PRML-lab2-分类

概述

本次实验实践了几种经典的分类方法：Least Square Model, Perceptron Algorithm, Logistic Regression。实验中，除基本要求的公式推导与模型构建外，也额外探究了部分模型性能的影响因子。整体而言，实验加深了我对经典分类算法的理解，也让我认识到线性代数在机器学习模型构建中的重要性。

Part 1

Least Square Model

算法描述

一言以蔽之，Least Square Model分类算法以最小平方误差（最小二乘）来估计参数，实现分类。对于多维线性求解

$$X\beta = y$$

而言，其损失函数为：

$$S(\beta) = \|y - X\beta\|^2$$

由于此处需要求出使得损失函数最小的模型，通过微分，可得模型解：

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

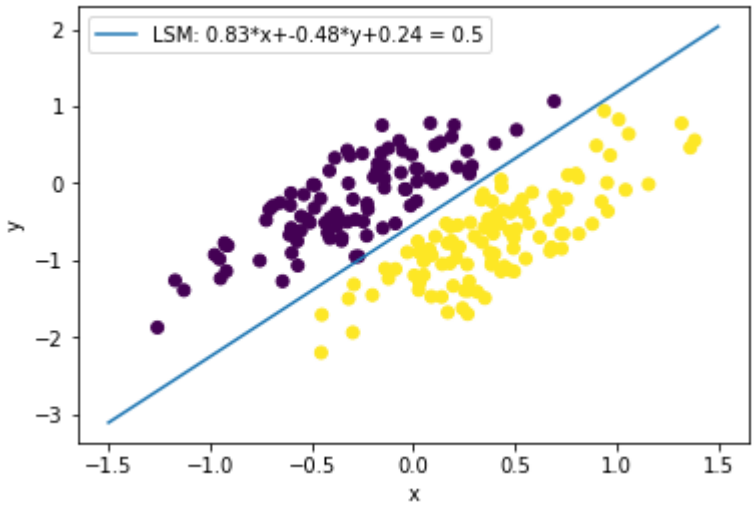
综上，其python实现核心代码如下：

```
def ListSquareModel(X,y):
    dataLength = len(X)
    Xexpand = np.insert(X,2,1,axis = 1)
    w = np.linalg.solve(np.dot(Xexpand.T,Xexpand),np.dot(Xexpand.T,y))
```

算法原理不难理解，但在模型构建中需要注意将输入集拓展维度，以此表示模型的常项（偏离值），此扩展的维度元素值全为1。

实验结果

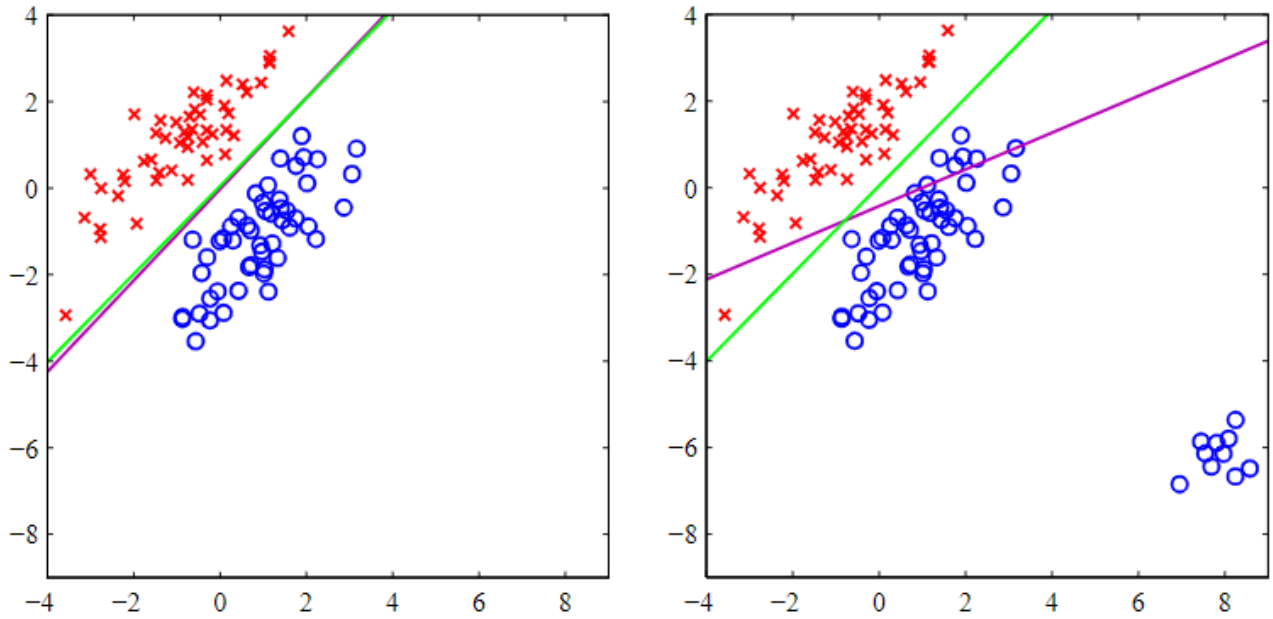
实验得到的模型分界线如下图所示：



从图中可以看出，分类的正确性为100%。此正确性也通过代码得到了验证。这说明，Least Square Model对本数据集而言效果尚佳。

模型探究

Least Square Model的好处在于原理清晰，计算简便。但值得注意的是，虽然在本实验中，Least Square Model达到了100%的正确率，但实际应用上Least Square Model有其较大的缺陷。如课堂中提及，



模型在添加一些较为“偏远”但正确的点之后反而降低了准确率。在实际分类中，上右图的情况是非常可能出现的，这也导致了Least Square Model在应用上的局限性。

Perceptron Algorithm

算法描述

Perceptron Algorithm的步骤如下：

1. 随机初始化模型w
2. 对每个训练样本x进行：

1. 利用模型w计算预测值
2. 检查预测值与实际值偏差，并利用偏差值更新模型：

$$\Delta W = \eta * (\hat{y} - y) * x$$

3. 迭代步骤2，直到收敛。

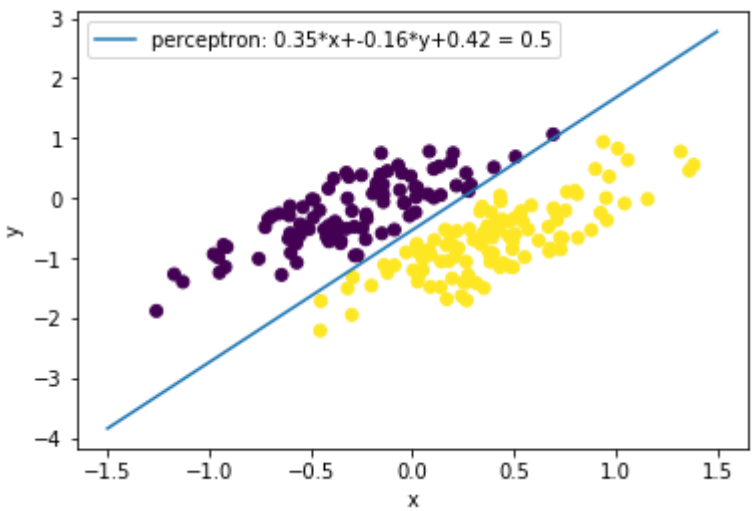
其python核心实现如下：

```
def perceptron(X,y,times,rate):
    #initial w
    w = np.random.rand(3)
    #expand X
    print("w init:"+str(w))
    Xexpand = np.insert(X,2,1,axis = 1)
    for i in range(times):
        for j in range(len(X)):
            result = np.dot(Xexpand[j].T,w)
            if result > 0.5:
                yPre = 1
            else:
                yPre = 0
            w += rate*(y[j] - yPre)*Xexpand[j]
    print("w final:"+str(w))
```

在核心代码之外，可做额外优化：在收敛时即返回结果，可以减少模型训练的时间；逐渐减少学习率，使模型更平缓地趋近收敛等。由于在之后另作训练次数与学习率的探究，略去此处算法的改进。

实验结果

实验得到的模型分界线如下图所示（times: 20，rate: 0.01）：

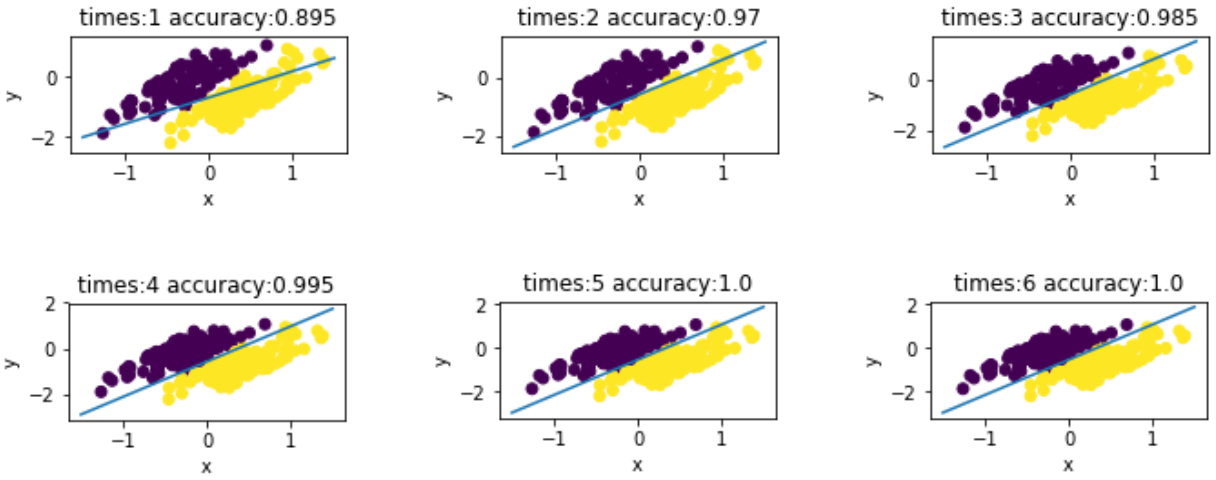


与Least Square Model一样，模型的正确率为100%。当然，在收敛前，模型的训练次数与速度会影响模型正确性。

模型探究

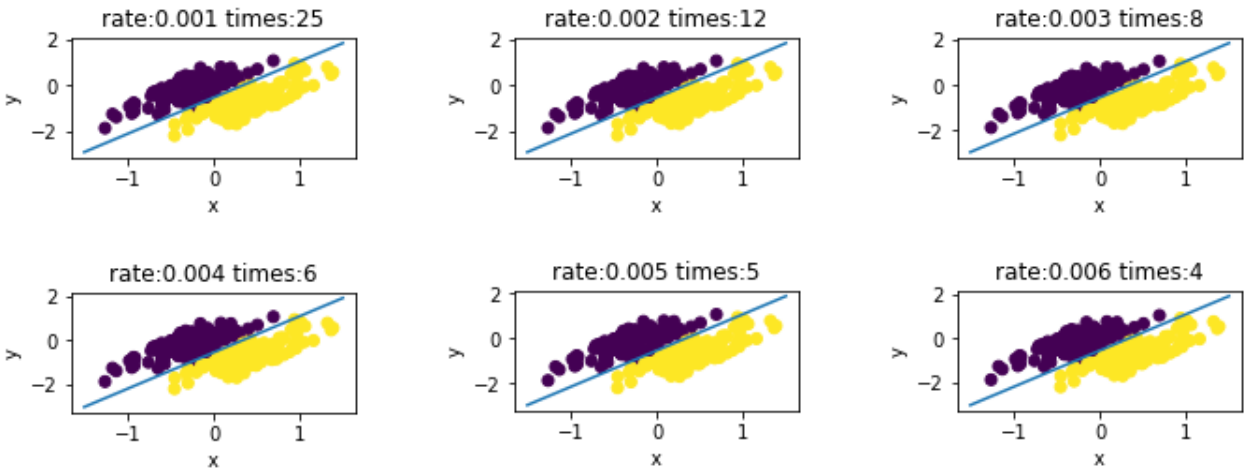
对于Perceptron Algorithm，模型的训练次数与训练速度值得关注。为了防止w随机初始化的影响，我们将其设置为0。

首先，我们探究模型训练次数的影响（此处设训练速率为0.005）：



通过5轮的训练，模型达到100%的准确率。另外，我们可以明显地看到，准确性函数为凸函数，即准确性一开始上升很快（如第一轮与第二轮之间，准确性相差0.075），之后逐渐趋于平缓（如第三轮与第四轮之间，准确性仅差0.010）。这与我们的认知相符合。

接着，我们探究训练速率的影响（保持速率不变，直到准确率为1，记录训练次数）：



同样地，随着速率的提高，训练次数一开始下降较快，之后逐渐趋于平缓。

Perceptron Algorithm同样有其缺陷，最重要的一点就是难以解决非线性分类问题。但能够解决Least Square Model遇到的“增加部分训练样本反而影响正确性”问题，因而泛用性比Least Square Model要高。

Part 2

Requirement1

Requirement1要求对数据做预处理操作。

input to multi-hot vector representation

假设某输入字符串为data，则预处理过程如下：

1. 小写化

   使用data.lower()将输入转为小写

2. 去除特殊符号

   ```
   for i in data:
           if i in string.punctuation:
               data = data.replace(i," ")
   ```

   将属于punctuation内的字符转为空格即可。

3. 分离为词语列表

   使用data.split()将输入分离为词语列表

4. 构建词典

   对dataset中所有data的所有word做计数，并删除计数小于10的word，剩余的word构成词典

5. 得到输入集

   ```
   for data in dataset:
       j = 0
       multiHotData = []
       for word in list(dic.keys()):
           if word in data:
               multiHotData.append(1)
           else:
               multiHotData.append(0)
       multiHotList.append(multiHotData)
   ```

   对于某个给定的输入数据data，multi-hot vector的长度等于词典长度。对词典中每个词逐一判断，若在data中，则为1；否则，为0。遍历所有的数据，得到所有multi-hot vector的列表。

在输入数据预处理时，是否应该去除数字是一个值得思考的点。在大多数情况下，单独数字的含义不是很大，会成为模型的干扰项，不过部分类型的数字（如年份）有其一定含义。在最终测量模型准确性时，我在是否去除数字这一点上做了对比。

target to one-hot representation

暴力解法：

```
target = []
for i in range(len(trainData['target'])):
    if trainData['target'][i] == 0:
        target.append([1,0,0,0])
    elif trainData['target'][i] == 1:
        target.append([0,1,0,0])
    elif trainData['target'][i] == 2:
        target.append([0,0,1,0])
    else:
        target.append([0,0,0,1])
```

合理解法：

```
target = []
for i in range(len(trainData['target'])):
    t = [0,0,0,0]
    t[trainData['target'][i]] = 1
    target.append(t)
```

Requirement2

Differentiate the loss function for logistic regression

step1 简化与表示

在计算开始前，我们规定x的初始维数为k，y的维度为c，样本数量为N，W为[k*c]的矩阵。

首先，我们考虑W与b的合并，以简化计算。已知公式：

$$\hat{y} = softmax(W^T x + b)$$

则可利用对x的拓维（第k+1维为1），使W转置的第k+1列表示b。此处记拓展维度之后（包含b）的W为W1

接下来，我们考虑对损失函数的求导：

$$L = -\frac{1}{N}\sum_{1}^{N} y_n^T log\hat{y}_n + \lambda\|W\|^2$$

将softmax函数带入，得：

$$L = -\frac{1}{N}\sum_{1}^{N} y_n^T log\left(softmax(W_1^T x_n)\right) + \lambda\|W\|^2$$

这里，将前项称作损失项，后项称为惩罚项，两者共同作用为损失函数。

step2 惩罚项求导

值得注意的是，上面的惩罚项为W而不是W1，即偏移量b不考虑在L2正则化中；这会在讨论的下一部分说明。惩罚项的求导结果显然：

$$\frac{\partial\left(\lambda\|W\|^2\right)}{\partial W} = 2 * \lambda * W$$

######

step3 损失项求导

为了简便计算，我们先考虑对单样本的求导：

$$L_0 = -y^T log\left(softmax(W_1^T x)\right)$$

已知softmax公式为:

$$softmax(v) = \frac{exp(v)}{1^T exp(v)}$$

由于y中仅有一项为1，故：

$$y^T \mathbf{1} = 1$$

因此，根据对数函数公式，可得：

$$L_0 = -y^T (log(exp(W_1^T x))) + log((1^T exp(W_1^T x)))$$

基于链式求导得：

$$dL_0 = -y^T dW_1^T x + \frac{1^T (exp(W_1^T x) \odot dW_1^T x)}{1^T exp(W_1^T x)}$$

到这一步，遇上了瓶颈。查阅资料得知，可将标量套上迹（trace）并利用迹公式结合律辅助求解：

$$a = tr(a)$$
$$tr(A^T (B \odot C)) = tr((A \odot B)^T C)$$

则：

$$\begin{aligned}
dL_0 &= -y^T dW_1^T x + \frac{1^T (exp(W_1^T x) \odot dW_1^T x)}{1^T exp(W_1^T x)} \\
&= -y^T dW_1^T x + \frac{tr(1^T (exp(W_1^T x) \odot dW_1^T x))}{1^T exp(W_1^T x)} \\
&= -y^T dW_1^T x + \frac{tr((\mathbf{1} \odot exp(W_1^T x))^T dW_1^T x)}{1^T exp(W_1^T x)} \\
&= -y^T dW_1^T x + \frac{tr(exp(W_1^T x)^T dW_1^T x)}{1^T exp(W_1^T x)} \\
&= (x(softmax(W_1^T x) - y))^T dW_1^T \\
&= ((softmax(W_1^T x) - y)x^T) dW_1^T \\
&= (\hat{y} - y)x^T dW_1^T
\end{aligned}$$

step4 最终结果

整合惩罚项与损失项，得到最终结果：

$$\frac{\partial L}{\partial W_1} = \frac{1}{N}(\hat{y} - y)x^T + 2 * \lambda * W$$

公式在程序实现上难度不大，主要难点在于不能用循环计算矩阵，而应在求导过程中保持矩阵的整体性。

Whether to regularize the bias term

正如上文提到的，我们没有必要对偏移项正则化。为了阐明这点，我们需要进一步了解L2正则的作用。

简单地说，L2正则化的目的是为了防止过拟合，提升模型的泛化能力。一般来说，我们认为参数值小或少的模型更能适应不同数据集，抗干扰能力强，从而在某种程度上避免过拟合。但限制参数w数量使高维参数为0属于NP-hard问题，我们退而求其次选择

$$\sum W^2 < C$$

以达到避免过拟合的目的。这就说明不能只考虑向梯度下降最快方向调整权重，而应在一定范围内（如上式代表的N维球体）调整参数。添加L2正则项的作用即在不影响梯度下降的前提下通过添加限制条件对模型化繁为简，减少高维参数大小，达到防止过拟合的目的。在一些实际应用范围较广的算法中（如岭回归），L2正则项是其中的重要一部分。

了解这一点，我们再考虑偏移项。由于偏移项参数对模型的曲率没有任何影响，故对偏移项正则没有意义，反而增加计算量。故我们没有必要对偏移项正则化。

check the gradient calculation

证明梯度（求导）正确性，只要验证某一点导数与其物理意义值近似相等即可：

$$f^{'}(x) = \frac{f(x+\Delta x) - f(x)}{\Delta x},\ \Delta x \to 0$$

我们将Δx设定为0.001，得出一系列偏导导数计算值(calculation)与实验近似值（approximation）,可以发现几乎相等。故梯度正确性得证。

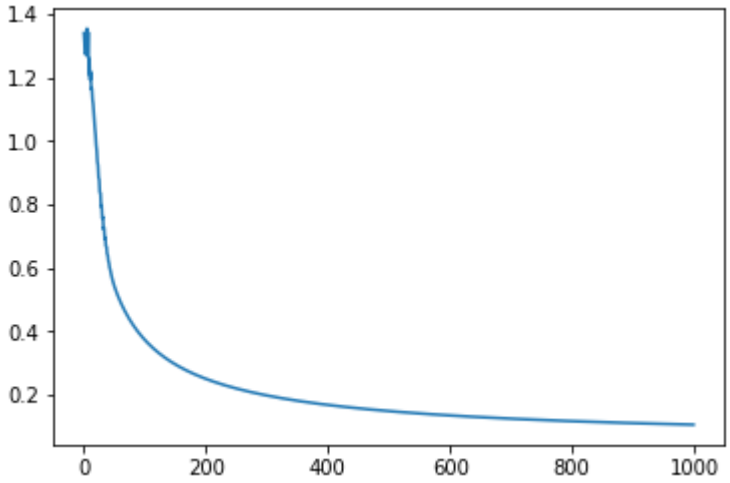
```
calculation: 0.000000   approximation: 0.000001
calculation: -0.003560   approximation: -0.003521
calculation: -0.011012   approximation: -0.011009
calculation: -0.002226   approximation: -0.002224
calculation: 0.001112   approximation: 0.001114
calculation: -0.000667   approximation: -0.000666
calculation: 0.001780   approximation: 0.001782
calculation: -0.001892   approximation: -0.001890
calculation: 0.000000   approximation: 0.000001
calculation: -0.000334   approximation: -0.000332
calculation: 0.014001   approximation: 0.014058
calculation: 0.006902   approximation: 0.006913
calculation: -0.029249   approximation: -0.029236
calculation: 0.000201   approximation: 0.000221
calculation: 0.001001   approximation: 0.001002
calculation: 0.003337   approximation: 0.003340
calculation: -0.002116   approximation: -0.002113
calculation: 0.002115   approximation: 0.002117
calculation: -0.003672   approximation: -0.003671
calculation: 0.000000   approximation: 0.000001
```

此外，还可以通过对导函数积分与原函数比较的方法验证正确性，在此不做展开。

Requirement3

训练结果

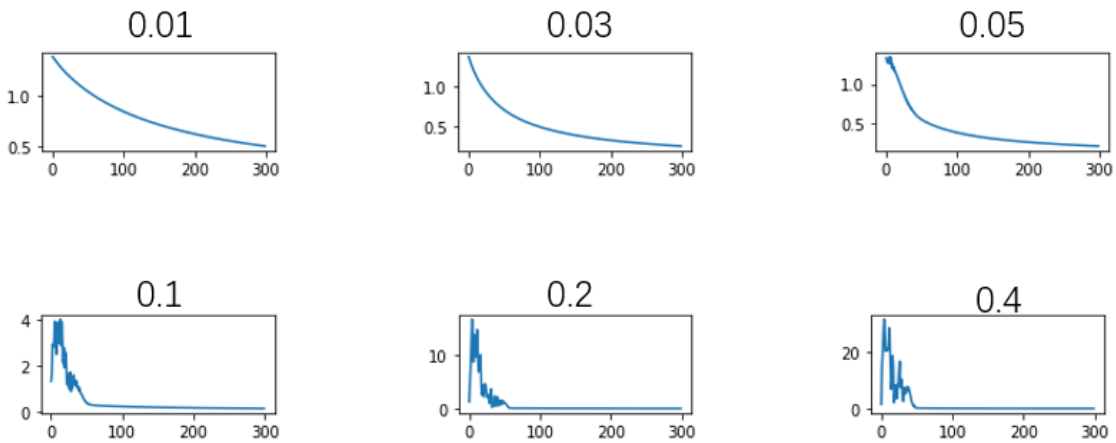
固定学习速率为0.05，λ为0.001，得到损失函数随训练轮次变化图如下：



可以看到，损失函数在刚开始训练时下降较快，之后逐渐趋于收敛。

how to determine the learning rate

以训练速率为变量，绘制不同训练速率下损失函数变化图如下：

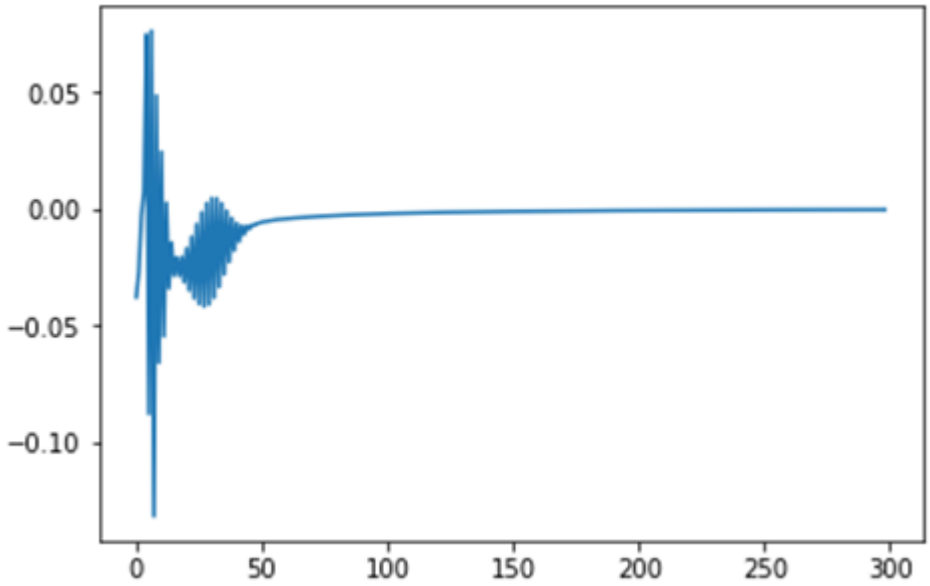


可以看到，训练速率较小（如0.01）时，训练300轮后损失函数仍然很大，影响了训练的效率；训练速率较大（如0.4）时，一方面在训练过程中图象扰动很大，另一方面其达到收敛的训练轮次相比更小的训练速率下也没有显著提升。在本实验中，训练速率为0.05为一种合理的选择。

与perceptron algorithm类似，随着训练不断减少训练速率，是一个折中效率与准确性的合理方法。

how to determine when to terminate the training procedure

随着训练轮次的不断增加，损失函数变化逐渐减少，再接下来的训练就没有很大意义了，故我们需要确定某点终止训练。显然，最直接的方法是当两次训练损失函数差值小于某阈值时终止训练。此阈值的设定是经验主义的，太小则耗费较长时间在无效训练上，太大则模型不够准确。我们查看不同轮次下两次训练结果的损失函数差值（此处训练速率为0.05）：

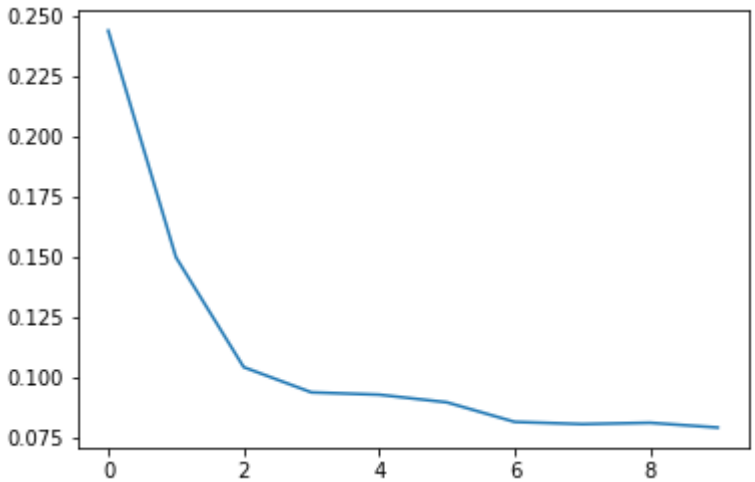


可以看到，当阈值设立为0.01以上时，训练在50轮内就终止，之后损失函数变化的累积量仍然是一个不小的值。根据此图估计，取阈值为0.0005不失为一个合适的选择。

Requirement4

stochastic gradient descent

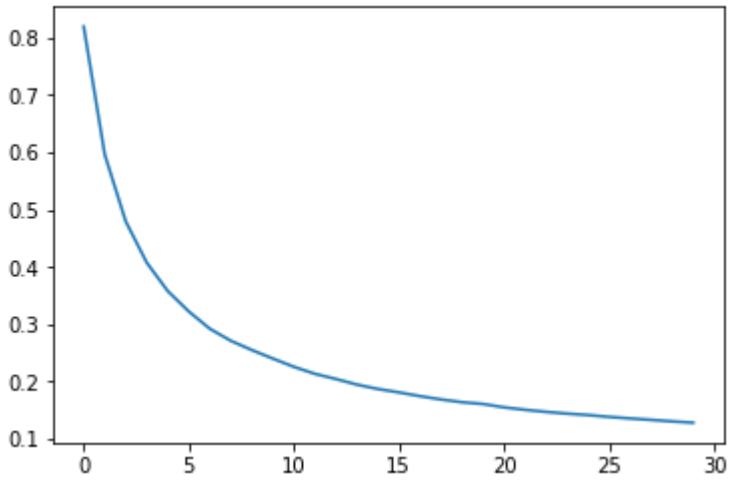
随机选取一条数据进行训练，并更新W，训练$dataLength$次为一轮。固定学习速率为0.01，lambda为0.001，得到训练结果如下：



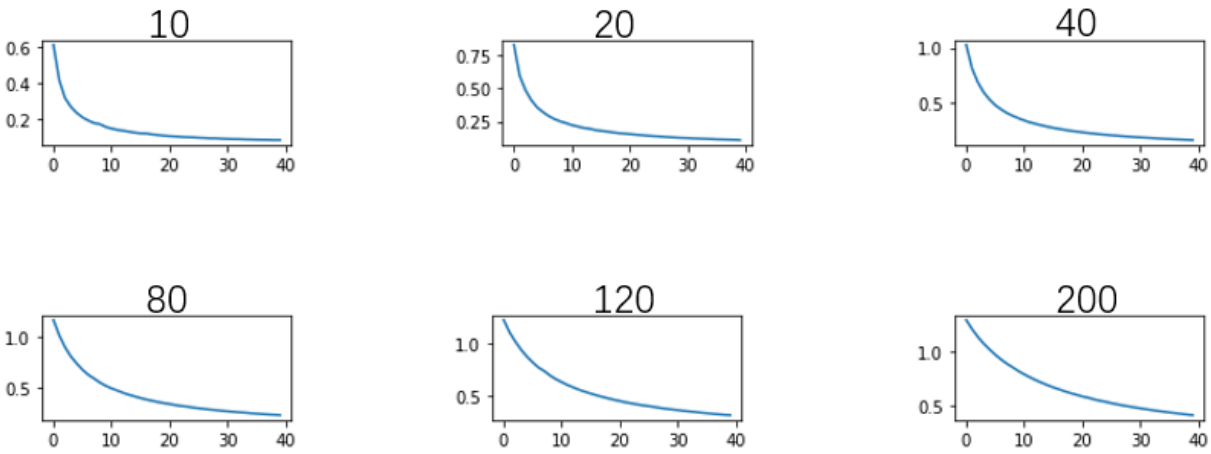
可以看到，个位数的训练轮次即可使损失函数下降到0.1以下，训练速度非常快。

batched gradient descent

随机选取连续batchSize条数据进行训练，并更新W，训练$dataLength/batchSize$次为一轮。固定学习速率为0.01，lambda为0.001,batchSize为20，得到训练结果如下：



值得注意的是，batchSize是影响miniBGD很重要的因素。下图列举了部分batchSize下loss function下降的情况：



可以看到，随着batchSize变大，收敛速度越来越慢。但是否batchSize越小越好呢？并不然。当batchSize太小时，容易出现SGD一样的缺陷，即可能错过global minima，导致准确率降低。

pros and cons

我们首先从效率角度比较三种模型。

三种模型的一轮均用到了dataLength次数据，但时间有区别，分别如下：

```
time per round for batched gradient descent:0.29122161865234375
time per round for stochastic gradient descent:0.42685842514038086
time per round for full batch gradient descent:0.23836040496826172
```

结果显而易见，SGD相比BGD调用更多次的softmax，BGD相比FBGD调用更多次的softmax，故一轮花费更多的时间，但时间差距不大（此处略有些出乎意料）。然而，训练轮次上，SGD与BGD相比FBGD要少得多；从效率角度出发，选取SGD与BGD是更为明智的选择。

下面是对三种模型综合pros与cons的讨论：

full batch gradient descent

pros

- 简单直接，易于想到
- 终止条件易判断

cons

- 占内存空间大
- 效率非常低

stochastic gradient descent

pros

- 速度快
- 占内存小
- 当存在许多local maxima/minima时表现较好
- 能用在一步一更新的online machine learning

cons

- 虽然其能够避免一些较差的local minima，但也有可能错过global minima
- 更新非常频繁，每一轮速度更慢
- 学习过程不可预测

batched gradient descent

pros

- 速度比FBGD快很多，占内存也不大
- 解决了单样本的噪声问题，相较SGD更可能达到global minima

cons

- 速度比SGD稍慢
- 需要额外调参（batchSize）
- 随机batch情况下学习过程不可预测

Requirement5

result for the three differently trained model on the test dataset

对训练速率0.05，λ为0.001，存在改变量小于0.0005收敛条件的FBGD，准确率为90.24%

对训练速率0.01，λ为0.001的SGD，准确率为91.04%

对训练速率0.01，λ为0.001，batchSize为20的miniBGD，准确率为91.24%

可见准确率miniBGD > SGD > FBGD，但差距不是太大。

值得注意的是准确率不仅与训练方式有关，还与训练数据有关。如去除字典里的数字，准确率会稍有下降，分别为90.17%，90.37%，90.44%。这也告诉我们数据预处理的重要性。