逻辑回归与感知机的实现

17341146 王程钥

1 算法原理

1.1 感知机学习算法

感知机(PLA)是一个分类学习算法，它的输入是实例的特征向量X ∈ R^{N*D}，其中 N 是实例个数，D 是特征维数。感知机学习是个硬分类模型，它的输出是预测结果 Y，可能是+1 或 -1，其中+1 表示为正例，-1 表示为负例。感知机学习算法需要学习一个 D 维向量 w 和一个偏置值 b，使用 sgn 函数作为激活函数，通过(1)式得到每个实例的预测结果。

将(1)式对 w 求偏导数可以得到(2)式，即感知机学习损失函数的梯度。由此可以推出感知机学习的迭代方式。

感知机学习算法每次迭代时在训练集的 N 个实例中选择一个误分类的实例 X1，并根据(3)式和(4)式更新权重（其中γ表示学习率）。若将每个实例的特征向量加上一维 1，将偏置项放入 w 中，则只需要使用(3)式即可完成更新。

$$\check{y} = \mathrm{f(x)} = \mathrm{sgn}(w^T * \mathrm{x} + \mathrm{b}) \qquad (1)$$

$$\frac{\partial \mathrm{f(x)}}{\partial \mathrm{w}} = \mathrm{y} * \mathrm{x} \qquad (2)$$

$$w^{t+1} = w^t + \gamma y_i x_i \qquad (3)$$

$$b^{t+1} = b^t + \gamma x_i \qquad (4)$$

1.2 逻辑回归算法

逻辑回归算法(PLA)是一个分类学习算法。和感知机算法相同，它的输入是实例的特征向量X ∈ R^{N*D}，其中 N 是实例个数，D 是特征维数。逻辑回归是一个软分类模型，对于每个实例 x，模型的输出结果是实例 x 属于每个类别 y 的概率分布。我们根据概率大小判断实例的类别。

逻辑回归学习算法需要学习一个 D 维向量 w 和一个偏置值 b。每个实例 x 通过 w 和 b 会获得一个得分，与感知机使用 sgn 函数不同，逻辑回归使用了 sigmod 函数作为激活函数，将 w*x+b 得分的范围从[−∞,∞]映射到[0,1]，并将这个预测值作为分类结果为正的概率。我们可以得到逻辑回归的输出函数(5)。

逻辑回归使用交叉熵(6)作为损失函数，交叉熵函数也等价于最大似然估计取负。我们将损失函数对 w 求偏导，即可得到逻辑回归算法的梯度(7)。与感知机相同，将偏置维加入特征向量中，按照(8)式更新答案即可。

$$\mathrm{f(x)} = \frac{1}{1 + e^{w^T x + b}} \qquad (5)$$

$$\mathrm{Loss} = -\sum_{i=1}^{n} (y_i * \log(f(x_i))) + ((1 - y_i) * \log(1 - f(x_i))) \qquad (6)$$

$$\frac{\partial \mathrm{L}(w_i)}{\partial w_i} = -\sum_{n=1}^{N} \big(y_n - f(x_n)\big) * x_{n,i} \qquad (7)$$

$$w^{t+1} = w^t - \gamma \sum_{n=1}^{N} \left(y_n - \frac{1}{1 + e^{w^{tT} x_n + b}} \right) * x_n \qquad (8)$$

2 算法流程

Algorithm1 PLA.fit

```
Input: 训练集 Train_x, Train_y
Output: 模型 weight, bias
for i in range(iter_size) do:
    for x,y in train_data do:
        if 分类错误 calc(x)!=y do:
            根据梯度更新权值
            weight += lr*y*x
            bias += lr*y
            break
    if 当前参数更优 test(x, y, weight, bias) > max_accuracy do
        保存参数 save_model(weight, bias)
return weight, bias
```

算法 1 为感知机学习的训练算法，输入训练集和测试集，返回模型的权重。

Algorithm2 LR.fit

```
Input: 训练集 Train_x, Train_y
Output: 模型 weight, bias
for i in range(iter_size) do:
    计算梯度 det_w, det_b = calc_grads(train_x, train_y)
    weight = weight-lr*det_w
    bias = bias-lr*det_b
    根据梯度更新权值
    if 收敛 det_w<eps and det_b<eps do break
return weight, bias
```

算法 2 为逻辑回归的训练算法，这里使用批梯度下降的方法对损失函数进行最优化，输入训练集和测试集，返回模型的权重。

3 代码解析

本次实验中需要实现感知机学习和逻辑回归两种算法，我在对于这两个模型各封装了一个类，基于这两个类进行操作。

3.1 PLA 类

```
def __init__(self,num_class,lr=1,no_bias=False,iter_method='whole'):
    self.num_class=num_class
    self.lr=lr
    self.weight=np.zeros(num_class)
    self.saved_weight=np.zeros(num_class)
    self.bias=0
```

```
    self.saved_bias=0
    self.no_bias=no_bias
    self.pos=0
    self.iter_method=iter_method
```

PLA 类初始化

```
def calc(self, x, type='save'):
    if type=='current':
        result=np.dot(x, self.weight)
        if not self.no_bias: result+=self.bias
    elif type=='save':
        result=np.dot(x, self.saved_weight)
        if not self.no_bias: result+=self.saved_bias
    else: assert(0)
    return np.sign(result)
```

计算预测结果。Current 类计算当前参数的结果，save 类计算最优参数的结果。

```
def fit(self, train_x, train_y, iter_size=50000):
    length = len(train_x)
    self.reset_weight()
    max_accuracy = 0
    for i in range(iter_size):
        cnt_correct = 0
        if self.iter_method=='first': self.pos=0
        while cnt_correct<length:
            x = train_x[self.pos]
            y = train_y[self.pos]
            self.pos = (self.pos+1)%length
            if y==0: y=-1
            if self.calc(x, type='current')!=y:
                self.weight += self.lr*y*x
                self.bias += self.lr*y
                break
            # 更新权重
            else: cnt_correct+=1
        if i%10==0:
            pred = self.predict(train_x,type='current')
            accuracy = self.evaluate(train_y, pred)
            if accuracy> max_accuracy:
                max_accuracy = accuracy
                self.save_weight()
            # 根据训练集准确率存储最优参数
#训练
```

训练感知机模型

3.2 PLA 类

```
def __init__(self, num_class, lr=1, optimizer='Adam',
             no_bias=False, beta1=0.9, beta2=0.999, epislon=1e-8):
        assert(optimizer=='BGD' or
               optimizer=='MBGD' or optimizer=='Adam')
    self.num_class=num_class
    self.lr=lr
    self.weight=np.zeros(num_class)
    self.bias=0
    self.no_bias=no_bias
    self.optimizer=optimizer
    # 初始化参数
    self.beta1=beta1
    self.beta2=beta2
    self.epislon=epislon
    self.m_weight=np.zeros(num_class)
    self.v_weight=np.zeros(num_class)
    self.m_bias=0
    self.v_bias=0
    self.t=0
    # Adam 优化器参数
```

PLA 类初始化

```
def calc(self, x):
    result=np.dot(x, self.weight)
    if not self.no_bias: result+=self.bias
    return 1/(1+np.exp(result))
```

计算预测结果。

```
def calc_grads(self, x, y):
    val = y-self.calc(x)
    tmp_x = x.transpose()
    det_w = np.sum(val*tmp_x,axis=1)
    if self.no_bias: det_b=0
    else: det_b = sum(val)
    det_w /= len(x)
    det_b /= len(x)
    return det_w, det_b
```

计算交叉熵的梯度。

```
def fit(self, train_x, train_y, iter_size=5000, batch_size=64):
    self.reset_weight()
    length = len(train_x)

    if self.optimizer=='BGD': batch_size=length
    else: batch_size = min(batch_size,length)
    p_list = np.linspace(0,length,num=length/batch_size,
```

```
                    endpoint=False).astype(int)

    eps = 1e-8; pre_loss=1e18
    for i in range(iter_size):
        for j in p_list:
            R = min(j+batch_size, length)
            det_w, det_b=self.calc_grads(train_x[j:R], train_y[j:R])

            if self.optimizer=='Adam':
                self.t += 1
                lr = self.lr*(1-self.beta2 ** self.t) ** 0.5
                     / (1-self.beta1 ** self.t)
                self.m_weight = self.beta1*self.m_weight
                                + (1-self.beta1)*det_w
                self.v_weight = self.beta2*self.v_weight
                                + (1-self.beta2)*(det_w * det_w)
                self.weight -= lr*self.m_weight
                               /(self.v_weight ** 0.5 + self.epislon)

                if not self.no_bias:
                    self.m_bias = self.beta1*self.m_bias
                                  + (1-self.beta1)*det_b
                    self.v_bias = self.beta2*self.v_bias
                                  + (1-self.beta2)*(det_b*det_b)
                    self.bias -= lr*self.m_bias
                                  /(self.v_bias ** 0.5+self.epislon)
                # Adam 优化
            else:
                self.weight -= self.lr * det_w
                if not self.no_bias: self.bias -= self.lr * det_b

        if i%50==0:
            loss=self.cross_entropy(train_x, train_y)
            if pre_loss-loss<eps:
                if self.lr>eps: self.lr*=0.1
                else: break
            pre_loss=loss
        # 判断收敛，动态更新学习率
```

训练逻辑回归模型

4 优化点与部分说明

本次实验我采取的 5 折交叉验证的方法，即将训练集分成 5 份，每次枚举一份作为验证集，其余四份作为训练集。先使用 PLA 或 LR 模型对训练集进行拟合，后使用上述模型对

测试集进行预测并计算准确率。将 5 折交叉验证的准确率取平均作为对训练结果的评估。

因为当 batch 较大时梯度值也较大，为了让梯度不受 batch 大小的影响，我将梯度大小先除以 batch 大小后用于更新。

4.1 PLA 算法的迭代次序

ppt 中的 PLA 算法每次从头开始找到第一个分类错误的点，修改权重。经试验发现，由于数据集并不线性可分，采用这种方式基本只能跑到前几十个样本，后面的样本都会被忽略。因此我采用了另一种遍历方法，每次从上一次失败的样本后一个样本开始遍历。

4.2 非线性可分的处理

5.1 节的实验结果表明本次实验的数据集是非线性可分的。为了处理这种情况，我针对 PLA 算法和 LR 算法分别提出了不同的方案。

4.2.1 感知机学习

对于感知机学习算法，我会迭代一定次数，并在训练集上对参数的准确率进行测试，从中选择准确率最高的参数存储下来作为测试参数。

4.2.2 逻辑回归

因为逻辑回归使用交叉熵，交叉熵是一个凸函数，所以它不存在收敛到鞍点和局部最优点的问题。只需要提供一种收敛判定方法即可。如果两次迭代的交叉熵的差小于一个指定参数 eps，或迭代后的交叉熵大于迭代前，则认为模型在当前学习率下已经收敛到的最低点，我会降低学习率继续优化。如果迭代超过一定限度，或学习率降低到一定数值，则认为完全收敛，结束训练。

4.3 梯度下降算法的优化

使用批梯度下降算法，在合适的学习率下可以很好地收敛到最优解。不过它收敛速度较慢，且每次迭代需要计算所有样本，内存开销较大。

常见的思路是使用小批量梯度下降。它每次选择若干个样本（一般为 50-200 个）进行更新。使用这种算法可以提升收敛速度，减少内存开销。

近年提出的 Adam 优化算法可以很快地完成收敛，不过这个算法有时不能收敛到最优解。Adam 优化算法基于训练的历史梯度和梯度平方，动态调整学习率。Adam 算法定义了 m 和 v 两个变量分别表示梯度和梯度平方的加权平均值，它通过梯度值g_t进行更新。根据原始论文，默认$\beta_1 = 0.9$，$\beta_2 = 0.99$，$\varepsilon = 1e-8$，更新公式如下。

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \qquad (9)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) {g_t}^2 \qquad (10)$$

$$\mathrm{lr} = \mathrm{init\_lr} * \frac{1-{\beta_2}^t}{1-{\beta_1}^t} \qquad (11)$$

$$w_{t+1} = w_t - lr * \frac{m_t}{\sqrt{v_t} + \varepsilon} \qquad (12)$$

使用 Adam 优化算法或 MBGD 算法可能无法收敛到最优解，因此我采用了动态调整学习率的方法。每次优化的到收敛后将学习率乘以 0.1，降低学习率继续优化直到完成 20000 个 epoch 的迭代。5.2 节表明，这种办法效果显著。

5 实验结果

5.1 节及表 1，表 2 的测试结果均采用 5 折交叉验证，5.2 和 5.3 节的结果则使用第一

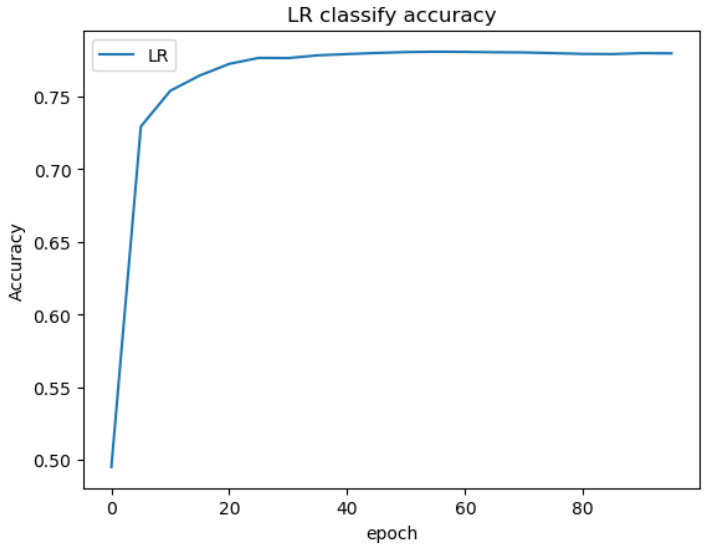
折作为验证集，后 4 折作为训练集进行试验。

5.1 算法正确性分析

对于 PLA 算法，根据 ppt 要求初始化零向量，每次迭代按顺序从第一个样例开始找下一个错误的样例，学习率为 1，迭代 20000 次，保存在训练集上准确率最高的权重，进行 5 折交叉验证，最终在验证集上获得 69.39%的平均准确率。

对于 LR 算法，根据 ppt 要求初始化零向量，使用批梯度下降算法优化，学习率为 1e-5，迭代 20000 次，通过 5 折交叉验证，最终在验证集上获得 77.83%的平均准确率。图 1 为迭代前 100 个 epoch 在验证集上的准确率，可以看到当迭代次数超过 60 次后有轻微的过拟合现象，总体准确率稳定在 78%左右。

作为对比，我调用了 sklearn 库的 PLA 算法，LR 算法和线性核的 SVM。表 1,2 为以上实验的结果，可以看到使用线性模型分类的最高准确率基本在 76%-78%左右，78%应该是线性模型在本数据集上的极限。我简单地调用了 rbf 核（非线性）的 SVM 分类器，可以看到训练集上的准确率得到了很大的提升，我相信如果进行调参，验证集的准确率也会得到提升。由此可见线性模型的局限性。



（图 1）

方法	数据集	Fold1	Fold2	Fold3	Fold4	Fold5	Avg
PLA+首误遍历	训练集	65.78%	70.36%	70.59%	70.38%	70.34%	69.49%
	验证集	66.18%	70.69%	69.75%	70.62%	69.69%	69.39%
PLA+全局遍历	训练集	76.80%	76.81%	76.92%	76.56%	77.23%	76.87%
	验证集	**76.50%**	**77.06%**	**77.50%**	**77.00%**	**75.43%**	**76.70%**
Sklearn_PLA	训练集	70.89%	76.50%	73.37%	72.80%	51.09%	68.93%
	验证集	71.12%	76.31%	71.56%	72.62%	50.81%	68.49%

（表 1 感知机学习算法的准确率）

方法	数据集	Fold1	Fold2	Fold3	Fold4	Fold5	Avg
LR+BGD	训练集	78.05%	78.34%	78.48%	77.98%	78.38%	78.25%
	验证集	78.13%	**78.19%**	**77.31%**	78.00%	77.50%	77.83%

LR+MBGD+ dynamic_lr	训练集	78.09%	78.33%	78.50%	77.86%	78.34%	78.23%
	验证集	**78.19%**	78.06%	77.25%	**78.25%**	**77.63%**	**77.88%**
LR+Adam+ dynamic_lr	训练集	78.09%	78.33%	78.50%	77.94%	78.36%	78.24%
	验证集	**78.19%**	78.06%	77.25%	**78.25%**	77.56%	77.86%
Sklearn_LR	训练集	78.06%	78.33%	78.47%	77.98%	78.41%	7825%
	验证集	78.19%	78.13%	77.19%	78.13%	77.50%	7782%
Sklearn_SVM+ Linear_kernal	训练集	78.13%	78.19%	78.48%	78.12%	78.00%	78.26%
	验证集	78.13%	77.81%	77.19%	78.06%	78.42%	77.85%
Sklearn_SVM+ rbf_kernal	训练集	82.50%	83.04%	82.48%	82.90%	82.81%	82.81%
	验证集	77.56%	76.31%	77.25%	77.56%	76.87%	77.11%

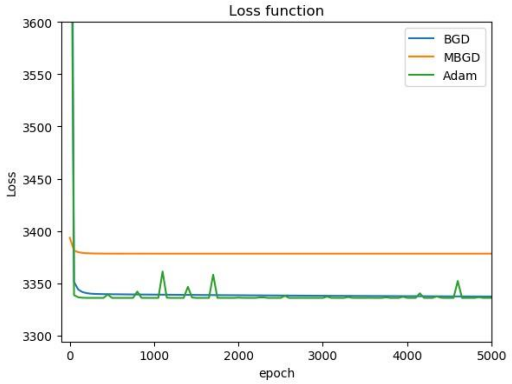
（表 2 逻辑回归算法的准确率）

5.2 梯度下降法的优化与 LR 收敛速度

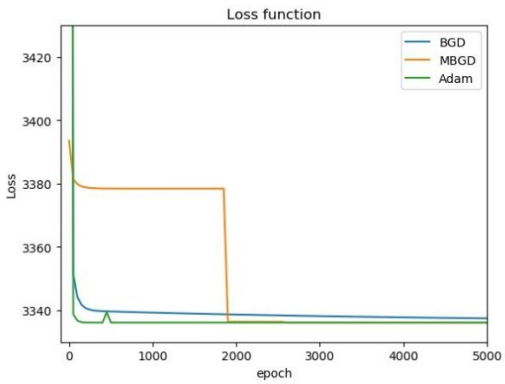
我使用了批梯度下降(BGD)，随机梯度下降(SGD)，小批量梯度下降(MBGD)和亚当优化(Adam)四种算法，使用第一折作为验证集，后 4 折作为训练集，使用 0.05 的初始学习率迭代 20000 个 epoch。我们可以通过观察损失函数的变化判断收敛速度。

如图 3，在不动态调整学习率的情况下，使用 BGD 算法的收敛速度最慢，20000 次迭代后还没有完全收敛，不过 BGD 算法得到的解最优秀。Adam 优化算法一直在最优解附近振荡，而 BGD 算法则并没有收敛到最优解。

我尝试了动态调整学习率，结果如图 4 可以看到，Adam 优化算法在 1000 个 epoch 不到时就已经收敛到了最优解，MBGD 算法在 2000 个 epoch 时学习率发生一次下降后也基本收敛到最优解，BGD 算法则始终没有收敛。



（图 2）

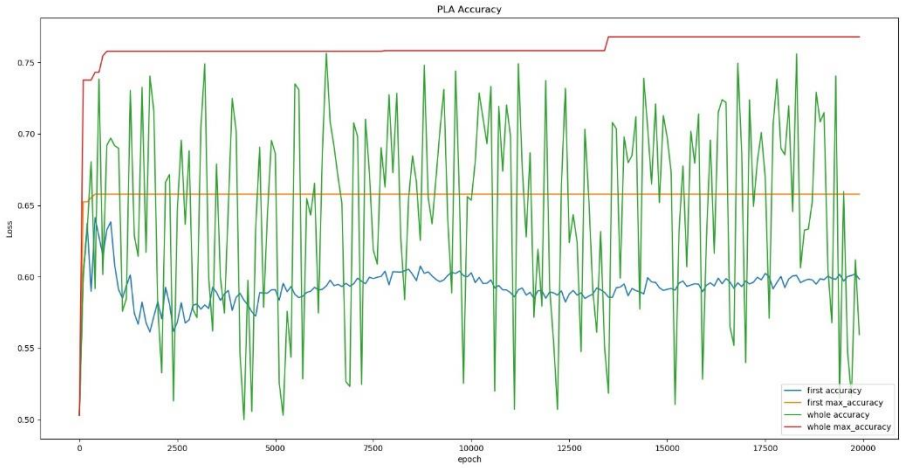


（图 3）

因此，我选择了动态调整学习率的 Adam 优化算法，因为它收敛得又快又准。

5.3 PLA 算法的迭代次序

如图所示，使用首误遍历法训练集上的准确率能够基本收敛，但总的准确率较低，只有 66%左右。而使用全局错误遍历，准确率在 50%到 70%之间震荡，但最高准确率大幅提升可以达到接近 77%，在测试集上的准确率也超过了 76%。



(图 4)

6 思考题

6.1 有什么手段可以使 PLA 适用于非线性可分的数据集

主要有两类手段，第一类手段是在线性分类的基础上找到一组较优的解（如(1)），另一类手段是使用一些方法（如(2),(3),(4)），在非线性的域上进行分类。

1）限制迭代次数，使用训练集或在训练集中再划分一个验证集，根据数据集上的准确率选择准确率最高的权重作为最终权重，用于分类测试。

2）引入核函数，将线性空间上的数据点映射到非线性空间，在非线性上分类。

3）引入 boosting，使用多个 PLA 作为弱分类器，加权获得分类结果。

4）引入多层 PLA，即构建一个神经网络，获得非线性的分类结果。

6.2 不同的学习率对模型收敛有何影响

1）收敛速度：在凸函数中，使用过小的学习率会降低收敛速度。在非凸的函数中除了速度降低外，过小的学习率还很容易使优化器收敛于局部最优解。使用合适的较大的学习率，收敛的速度则会较快，对于非凸的函数还可以一定程度上防止陷入局部最优解。

2）是否收敛：无论是在凸函数还是非凸函数，使用小的学习率一定会收敛。过大的学习率会导致梯度下降不收敛，每一步优化有可能会使权值移动到 Loss 值更大的地方。

6.3 使用梯度的模长是否为零作为梯度下降的收敛终止条件是否合适？为什么？一般如何判断模型收敛？

使用梯度的模长大小是否趋近于零作为梯度下降的收敛终止条件并不合适。梯度趋于零可能是落入到了局部最优解，或鞍点，或大面积的平坦区域。它有可能并没有收敛到最优点。对于逻辑回归，因为它是凹的所以不存在局部最优和鞍点。一般当迭代了一定次数且损失函数长期处于一个固定值时，可以认为模型收敛了。

7 结论

本次试验，我实现了逻辑回归和感知机学习两种线性分类模型，并对同一数据集进行分

类。在 PLA 算法上我的模型取得了 77%的分类准确率，这一结果超过了 sklearn 库上的结果。在 LR 算法上我的模型取得了 78%的分类准确率，这个结果也与 sklearn 库上的结果相当。

由于本次实验中模型在训练集和验证集上准确率基本相同，我认为基本没有出现过拟合的现象，也就没有执行正则化的操作。

本次实验的数据集是线性不可分的，本次实验的结果也证明了线性模型的局限性。想要再该数据集上提升准确率，只能使用非线性的模型，比如引入核函数，比如引入 boosting，比如引入多层的神经网络。