机器学习Lab1 逻辑回归

- PB20111635
- 蒋磊

1、实验要求

本次试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测试。

具体需要实现这些步骤：

- 训练集和测试集的读取
- 对数据进行预处理
- 初始化逻辑回归模型
- 划分训练集与测试集
- 用建立的逻辑回归模型对训练集进行训练，并绘制损失函数
- 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，通过采用不同的参数（如学习率，正则化等）比较它们对应的准确性

2、实验原理 逻辑回归

逻辑回归的基本形式为：

$$f(x)=\frac{1}{1+e^{-(\omega^Tx+b)}}\ \ s.t\ f(x)\approx y \tag{13}$$

其中 $y\in\{0,1\}$ ，故可用该方法实现一个二分类模型。

可以看出逻辑回归是广义线性模型的一个特例，其中

$$g(y)=ln\frac{y}{1-y} \tag{14}$$

可以用极大似然法优化参数，最大化对数似然

$$l(\omega,b)=\sum_{i=1}^{m}y_i logP(y=1|x_i;\omega,b)+(1-y_i)logP(y=0|x_i;\omega,b)=\sum_{i=1}^{m}logP(y=y_i|x_i;\omega,b) \quad (15)$$

其中，用 $y=\frac{1}{1+e^{-(\omega^T x+b)}}$ 来估计 $P(y=1|x_i;\omega,b)$ ，则有：

$$\begin{aligned} P(y=1|x_i;\omega,b)&=\frac{1}{1+e^{-(\omega^T x+b)}} \\ P(y=0|x_i;\omega,b)&=\frac{e^{-(\omega^T x+b)}}{1+e^{-(\omega^T x+b)}} \end{aligned} \quad (16)$$

并通过再x 后加入一个数值为1的特征，可将 b 加入到 ω 之中将$l(\omega,b)$ 转化为$l(\omega)$，再将上面两式带入 $l(\omega)$ ，最大化对数似然可以转化为最小化负对数似然：

$$l(\omega)=-\sum_{i=1}^{m}log\frac{e^{y_i w^T x_i}}{1+e^{w^T x_i}}=\sum_{i=1}^{m}-y_i\omega^T x_i+log(1+e^{w^T x_i}) \quad (17)$$

考察 $f(x)=-\alpha x+ln(1+e^x)$ ，可知 $f''(x)>0$，故$f(x)$ 是凸函数。

又 $l(\omega)$ 是 $f(x)$ 与 $g(\omega)=\omega^T x_i$ 的复合，故 $l(\omega)$ 是关于 ω 的凸函数。所以可以使用梯度下降法求解。

$$\begin{aligned} \nabla l(\omega)&=\sum_i f'(z_i)\frac{\partial z_i}{\partial\omega}=\sum_i(-y_i+\frac{1}{1+e^{-z_i}})x_i=-\sum_i(y_i-P(y=1|x_i;\omega))x_i \\ \nabla^2 l(\omega)&=\frac{\partial\nabla l(w)}{\partial w^T}=\sum_i p_1(1-p_1)x_i x_i^T \quad (p_1=P(y=1|x_i;\omega)) \end{aligned} \quad (18)$$

梯度下降法：

```
while ‖∇l(w)‖>δ do
    w = w - α∇l(w)
end while
```

由凸函数的性质可知，最终能求得最优解。

3、核心代码

3.1 定义需要使用的数学函数

```
def sigmoid(self, x):
"""The logistic sigmoid function"""
        return 1/(1 + np.exp(-x))

def loss(self, y_r, y_p):
        return y_r*np.log(y_p) + (1 - y_r)*np.log(1 - y_p )

```

```
    """
    负对数似然函数 loss_f，np.nan_to_num()函数是为了防止出现log0的情况，
    最后的loss_f还除以了一个训练数据总数即y_r.shape[0]
    """
def loss_f(self, y_r, y_p):
    return -np.sum(np.nan_to_num(self.loss(y_r, y_p))) / y_r.shape[0]

# sign函数是一个阶跃函数，将小于0.5的数映射到0，大于0.5的数映射到1
def sign(self, y):
    for i in range(y.shape[0]):
        if y[i]<0.5:
            y[i] = 0
        else:
            y[i] = 1
    return y

# get_acc用来比较测试结果y_p与实际结果y_r是否相等，然后返回正确率
def get_acc(self, y_r, y_p):
    y_p = self.sign(y_p)
    s = 0
    for i in range(y_r.shape[0]):
        if y_r[i] == y_p[i]:
            s += 1
    return s/y_r.shape[0]
```

3.2 根据助教老师提供的代码框架进行完善

3.2.1 训练过程

```
  def fit(self, X, y, lr=0.01, tol=1e-7, max_iter=1e7):
        """
        Fit the regression coefficients via gradient descent or other methods
        """
        # 首先对X进行标准化
        X = np.array(X)
        n,m = X.shape
        y = np.array(y)
        mu = np.mean(X, 0)      # 列的均值
        sigma = np.std(X, 0)    # 列的标准差
        X = (X - mu)/sigma
        # add ones
        X = np.c_[X, np.ones((n,1))]
        # w
```

```
        w = np.random.randn(m+1)    # 标准正态的随机浮点矩阵
        loss = []
        iters = []
        # gradient descent
        for i in range(max_iter):
            y_p = self.sigmoid(np.dot(w, np.transpose(X)))
            if i%1000 == 0:
                print(f'itr=={i}, loss=={self.loss_f(y, y_p)}')
            loss.append(self.loss_f(y, y_p))
            iters.append(i)
            dw = np.dot((y_p - y), X)/n # 求dw
            w = w - lr*dw               # 梯度下降 lr为学习率
        y_p = self.sigmoid(np.dot(w, np.transpose(X)))
        print(f'finally loss:{self.loss_f(y, y_p)}')
        plt.plot(iters, loss)
        return w, mu, sigma
```

fit函数需要传入4个参数：X：特征矩阵(shape = (n,12))，y：X对应的标签(shape = (n,1))，itr：迭代次数，lr：学习率。注意传入的X是没有进行加一列1向量的处理的，所以要在函数中进行处理。

首先对X数据进行一些预处理，**预处理的方式是进行标准化**，标准化的方法是对X的每一个属性减去它的均值除以它的方差，其中均值与方差是要保留下来传递给后面的测试函数的，即用训练集的分布估计整体的数据分布，并在测试集数据中进行同样的标准化处理。

之后如上所说加入一列1向量并初始化w，用梯度下降得到最终参数w。

注意其中 $dw = np.dot((y_p - y), X)/n$ 是除以了一个n的，这是因为我们设的loss_f除以了一个n，并不影响梯度下降的结果但能有效的避免数值计算上出现的一些问题，使用标准化的预处理也是为了这个目的。

3.3.2 得到测试集结果并输出

```
    def predict(self, X, y, w, mu, sigma):
        """
        Use the trained model to generate prediction probabilities on a new
        collection of data points.
        """
        """
        函数非常简单，先对测试集数据进行与训练集相同的预处理，
        进行标准化并加一列1向量,再通过fit得到的参数得到对数机率，
        并用sign函数得到最终要输出的估计结果
        """
        # 得到测试结果并输出
        X = np.array(X)
        n,m = X.shape
        X = (X - mu)/sigma
```

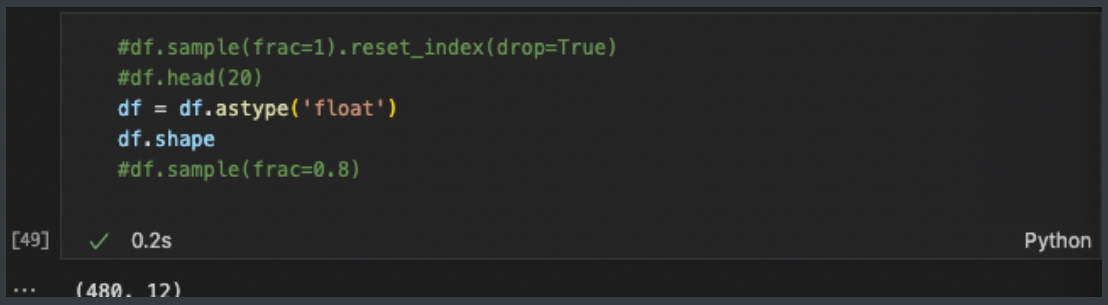
```
        # add ones
        X = np.c_[X, np.ones((n,1))]
        y = np.array(y)
        y_p = self.sigmoid(np.dot(w, np.transpose(X)))
        acc = self.get_acc(y,y_p)
        print(f'acc = {acc}')   # 这里打印的是对测试集预测的准确率
        return y_p
```

函数非常简单，先对测试集数据进行与训练集相同的预处理，进行标准化并加一列1向量，再通过fit函数得到的参数得到对数机率，并用sign函数得到最终要输出的估计结果。

4、实验结果

4.1 表格数据处理

```
# Task1 deal with NULL rows, you can either choose to drop them or replace them with mean or other value
df.drop("Loan_ID", axis=1, inplace=True)
# Checking the Missing Values
df.isnull().sum()
# 直接舍弃掉有nan的行
df = df.dropna()
df.shape
```



可以看到，直接舍弃掉有nan的行之后，还剩下480条有效数据。

```
# Task2 deal with categorical features
# Tip df.Gender=df.Gender.map({'Male':1,'Female':0})
df.Gender = df.Gender.map({'Male':1,'Female':0})
df.Married = df.Married.map({'Yes':1, 'No':0})
df.Education = df.Education.map({'Graduate':1, 'Not Graduate':0})
df.Self_Employed = df.Self_Employed.map({'Yes':1, 'No':0})
df.Property_Area = df.Property_Area.map({'Urban':3, 'Rural':1, 'Semiurban':2})
df.Loan_Status = df.Loan_Status.map({'Y':1, 'N':0})
df.Dependents = df.Dependents.map({'3+':3, '2':2, '1':1, '0':0 })
```

接下来是对表格中的字符串进行处理，将它们转化为对应的数，这里需要注意的是表中第3列数据是字符串。

4.2 划分训练集与测试集

```
# Task3 split the dataset into X_train, X_test, y_train, y_test
# Optional: you can also use normalization
df = df.iloc[np.random.permutation(len(df))]

X_train = df.iloc[0:400, 0:10]
y_train = df.iloc[0:400, 11]

x_test = df.iloc[401: , 0:10]
y_test = df.iloc[401: , 11]
```

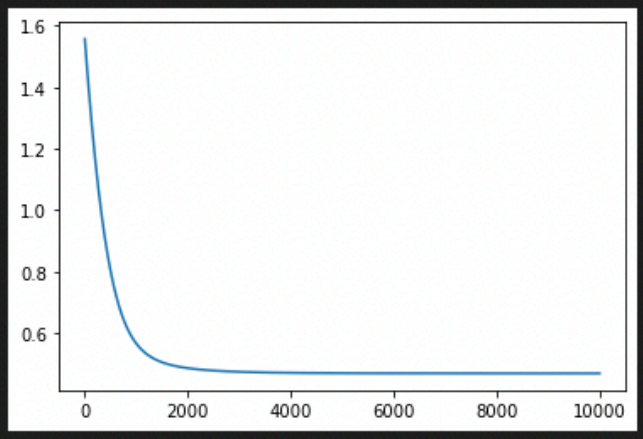
这里为了简单起见，大致按照训练集占80%，测试集占20%的比例进行划分，把前400行数据作为训练集，把后80行数据作为测试集。

4.3 训练过程

```
import matplotlib.pyplot as plt

lr = 0.01
tol = 1e-7
max_iter=1e4

def train(X, y, lr=0.01, tol=1e-7, max_iter=1e4):
        """
        Fit the regression coefficients via gradient descent or other methods
        """
        # 首先对X进行标准化
        X = np.array(X)
        n,m = X.shape
        y = np.array(y)
        mu = np.mean(X, axis=0)     # 列的均值
```

```
        sigma = np.std(X, axis=0)   # 列的标准差
        X = (X - mu)/sigma
        # add ones
        X = np.c_[X, np.ones((n,1))]
        # int w
        w = np.random.randn(m+1)    # 标准正态的随机浮点矩阵

        loss = []
        iters = []
        # gradient descent
        for i in range(int(max_iter)):
            y_p = logis.sigmoid(np.dot(np.transpose(w), np.transpose(X)))
            if i%1000 == 0:
                print(f'itr=={i}, loss=={logis.loss_f(y, y_p)}')
            loss.append(logis.loss_f(y, y_p))
            iters.append(i)
            dw = np.dot((y_p - y), X)/n # 求dw
            w = w - lr*dw               # 梯度下降 lr为学习率
        y_p = logis.sigmoid(np.dot(w, np.transpose(X)))
        print(f'finally loss:{logis.loss_f(y, y_p)}')
        plt.plot(iters, loss)
        return w, mu, sigma

# Task4 train your model and plot the loss curve of training
w, mu, sigma = train(X_train, y_train, lr, tol, max_iter)
```

```
itr==0, loss==1.5559617344490195
itr==1000, loss==0.5657437615644377
itr==2000, loss==0.4858842524509723
itr==3000, loss==0.47396625317832
itr==4000, loss==0.47061756251984377
itr==5000, loss==0.4695379982659959
itr==6000, loss==0.46917193843555255
itr==7000, loss==0.4690444158215432
itr==8000, loss==0.46899933044670306
itr==9000, loss==0.4689832687625304
finally loss:0.46897752516296903
```



可以看到，在迭代大约10000次之后模型就趋于收敛了，这里我取的学习率为0.01

4.3 测试结果

```
1 # Task5 compare the accuracy(or other metrics you want) of test data with different
  parameters you train with
2 y_p = logis.predict(x_test, y_test, w, mu, sigma)
3 print(y_p)
```

```
# Task5 compare the accuracy(or other metrics you want) of test data with differ
y_p = logis.predict(x_test, y_test, w, mu, sigma)
print(y_p)
```

[52] ✓ ✓ 0.2s Python

```
acc = 0.7974683544303798
[1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 0. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 0. 1.
 1. 1. 1. 0. 1. 1. 0. 0. 1. 1. 1. 1. 1. 0. 1. 1. 0. 1. 0. 1. 0. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1.
 1. 1. 1. 1. 1. 1. 0.]
```

```
# Task5 compare the accuracy(or other metrics you want) of test data with differ
y_p = logis.predict(x_test, y_test, w, mu, sigma)
print(y_p)
```

[36] ✓ 0.2s Python

```
acc = 0.810126582278481
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 0. 1.
 1. 1. 1. 0. 1. 1. 0. 0. 1. 1. 1. 1. 1. 0. 1. 1. 0. 1. 0. 1. 0. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1.
 1. 1. 1. 1. 1. 1. 0.]
```

```
# Task5 compare the accuracy(or other metrics you want) of test data with differ
y_p = logis.predict(x_test, y_test, w, mu, sigma)
print(y_p)
```

[340] ✓ 0.2s Python

```
acc = 0.8227848101265823
[0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 0. 1. 1. 1. 0. 1. 0. 1. 1.
 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 0. 1.
 0. 1. 0. 1. 1. 0. 1. 1. 0. 0. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1.]
```

这里我跑了大约10次，得到的正确率范围大概在79%～82%(去掉了最低值和最高值），可以看到测试正确率大约为80%

4.4 通过修改参数对模型进行优化

4.4.1 通过增大训练集来提高准确率

1、只增加10条数据

```
df = df.iloc[np.random.permutation(len(df))]
X_train = df.iloc[0:410, 0:10]
y_train = df.iloc[0:410, 11]
x_test = df.iloc[411: , 0:10]
y_test = df.iloc[411: , 11]
```

只增加10个数据进入训练集，进行测试（这里仍然是跑10次取了平均值）：

```
# Task5 compare the accuracy(or other metrics you want) of test data with differ
y_p = logis.predict(x_test, y_test, w, mu, sigma)
print(y_p)
```

[171] ✓ ✓ 0.3s Python

```
acc = 0.8260869565217391
[1. 0. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 0. 1. 1. 0. 0. 1. 1.
 1. 1. 1. 0. 1. 1. 0. 1. 0. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1.
 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 0.]
```

其余代码不变，重新训练，通过测试，效果不明显，大概只有不到1%的准确率提升，可能是因为增加的训练集数据不够多，下面再适当增大一些训练集大小。

2、增加20条数据

```
df = df.iloc[np.random.permutation(len(df))]
X_train = df.iloc[0:420, 0:10]
y_train = df.iloc[0:420, 11]
x_test = df.iloc[421: , 0:10]
y_test = df.iloc[421: , 11]
```

再稍微扩大训练集的大小，大致按照训练集87%，测试集13%进行划分，仍然按照测试10次取平均值的方法：

```
# Task5 compare the accuracy(or other metrics you want) of test data with differ
y_p = logis.predict(x_test, y_test, w, mu, sigma)
print(y_p)
```

[162] ✓ 0.7s Python

```
acc = 0.8305084745762712
[1. 1. 0. 1. 1. 1. 1. 0. 1. 1. 0. 0. 1. 1. 1. 1. 1. 0. 1. 1. 0. 1. 0. 1.
 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 0.]
```

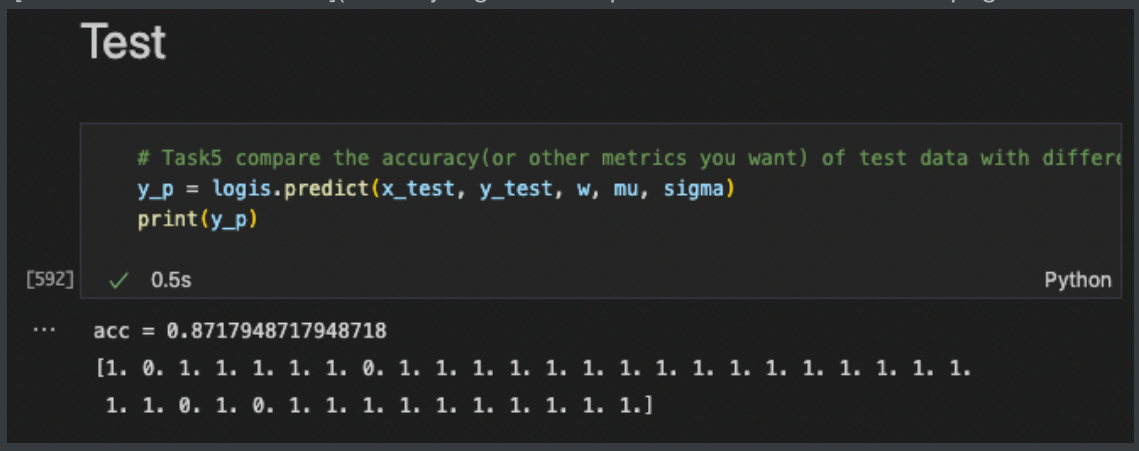
通过测试，准确率有所提升，但效果仍然不明显，大概只有2%。

3、增加40条数据

```
df = df.iloc[np.random.permutation(len(df))]
X_train = df.iloc[0:440, 0:10]
y_train = df.iloc[0:440, 11]
x_test = df.iloc[441: , 0:10]
y_test = df.iloc[441: , 11]
```

再继续扩大训练集的大小，大致按照训练集90%，测试集10%进行划分：

![截屏2022-10-14 20.47.38](/Users/jianglei/Desktop/截屏2022-10-14 20.47.38.png



可以看到，准确率大概提升了5%，来到了87%。

到这里，可以看出，适当增大训练集大小貌似可以有效提高测试准确率，但我不打算继续增大训练集的大小了，因为此时测试集已经很小了，继续增大训练集的意义不大。

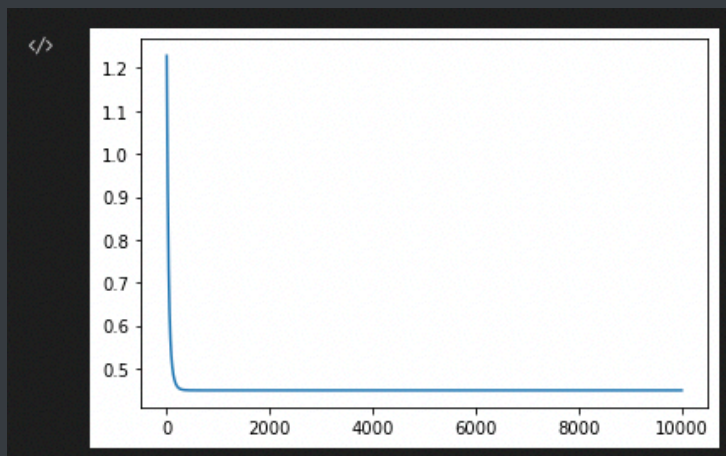
不过可能是因为此次实验给的数据量太小，训练集大小受限制，所以效果很不明显。

4.4.2 通过改变学习率进行优化

初次训练我才用的学习率为0.01，得到的测试准确率大约为80%，接下来我将采取不同的学习率，并比较不同学习率对测试准确率的影响。

1、学习率为0.1

```
1  lr = 0.1
```



Test

```
# Task5 compare the accuracy(or other metrics you want) of test data with d
y_p = logis.predict(x_test, y_test, w, mu, sigma)
print(y_p)
```

[178] ✓ 0.3s

```
acc = 0.7468354430379747
[0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 0. 1. 0. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 0. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1.]
```

通过多次测试取平均值，最后发现正确率在75%左右，比学习率为0.01时大概减小了5%

2、学习率为0.2

```
# Task5 compare the accuracy(or other metrics you want) of test data with differ
y_p = logis.predict(x_test, y_test, w, mu, sigma)
print(y_p)
```

[889] ✓ 0.2s Python

```
acc = 0.8354430379746836
[1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 1. 1. 1. 1. 1.
 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 0. 1. 1. 1. 1. 1. 0. 1.
 1. 1. 1. 0. 0. 1. 1. 1. 1. 1. 1. 1. 0. 1. 0. 1. 1. 1. 1. 0. 1. 1. 0. 1.
 1. 0. 1. 1. 1. 1. 1.]
```

通过多次测试取平均值，最后发现正确率在83%左右，比学习率为0.01时大概提高了1%

不过总体来说，当学习率在0.01～0.2的范围内时，我在此次实验并没有感受到正确率有太大变化。