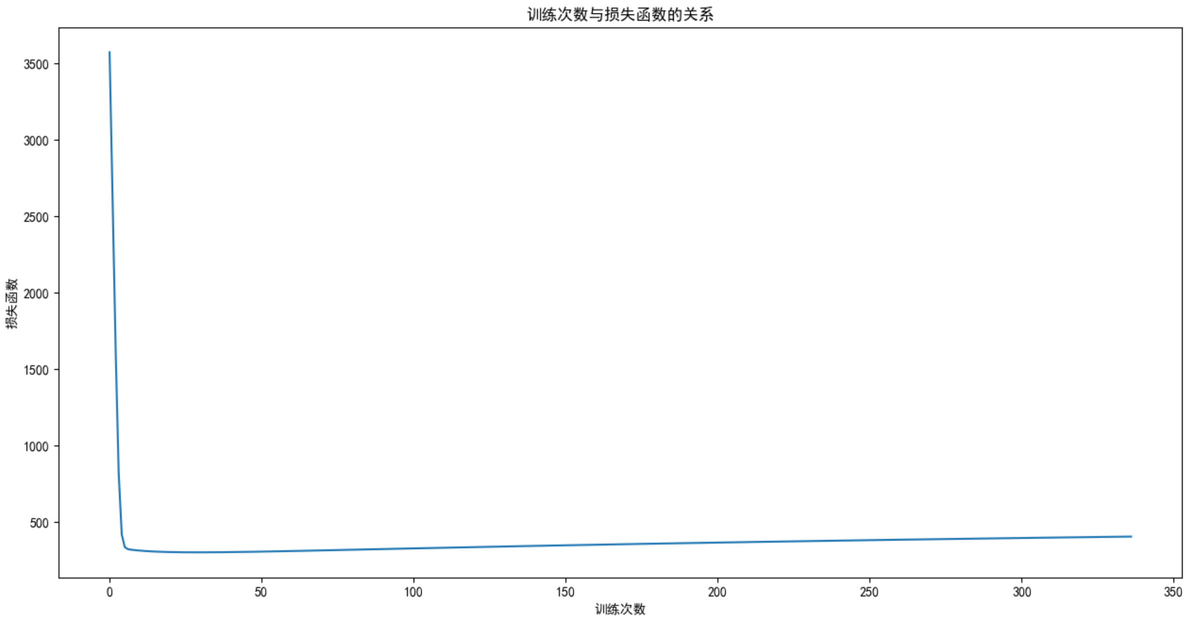
· **数据可视化**

下图为某次随机划分训练时损失函数的变化（使用了 matplotlib 库）

经过对程序多轮的优化，目前代码基本上可以在几秒之内完成训练，并且输出



另一次的损失曲线



p.s.出现图二可能是阈值设置过小，不过好在对预测结果没有负提升。

· 总结

经过对程序多轮的优化，目前代码基本上可以在几秒之内完成训练，并且输出

的预测的准确率能在 95%左右。