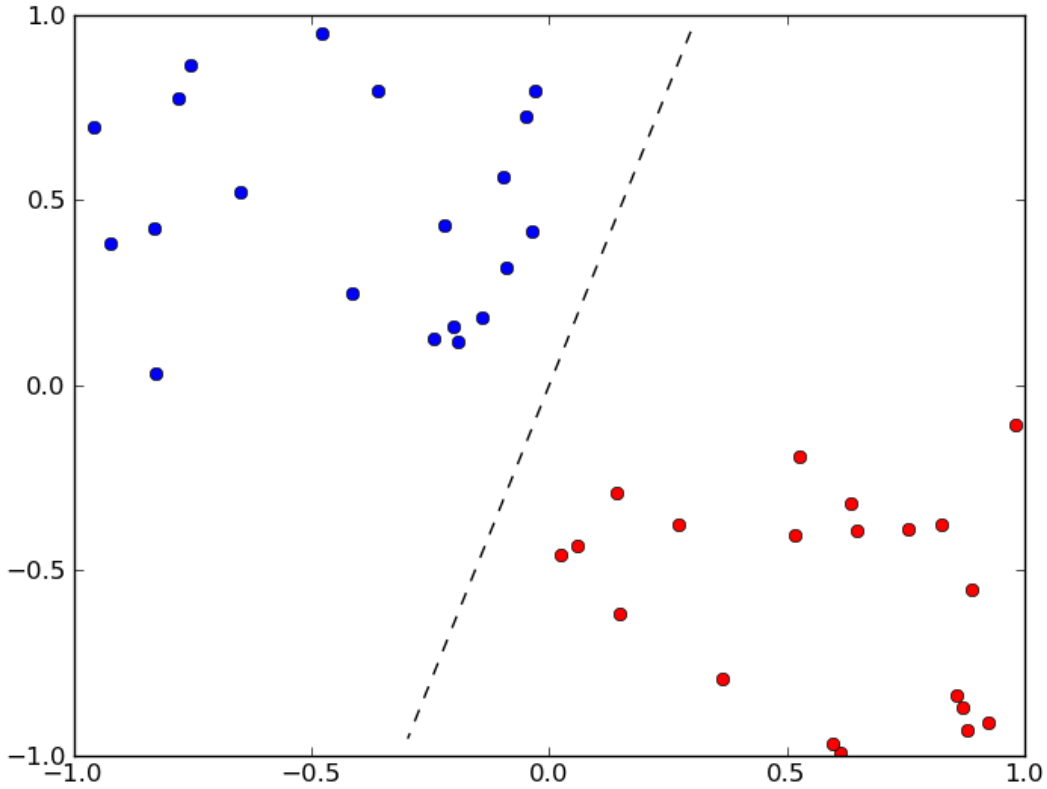
人工智能lab3实验报告

学号：**17338233**　专业：计科　姓名：郑戈涵

感知机(PLA)

算法原理

感知机是一种二分类的线性分类模型，用于解决线性可分问题，实际上是寻找一个超平面将特征空间划分为两个部分。二维情况的示意图如下：



输入的是样本的特征向量，输出的是数据的标签，只有两种，+1或-1。由于使用的函数（内积）是连续的，所以使用符号函数将结果转换为+1，-1。

函数定义如下：

$$f(x) = \text{sign}(w \cdot x + b)$$

其中w称为权重向量，b称为偏置。

目标是要最小化损失函数，即

$$\min L(w,b) = -\sum_{x_i \in M} y_i (w \cdot x_i + b)$$

损失函数的梯度为：

$$\nabla_w L(w,b) = -\sum_{x_i \in M} y_i x_i$$

$$\nabla_b L(w,b) = -\sum_{x_i \in M} y_i$$

通过使用梯度不断更新w，寻找最小损失函数的w，就是感知机的处理步骤。

实际操作时使用的是增广的权重向量，以及增广的特征向量。

$$\tilde{w} = \begin{pmatrix} w \\ b \end{pmatrix}$$
$$\tilde{x} = \begin{pmatrix} x \\ 1 \end{pmatrix}$$

步骤：

1. 预处理数据集，将特征向量处理成增广的特征向量$\tilde{x}$。
2. 初始化权重向量$\tilde{w}$。
3. 遍历每个样本$\tilde{x}_i$，计算$f(\tilde{x}_i)$，判断预测结果是否和y_i相等，不相等则进行更新，公式如下：
$$\tilde{w} \leftarrow \tilde{w} + \eta y_i \tilde{x}_i$$
4. 记录下正确率最高的权值向量。
5. 若没有样本预测错误或达到迭代次数最大要求，停止遍历。
6. 用获得的$\tilde{w}$预测测试集的样本。

伪代码或者流程图

```
procedure PLA(dataSet,maxIterTime,eta):
    input:dataSet 预处理得到的样本集合
          maxIterTime 最大迭代次数
          eta 学习率
    output:w 权重向量
    i<-0
    w<-[0,....0]
    best_accuracy<-0
    best_w<-w
    while i<maxIterTime
        miss<-0
        for row in dataSet
            if dot(w,row)!=label(row)
                miss<-miss+1
                w<-w+label(row)*eta*row
            end if
        end for
        if miss=0
            break
        end if
        accuracy<-miss/sizeof(dataSet)
        if accuracy > best_accuracy
            best_accuracy<-accuracy
            best_w<-w
        end if
    end while
    return best_W
end procedure
```

代码展示

PLA

普通的PLA不断遍历所有数据集，遇到错误的结果就更新w，终止条件为迭代次数达到要求。训练时会记录最好的准确率和w。

```
# 训练集经过预处理，每一列为样本增广特征向量及标签 [样本,1,标签]
def PLA(train_set,eta=1, maxIter=1e3):
# 初始化权值w，学习率，迭代变量，最佳准确率和最佳权值
    w=np.zeros((train_set.shape[1]-1))
    learnRate=1.0
    iterTime=1
    bestacc=0.0
    bestw=w
    while iterTime<maxIter:
        # 记录是否有样本出错
```

```
        miss=0
        accuracy=0.0
        for row in train_set:
            x=row[:-1]
            # 感知机公式
            res=np.sign(np.dot(x,w))
            if res==row[-1]:
                continue
            else:
                miss+=1
                # 更新权值
                w+=eta * row[-1] * x
        # 完全预测正确
        if miss==0:
            break
        iterTime+=1
        # 统计正确率
        accuracy=(train_set.shape[0]-miss)/train_set.shape[0]
        if(accuracy>bestacc):
            bestacc=accuracy
            bestw=w
#             print('accuracy=',accuracy)
    return bestw
```

但是使用该方法会发现，每次遍历完训练集得到的w，下一次遍历时有可能准确率更差，原因是其中有异常（不符合二分类）的样本影响了权值的更新，而每次遍历都会保留这些有问题的样本对权值的影响，因此可以使用随机样本来更新，保留当前最好的w。该算法被称为口袋算法。

口袋算法

```
def PLA_pocket(train_set,eta=1, maxIter=1e3):
    w=np.zeros((train_set.shape[1]-1))
    learnRate=1.0
    iterTime=1
    bestacc=0.0
    bestw=w
    accuracy=0.0
    while iterTime<maxIter:
        w=bestw
# 直接使用用户指定的学习率
        learnRate=eta
# 随机挑选出更新权值的样本
        x=random.choice(train_set)
        xres=x[-1]
        x=x[:-1]
# 计算预测结果
        res=np.sign(np.dot(x,w))
        if res==xres:
            pass
# 更新权值
        else:
            w+=x * xres *learnRate
            # 计算当前w的准确率
            accuracy=validate(train_set,w)
            # 若准确率更高，更新w
            if(accuracy>bestacc):
                bestacc=accuracy
                bestw=w
        iterTime+=1
        if accuracy==1:
            return bestw

    return bestw
```

评估模块

划分数据集使用的方法是和上一次一样的k_fold，即将数据集分为k份，用其中k-1份来训练得到模型，剩下一份用于验证，模型评估通过计算k次准确率，取均值得到，代码没有需要说明的部分因此不放出了。训练代码如下：

```
def evaluate(dataSet, maxIter=1e3, method=PLA, eta=1,k=10):
    accuracy=0.0
    for i in range(k):
# 第i份训练集和测试集
        s1,s2=k_fold(dataSet,k,i)
# 在训练集上使用PLA算法得到w
        w=method(s1,eta,maxIter)
        currAc=validate(s2,w)
# 在验证集上计算准确率
        accuracy+=currAc
        print(i," finished, accuracy:",currAc)
# 返回平均准确率
    return accuracy/k
```

测试/验证模块

测试时可以使用矩阵进行计算，只需要将数据矩阵和权值向量求乘积就可以得到预测结果。

评估时只需要将预测结果与标签做差，计算向量的平均值除以2（因为结果错误时差为2，比如1-（-1）=2）即可。使用矩阵计算相比遍历矩阵的每一行速度有很大的提升。

```
def test(testSet,w):
    return np.sign(testSet*np.mat(w).transpose())
def validate(dataSet,w):
    res=test(dataSet[:,:-1],w)
    return 1-np.mean(abs(res-np.mat(dataSet[:,-1]).transpose()))/2
```

实验结果以及分析

结果展示和分析

下面是使用训练集训练后在训练集上预测后输出的结果，可以看到，对于训练集，该模型的正确率还是比较高的。

```
In [23]:
for i in res:
    print(int(i))
executed in 1.04s, finished 01:06:02 2020-10-08
```

```
1
-1
-1
-1
1
-1
-1
-1
-1
1
-1
-1
-1
-1
-1
1
1
1
-1
```

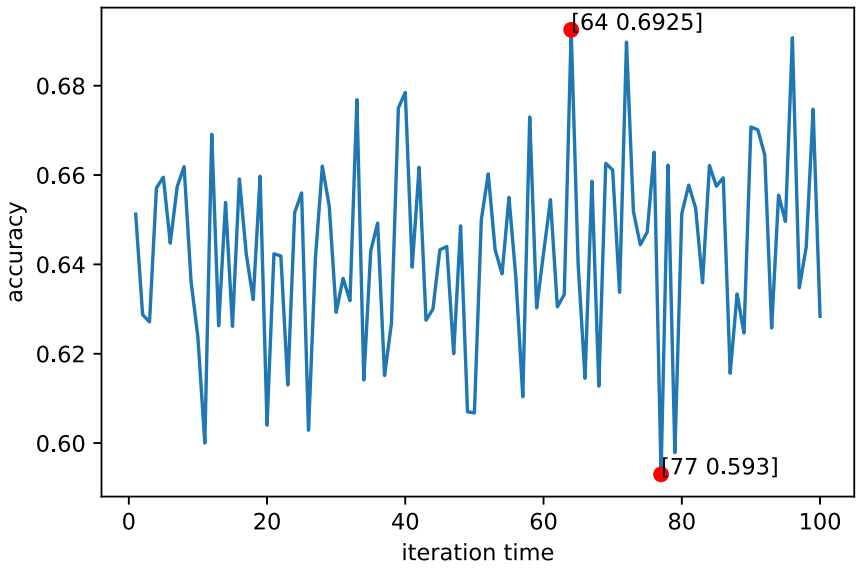
```
In [24]:
for i in train_set[:,-1]:
    print(int(i))
executed in 1.16s, finished 01:06:16 2020-10-08
```

```
-1
-1
-1
-1
1
-1
1
-1
-1
-1
-1
-1
1
-1
-1
-1
-1
1
-1
```

模型性能展示和分析

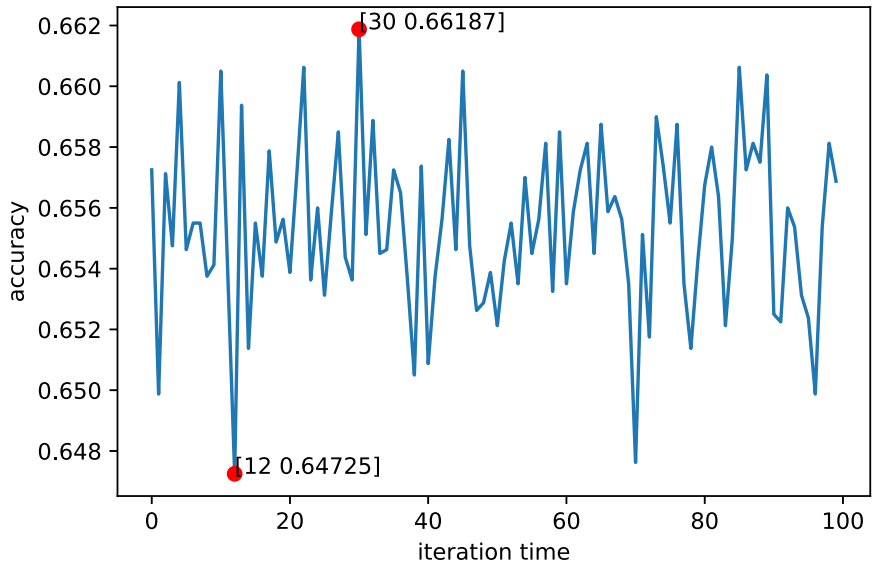
PLA

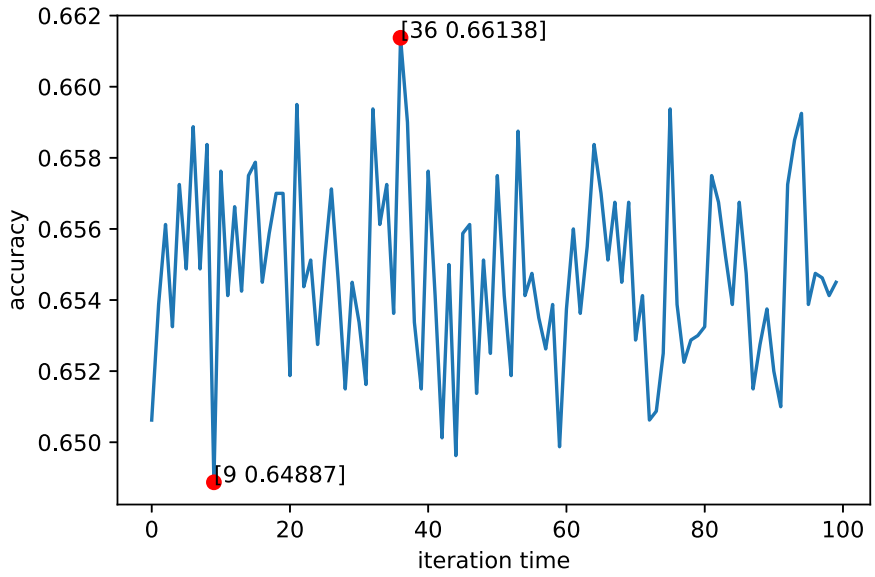
实验使用了k=10的k-fold方法划分数据集，使用不同的迭代次数（最高100次）进行实验，学习率为1，结果如下图：



可以看到，准确率随迭代次数产生的波动非常大，可以认为与迭代次数没有相关性，最高的情况准确率也只有0.6925，原因是该数据集并不是线性可分的。

尝试学习率为0.1和0.01，结果如下：

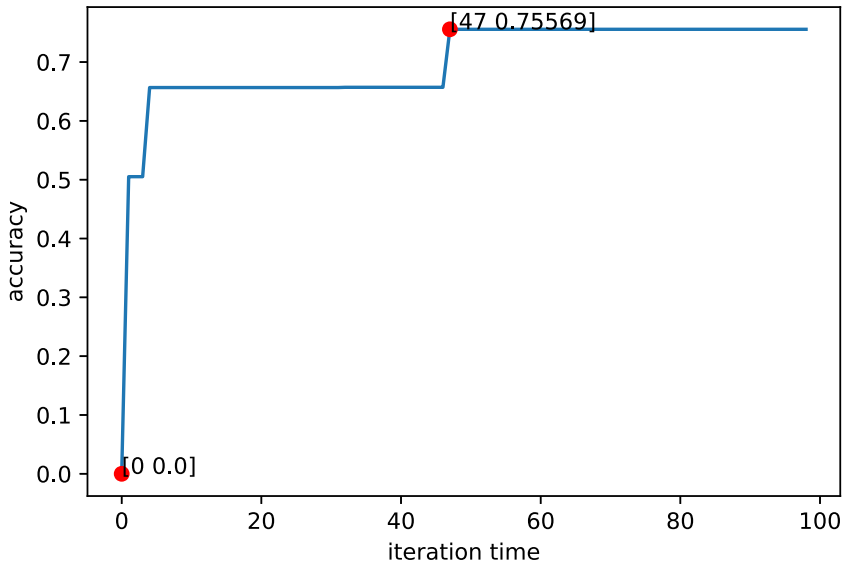




可以看到，准确率总体上并没有很大的变化，可以认为，对于PLA来说，不是线性可分的数据集，采用不同的固定学习率，并不会对学习过程有太大的影响。

口袋算法

使用了口袋算法后，迭代过程中的在训练集上的准确率变化如下图。可以看到，随着迭代次数的增加，准确率有稳步的上升，说明随机选择样本更新的对于提高训练集上准确率的效果更好。



下图是分别使用PLA，口袋算法，10重交叉验证，迭代100次，学习率为1的结果。可以认为，使用验证集测试训练集得到的模型时，两者的差别并不大，都约为65%。我认为是因为使用口袋算法会导致模型过拟合，泛化能力减弱。因此，对于不是线性可分的问题，采取何种PLA算法都难以得到满意的结果。

```
In [85]: evaluate('./lab3/train.csv', 1e2, PLA_pocket)
executed in 982ms, finished 23:25:25 2020-10-08

0 finished, accuracy: 0.5037499999999999
1 finished, accuracy: 0.63125
2 finished, accuracy: 0.6675
3 finished, accuracy: 0.6375
4 finished, accuracy: 0.6212500000000001
5 finished, accuracy: 0.66125
6 finished, accuracy: 0.7675
7 finished, accuracy: 0.6475
8 finished, accuracy: 0.71375
9 finished, accuracy: 0.68625

Out[85]: 0.6537499999999999
```

```
In [89]: evaluate('./lab3/train.csv', 100, PLA)
executed in 42.2s, finished 23:26:49 2020-10-08

0 finished, accuracy: 0.73125
1 finished, accuracy: 0.7362500000000001
2 finished, accuracy: 0.72125
3 finished, accuracy: 0.745
4 finished, accuracy: 0.6625
5 finished, accuracy: 0.645
6 finished, accuracy: 0.6675
7 finished, accuracy: 0.5375
8 finished, accuracy: 0.58375
9 finished, accuracy: 0.56125

Out[89]: 0.659125
```

逻辑回归(LR)

算法原理

LR是一种解决二分类问题的模型。最底层使用的是线性模型，并且假设事件发生的概率和线性模型的输出成指数关系，由于概率的取值范围为$[0,1]$，所以需要进行归一化，即

$$P(y=1|x)=\frac{e^{w^Tx}}{e^{w^Tx}+1}=\frac{1}{1+e^{-w^Tx}}$$
$$P(y=0|x)=1-\frac{e^{w^Tx}}{e^{w^Tx}+1}=\frac{1}{1+e^{w^Tx}}$$

令

$$\pi(x)=\frac{1}{1-e^{w^Tx}}$$

由于二分类的结果只有0和1，概率不高于0.5的认定为0，高于0.5的认定为1。

由于w是未知的，我们使用梯度下降法的方法更新w，直到满足条件。

使用的代价函数为似然函数，

$$\prod_{i=1}^{N} p(y|x_i) = \prod_{i=1}^{N} \pi(x_i)^{y_i}(1-\pi(x_i))^{1-y_i}$$

取对数可以得到：

$$L(\omega) = \sum_{i=1}^{N}[y_i \log \pi(x_i) + (1-y_i)\log(1-\pi(x_i))]$$

对上式计算梯度可得：

$$-\nabla_w L(w) = -\sum_{i=1}^{N}[y_i - \pi(x_i)]x_i$$

因此更新公式为：

$$w = w + \eta\sum_{i=1}^{N}[y_i - \pi(x_i)]x_i$$

为了提高迭代的速度，可以使用矩阵计算梯度，注意x是特征向量排成的矩阵。计算公式如下：

$$\nabla_w L(w) = x^T(y - sigmoid(x \cdot w))$$

也就是说，逻辑回归的做法是使用训练集计算梯度，使用公式不断迭代权重向量，直到权重向量收敛。

伪代码或者流程图

```
procedure LR(dataSet,maxIterTime,eta):
    input:dataSet 预处理得到的样本集合
          maxIterTime 最大迭代次数
          eta 学习率
    output:w 权重向量
    i<-0
    w<-[0,....0]
    while(i<maxIterTime)
        gd <- gradient(dataSet, w)
        w <- eta * gd + w
        i += 1
    end while
    return w
end procedure
```

代码展示

梯度下降函数

原理中给出了计算梯度的矩阵公式，因此对数据集进行处理，取出相应矩阵即可计算梯度。梯度下降法首先初始化全0的权重向量，不断计算梯度进行迭代。最后返回最终得到的w，为了减少工作量，这里没有对w进行测试以保存最优w。

```python
def gradient(dataSet, w):
    grad = 0.0
    x = dataSet[:, :-1]
    w = np.mat(w)
    label=np.mat(dataSet[:, -1]).transpose()
    return x.transpose() * (label - sigmoid(x * w))

def gradientDescent(dataSet, iterTime, learningRate):
    i = 0
    w = np.zeros(((train_set.shape[1] - 1), 1))
    while i < iterTime:
        gd = gradient(dataSet, w)
        w += learningRate * gd
        i += 1
    return w
```

sigmoid函数

该函数本来直接代入公式即可，但是实际情况并不是那么简单，如果公式是分子为1，那么对于较大的负数输入会出现溢出，否则对于较大的整数输入会出现溢出。因此需要分开计算并求和返回。numpy提供了带筛选的exp函数，可以很方便的实现。如果遍历向量一一判断使用哪种公式，效率非常低。

```
def sigmoid(x):
    mask = (x > 0)
    # 初始化两个和x相同长度的向量
    positive_out = np.zeros_like(x, dtype='float64')
    negative_out = np.zeros_like(x, dtype='float64')
    # 只计算大于0的元素
    positive_out = 1 / (1 + np.exp(-x, positive_out, where=mask))
    # 小于等于0元素置0
    positive_out[~mask] = 0

    # 只计算小于等于0的元素
    expX = np.exp(x,negative_out,where=~mask)
    negative_out = expX / (1+expX)
    # 大于0元素的置0
    negative_out[mask] = 0
    # 返回两者之和
    return positive_out + negative_out
```

测试函数

测试时也可以使用矩阵进行计算，只需要将数据矩阵和权值向量求乘积并用sigmoid函数做变换，然后舍入为0和1，就可以得到预测结果。

验证函数和PLA的一样，只是比较结果不需要除以2。

```
def test(testSet,w):
    res=sigmoid(testSet*np.mat(w))
    res[res>0.5]=1
    res[res<0.5]=0
    return res
```

实验结果以及分析

结果展示和分析

下面是使用训练集训练后在训练集上预测后输出的结果，可以看到，逻辑回归得到的预测结果正确率也比较高。

```
w=gradientDescent(train_set)
res=test(train_set[:,:-1],w)
for i in res:
    print(int(i))
```

executed in 1.81s, finished 01:13:09 2020-10-08

```
1
0
0
0
1
0
0
0
0
1
0
0
1
0
0
1
1
1
0
1
```

In [14]:

```
for i in train_set[:,-1]:
    print(int(i))
```

executed in 1.17s, finished 01:14:06 2020-10-08

```
0
0
0
0
1
0
1
0
0
0
0
0
1
0
0
0
0
1
0
1
```

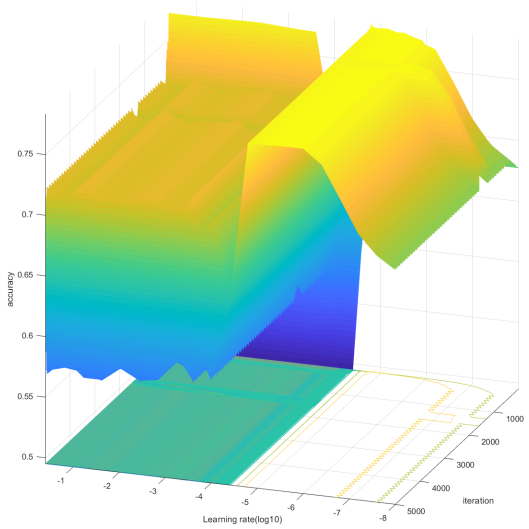
模型性能展示和分析

由于学习率对准确率的影响非常大，我统计了不同学习率下不同迭代次数的准确率，下图是k-fold参数为10的条件下训练的结果。由于数据量过大，这里只给出表格的一部分。

迭代次数\学习率	0.398107171	0.158489319	0.063095734	0.025118864	0.01	0.003981072	0.001584893	0.000630957	0.000251189	1.00E-04	3.98E-05	1.58E-05	6.31E-06	2.51E-06	1.00E-06	3.98E-07	1.58E-07	6.31E-08	2.51E-08	1.00E-08
4972	0.63975	0.664875	0.66	0.666875	0.661	0.656625	0.648875	0.672125	0.6475	0.670125	0.65025	0.778125	0.7785	0.778875	0.77875	0.779625	0.779375	0.78	0.76875	0.743625
4973	0.631875	0.665625	0.659875	0.666	0.653125	0.651375	0.63725	0.6635	0.623375	0.6705	0.6465	0.778125	0.7785	0.778875	0.77875	0.779625	0.779375	0.78	0.76875	0.743625
4974	0.6555	0.691875	0.6685	0.66225	0.666	0.678625	0.639875	0.638875	0.630125	0.67575	0.65025	0.778125	0.7785	0.778875	0.77875	0.779625	0.779375	0.78	0.76875	0.743625
4975	0.650375	0.683	0.647125	0.648125	0.653125	0.675375	0.631125	0.621375	0.630625	0.6665	0.6465	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.78	0.76875	0.743625
4976	0.682875	0.65675	0.6435	0.64075	0.65975	0.67875	0.66525	0.644875	0.66475	0.63425	0.65025	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.78	0.76875	0.743625
4977	0.67775	0.642875	0.638875	0.6295	0.652625	0.66525	0.661375	0.63775	0.663125	0.622125	0.6465	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.78	0.76875	0.743625
4978	0.654375	0.61725	0.643125	0.651125	0.636625	0.623375	0.671875	0.656375	0.6685	0.63575	0.65025	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4979	0.630875	0.597	0.63425	0.632625	0.6185	0.600375	0.6575	0.639375	0.662125	0.627375	0.6465	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4980	0.61225	0.638	0.647375	0.64675	0.63725	0.620125	0.635625	0.658375	0.6625	0.66575	0.65025	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4981	0.616125	0.649125	0.650375	0.654875	0.65	0.625	0.638625	0.66925	0.65025	0.654375	0.6465	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4982	0.664125	0.70075	0.68475	0.691375	0.691875	0.68875	0.66275	0.67175	0.6425	0.673125	0.65025	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4983	0.649	0.695	0.66775	0.670375	0.678875	0.679	0.642875	0.656625	0.63325	0.672625	0.6465	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4984	0.6735	0.68	0.65175	0.629	0.670375	0.686625	0.639	0.636375	0.655125	0.649875	0.65025	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4985	0.6805	0.670125	0.647	0.630375	0.665125	0.68525	0.646125	0.632	0.652	0.63575	0.6465	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4986	0.6895	0.638	0.64625	0.65425	0.658125	0.662375	0.681375	0.662625	0.66825	0.625	0.65025	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4987	0.664875	0.616	0.641125	0.6435	0.63675	0.64125	0.669375	0.641125	0.64775	0.612	0.6465	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4988	0.605	0.60625	0.642125	0.648625	0.610375	0.596875	0.638125	0.6395	0.649125	0.648875	0.65025	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4989	0.597375	0.595375	0.635125	0.64175	0.609875	0.595625	0.6235	0.6475	0.641	0.636375	0.6465	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4990	0.64575	0.67	0.678875	0.67975	0.672875	0.664625	0.651	0.677875	0.650875	0.66725	0.65025	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4991	0.628875	0.668875	0.6695	0.67275	0.658375	0.651125	0.63875	0.659375	0.626125	0.672625	0.6465	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4992	0.6595	0.693125	0.660125	0.649125	0.662	0.674125	0.64225	0.634625	0.638	0.6775	0.65025	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4993	0.660625	0.678375	0.643125	0.631625	0.65375	0.673375	0.63525	0.628875	0.64075	0.664	0.6465	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4994	0.68875	0.6405	0.639375	0.626875	0.659875	0.67575	0.667125	0.663125	0.668625	0.62875	0.65025	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4995	0.677125	0.631125	0.63175	0.620875	0.64575	0.659	0.65675	0.6465	0.662125	0.618375	0.6465	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4996	0.6345	0.612	0.63	0.653	0.6165	0.6085	0.654	0.648875	0.663375	0.638375	0.65025	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4997	0.617125	0.599	0.615875	0.633625	0.60725	0.58875	0.63775	0.640875	0.65	0.63075	0.6465	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4998	0.636875	0.66125	0.663625	0.66625	0.66475	0.637125	0.650375	0.669125	0.649	0.66625	0.65025	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.76875	0.743625
4999	0.634	0.66275	0.663625	0.668875	0.65925	0.634375	0.64575	0.666125	0.6325	0.65625	0.6465	0.778125	0.7785	0.778875	0.77875	0.779625	0.7795	0.780125	0.768875	0.743625

从表中可以看到，LR相比PLA在准确率上有非常大的提升，最高时可以达到0.78。

为了更清晰的看出两种因素对准确率的影响，我使用了matlab的surfc函数对上表进行了绘图，结果如下：



可以看到，迭代次数在超过300后对准确率基本没有影响，而且在学习率高于10^{-5}时准确率的波动非常大，几乎没有规律，而学习率从10^{-8}开始增加时，准确率有明显的上升，还可以注意到的是，在学习率非常小（接近10^{-8}）时，迭代次数在1500左右提升的比较快，但是之后又会下降。因此，一般来说，学习率接近10^{-5}会得到比较好的模型。

思考题

- 不同的学习率对模型收敛有何影响？从收敛速度和是否收敛两方面来回答。
  - 学习率高会使权重向量更新的更快，如果大小合适，可以加快收敛的速度，如果过大导致模型在低谷的两边反复移动，甚至不收敛。学习率低会导致收敛速度较低，因为模型的参数几乎不怎么变化。对于非凸的问题，低学习率可能导致收敛到局部最优解。
  - 学习率高有时候会因为反复移动时梯度不断增加而更加远离低谷，无法收敛并且准确率很低。学习率低不会出现这种问题，因为梯度近似估计时要求移动的距离非常小，学习率越低越符合要求，只要迭代次数足够多，一定能够收敛。
- 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？
  - 不合适，在特征非常多的时候，即使模型几乎收敛，模长依旧非常大，如果通过梯度的模长来终止收敛，有可能会导致训练无法结束。
  - 一般可以使用迭代次数，梯度的变化率，或者使用下降的学习率，在学习率低于一定程度时终止。