中山大学数据科学与计算机学院

移动信息工程专业-人工智能

本科生实验报告

（2017-2018 学年秋季学期）

课程名称：**Artificial Intelligence**

教学班级		专业（方向）	**移动互联网**
学号		姓名	**Jw**

一、 实验题目

感知机学习算法 -------- Perceptron Learning Algorithm

二、 实验内容

1. 算法原理

简介：感知机学习算法，是用来解决二维或者高维的二元分类的线性可划分问题。在这种问题中，每个特征向量的标签只有两种情况（+1，-1），我们要做的事情，就是在这个 n 维空间中找到一个超平面，这个超平面将这个空间划成两部分，其中一部分中的点的标签我们预测为+1，另一部分的点预测为-1，并且我们希望令预测错误的情况最小化。

算法步骤：设样本 x={x1,x2,…xd} 权重向量 w={w1,w3,…wd},样本实际标签 y={y1,y2,…yd},阈值 threshold，预测方式如下：

If $\sum_{i=1}^{d} wi * xi \geq threshold$, 预测为+1

If $\sum_{i=1}^{d} wi * xi < threshold$, 预测为-1

可得 predict = sign($\sum_{i=1}^{d} wi * xi - threshold$) =sign($\sum_{i=1}^{d} wi * xi + (-threshold)(+1)$)

为了简便计算，令 w0 = -threshold; x0 = 1，可得

predict = sign($\sum_{i=0}^{d} wi * xi$)

由此得到求解 PLA 问题的步骤：

1.给每个样本 x 前面加上 x0=1

2.初始化权重向量 w=随机向量（本次实验采用了全 1 向量）

3.每当遇到一个预测错误的样本，即 $sign(w_t^T x_n) \neq y_n$

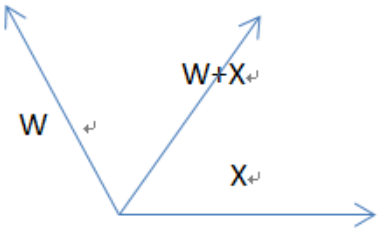
就更新 $w_{t+1} \leftarrow w_t + y_n x_n$

重复步骤 3，知道所有样本被划分正确，此时的 w 就是我们需要的权重向量，用这个训练得到的权重向量去预测测试集数据即可。

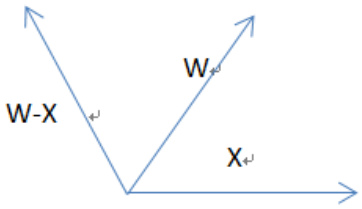
权重向量调整方式的理解:在这些步骤中，我觉得需要重点理解的是，权重向量的调整方式，下面通过例子来说明这个的调整策略的正确性。首先我们要先知道两个向量 ***A***、***B*** 的内积关系式 ***AB***=|A||B|cos(***A***，***B***)。需要调整的情况有两种:

1）正样例被预测为负。此时向量 w 和向量 x 的内积为负数，根据内积关系式，可

以得知它们之间的夹角是钝角，要使预测值正确，需要使 dot（w，x）>=0，即希望通过调整 w 向量，使得两者的夹角是锐角。很显然，这 w 和 x 相加的和向量 w'，能够让夹角缩小一半，如下图：



2）负样例被预测为正。此时就是 ***w*** 和 ***x*** 的夹角为锐角，而实际应该为钝角，这种情况刚好是 1）中的反例，所以让 ***w'=w – x*** 就可以让两者夹角扩大一倍，如下图：



当然这样子的调整有可能不是一步到位，所以需要多次调整。比如第二种情况扩大一倍之后还是锐角，就需要下一次遇到它再调整一次。对于线性可划分集合，PLA 算法是可收敛的。这里不作推导证明，我感性地认识了一下，由于每次调整都是成倍变化，类似于二分法，由于是线性数据集，所以总能通过“二分”找到最优点。

对于非线性可划分数据集: 这时通常无法找到合适的 w 使得所有点都被预测正确。此时一般有两种解决方法：

1.设置迭代次数，迭代完成就返回此时的 w，不管它是否能满足所有点。

2.先初始化一个 w，使得在训练集里以此 w 来划分后，分类错误的样本最少。即相当于有一个口袋放着一个 w，把算到的 w 跟口袋里的 w 比对，放入比较好的一个 w，这种算法又被称为口袋（pocket）算法。

2. 伪代码

PLA 原始算法

```
Initialize w = (1,1,1……,1)
Initialize tmax (迭代次数上限)
For t = 1,2,3……tmax
    For i = 1,2,3….len(matrix)
        y = dot(matrix[i], w)
        w = w + [y != label[i]] * xy
return w
```

PLA 口袋算法

```
Initialize w = (1,1,1……,1)
```

pocket_w - w

Initialize tmax (迭代次数上限)

For t = 1,2,3……tmax

  For i = 1,2,3….len(matrix)

    y = dot(matrix[i], w)

    w = w + [y != label[i]] * xy

    if error(w) < error(pocket_w) then

      pocket_w = w

3. 关键代码截图（带注释）

PLA 原始算法：

1.训练函数

```
def train(self, tot=278):
    cnt = 0
    self.cnt = tot
    self.w = np.ones(len(self.trainset[0][:-1])) #初始化权重向量
    while cnt < self.cnt: #迭代指定次数
        cnt += 1
        flag = 1 #判断当前是否全部划分正确
        sys.stdout.write("\rPercent: %d/%d" % (cnt, self.cnt)) #迭代次数进度条
        sys.stdout.flush()
        for i in range(len(self.trainset)):
            x = self.trainset[i][:-1]   #取出x向量
            label = int(self.trainset[i][-1])#取出真实label
            y = sign(np.dot(self.w, x))  #x和权重向量w做内积
            if y == label:
                continue
            flag = 0
            self.w = self.w + np.array(x) * label #预测错误，调整w
        if flag == 1: #如果已经全部划分正确就不继续划分了
            break
```

2.预测验证集并求的各种指标，注意这里使用的 w 是训练结束后求的的 w，在我的代码里它是类的全局变量

```
def PLA(self, folderpath)r()
    self.valset = self.loadcsv(join(self.folderpath, "val.csv"))
    for i in range(len(self.valset)):
        x = self.valset[i][:-1]  #取出x向量
        label = int(self.valset[i][-1]) #取出真实label
        y = sign(np.dot(self.w, x)) #预测的label
        self.update(y, label)  #根据预测label和真实label，更新4种指标的值
    #求准确率、精确率、召回率、F值
    Accuracy = (self.TP + self.TN) / (self.TP + self.FP + self.TN + self.FN)
    Precision = self.TP / (self.TP + self.FP)
    Recall = self.TP / (self.TP + self.FN)
    F1 = (2 * Precision * Recall) / (Precision + Recall)
    print("\ncnt = ", self.cnt)
    print("Accuracy = ", Accuracy)
    print("Precision = ", Precision)
    print("Recall = ", Recall)
    print("F1 = ", F1)
    print("")
    return [Accuracy, Precision, Recall, F1]
```

3.预测测试集的函数与预测验证集的函数基本一样，只是不用算 4 种指标，不再重复贴代码。

PLA 口袋算法：

1.训练函数，与原始算法不同的是，每次发现错误修正了 w 之后，要求新 w 造成的错误率，通过错误率比较判断是否更新 w

```
def train(self):
    for i in range(self.cnt):
        sys.stdout.write("\rPercent: %d/%d" % (i+1, self.cnt))
        sys.stdout.flush()
        self.init_indicator()#每次迭代需要初始化指标
        for j in range(len(self.trainset[0])):
            y = sign(np.dot(self.w, self.trainset[0][j]))#x*w
            label = self.trainset[1][j]
            if y != label:#发现预测错误
                tmp = self.trainset[0][j] * label
                nw = self.w + tmp #求解新w
                error = self.getError(nw)
                if error < self.error:#若新w的错误率比较小则更新口袋中的w
                    self.error = error
                    self.w = nw
    return self.w
```

2.准确率、精确率、召回率、F 值计算函数:

```
def update(self, y, label):
    if label == 1 and y == 1:
        self.TP += 1
    elif label == 1 and y == -1:
        self.FN += 1
    elif label == -1 and y == -1:
        self.TN += 1
    else:
        self.FP += 1
    self.Accuracy = (self.TP + self.TN) / (self.TP + self.FP + self.TN + self.FN)
    if self.TP + self.FP == 0:
        self.Precision = 0
    else:
        self.Precision = self.TP / (self.TP + self.FP)
    if self.TP + self.FN == 0:
        self.Recall = 0
    else:
        self.Recall = self.TP / (self.TP + self.FN)
    if self.Precision + self.Recall == 0:
        self.F1 = 0
    else:
        self.F1 = (2 * self.Precision * self.Recall) \
                  / (self.Precision + self.Recall)
```

3.预测验证集和测试集的函数与 原始算法的基本一样，不再重复贴代码。

4. 创新点&优化（如果有）

通过前面对口袋算法流程的分析，我们可以发现它的时间复杂度比较高，因为每一次迭代，都要做很多次整体的矩阵乘法，当数据集比较大的时候，显得尤其慢（我用 python 写很慢，不知道其他语言如何。）所以为了提速，我这里改成每一次迭代一旦能够成功更新，就退出当前迭代。代码如下

```
def train(self):
    for i in range(self.cnt):
        sys.stdout.write("\rPercent: %d/%d" % (i+1, self.cnt))
        sys.stdout.flush()
        self.init_indicator()#每次迭代需要初始化指标
        for j in range(len(self.trainset[0])):
            y = sign(np.dot(self.w, self.trainset[0][j]))#x*w
            label = self.trainset[1][j]
            if y != label:#发现预测错误
                tmp = self.trainset[0][j] * label
                nw = self.w + tmp #求解新w
                error = self.getError(nw)
                if error < self.error:#若新w的错误率比较小则更新口袋中的w
                    self.error = error
                    self.w = nw
                    break #跳出当前迭代
    return self.w
```

这样做本质上是减少了计算次数，不过这样子改，虽然明显能提速，但是会造成更容易陷入局部最优解的情况

三、 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

一个简单的小数据集：

Id	Feature1	Feature2	Label
Train1	-4	-1	+1
Train2	0	3	-1
Test1	-2	3	?

手动计算：

1.首先给每个样本数据加常数项 1 得

train1： x1 = {1, -4, -1} y1=+1

train2： x2 = {1, 0, 3} y2=-1

test1： x3 = {1, -2, 3} y3=?

2.初始化权重向量 w={1,1,1}

3.计算 $sign(w^Tx_1) = -1 \neq y1$

更新 w = w + {1*1，-4*1,-1*1} = {2,3,0}

计算 $sign(w^Tx_2) = +1 \neq y2$

更新 w = w + {1*-1，0*-1, 3*-1} = {1,-3,-3}

计算得 sign(w^Tx_1) = y1，sign(w^Tx_2)=y2

预测全正确，停止迭代

4.预测 test1，计算 sign($\mathbf{w}^Tx_3$) = -1，所以 y3=-1

PLA 原始算法结果:

```
C:\Users\freedom\Desktop\PLAtest>python PLA_initia1_15352408.py

w =  [ 1. -3. -3.]
label1 = -1
```

PLA 口袋算法结果:

```
C:\Users\freedom\Desktop\PLAtest>python PLA_pocket_15352408.py
w =  [ 1. -3. -3.]
label =  -1
```

可以看到两种算法的结果都正确

PLA 提速的口袋算法结果：

```
C:\Users\freedom\Desktop\PLAtest>python PLA_pocket_15352408.py
w =  [ 1. -3. -3.]
label =  -1
```

2. 评测指标展示即分析（如果实验题目有特殊要求，否则使用准确率）

PLA 口袋算法

口袋算法，由于每一次修改 W 后，都要先用跟整个特征矩阵做一次矩阵乘法，复杂度较高，我迭代次数设置了 300 次，以准确率衡量更新口袋中的 w 值，结果如下：

```
Percent: 300/300
Accuracy =  0.841
Precision =  0.6
Recall =  0.01875
F1 =  0.03636363636363636
```

上图中召回率很低，说明被误判的比例很高，这也是口袋算法的一个局限性，它用了贪心的思想，根据更新所比较的指标不同，使用了不同的贪心策略，从过程上看它跟爬山算法很像，很容易陷入局部最优，上图结果由于陷入了准确率的局部最优，而无法找到更好的解来提高其他指标的值。

PLA 提速的口袋算法：

```
Percent: 300/300
Accuracy =  0.837
Precision =  0.0
Recall =  0.0
F1 =  0
```

速度虽然变快了，但是从结果上看却更糟糕了，正样例全部预测错误。

PLA 原始算法：

我将迭代次数设置了从 1 到 500 次，每一次最后对验证集，由于这次测试集 label=-1 的占了 6/7，单纯以准确率衡量可能会造成过拟合，所以我选择按照 F 值从大到小排序，并从中选择正确率、召回率和精确率都尽可能大的迭代次数。结果如下：

	A	B	C	D	E
1	id	Accuracy	Precision	Recall	F1
2	474	0.803	0.428015564	0.6875	0.527578
3	278	0.836	0.488505747	0.53125	0.508982
4	53	0.8	0.418032787	0.6375	0.50495
5	209	0.796	0.412698413	0.65	0.504854
6	110	0.797	0.412244898	0.63125	0.498765
7	393	0.799	0.414937759	0.625	0.498753
8	198	0.786	0.398496241	0.6625	0.497653
9	297	0.79	0.403100775	0.65	0.497608
10	259	0.797	0.411522634	0.625	0.496278
11	216	0.821	0.451282051	0.55	0.495775
12	477	0.778	0.389285714	0.68125	0.495455
13	258	0.78	0.391304348	0.675	0.495413
14	418	0.811	0.431924883	0.575	0.493298
15	221	0.786	0.396946565	0.65	0.492891
16	424	0.788	0.399224806	0.64375	0.492823
17	168	0.796	0.409090909	0.61875	0.492537
18	466	0.781	0.391143911	0.6625	0.491879
19	441	0.785	0.395437262	0.65	0.491726
20	261	0.791	0.402390438	0.63125	0.491484
21	479	0.783	0.39245283	0.65	0.489412
22	416	0.791	0.401606426	0.625	0.488998
23	458	0.797	0.4092827	0.60625	0.488665
24	223	0.782	0.390977444	0.65	0.488263
25	326	0.789	0.398406375	0.625	0.486618
26	454	0.791	0.400809717	0.61875	0.486486
27	287	0.797	0.408510638	0.6	0.486076
28	83	0.799	0.411255411	0.59375	0.485934
29	490	0.782	0.390151515	0.64375	0.485849
30	280	0.788	0.396825397	0.625	0.485437
31	491	0.792	0.401639344	0.6125	0.485149
32	263	0.781	0.388679245	0.64375	0.484706
33	462	0.783	0.390804598	0.6375	0.484561
34	171	0.787	0.395256917	0.625	0.484262
35	211	0.791	0.4	0.6125	0.483951
36	111	0.795	0.405063291	0.6	0.483627
37	455	0.795	0.405063291	0.6	0.483627
38	499	0.812	0.431372549	0.55	0.483516

init_data

从表中可以看出，将迭代次数设置为 278，各个指标的值都比较高。说明这个参数调出来的模型，精确率和召回率高说明正确判定正样例的能力相对较高，分类效果相对较好。

四、 思考题

1. 有什么其他的手段可以解决数据集非线性可分的问题？

答：1. 更改衡量指标，准确率不是唯一标准，还可以使用精确率、召回率等

2. 当数据集很大的时候，可以考虑使用欠采样，即删掉一些数据，让数据集编的比较好划分。

2. 请查询相关资料，解释为什么要用这四种评测指标，各自的意义是什么。

答：**准确率 accuracy** = $\frac{\text{分类正确的样本数}}{\text{样本总数}}$

精确率 precision = $\frac{\text{预测正确的正样例数}}{\text{预测正确的正样例数}+\text{误判成正样例的负样例数}}$

召回率 recall = $\frac{\text{预测正确的正样例数}}{\text{预测正确的正样例数}+\text{误判成负样例的的正样例数}}$

F 值 = $\frac{2*precision*recall}{precision+recall}$

由上式可知，F 值就是精确率和召回率的调和均值，实际上是一种衡量精确率和召回率均值的一个指标。

在二元分类问题下，加入样本数据比较偏向于某一方，比如这次实验的测试集就偏向 label = -1， 这样即使全部预测-1 也可以得到较高的准确率，这个时候用准确率来评价分类器就显得很不足了。而且，也有可能会造成过拟合现象。

而精确率是为了求被预测为正的样本中有多少为正的样本。而召回率是求真正的正样例中有多少个被预测正确了。前者衡量一个分类器的准确性，后者衡量分类的完整性。用这两个指标就可以很好的衡量分类器是否可以真正的找出样例的能力（找正样例的能力高了，找负样例的能力自然也高）。

最后为了可以同时衡量两个指标，可以使用 F1，即精确率和召回率的调和平均值，F 值越高，且与两者相差越小，说明分类能力越好。