实验3：感知机&逻辑回归

> 教学班级：19计科2班
>
> 专业：计算机科学与技术
>
> 学号：19335262
>
> 姓名：张航悦

任务一：PLA感知机

一、 实验内容

1. 算法原理

感知机PLA

- 感知机是**二分类**的线性分类模型，输入为实例的特征向量，输出为实例的类别，分别是取+1和-1，属于判别模型。感知机学习的目标是求得一个能够将训练数据集**正实例点和负实例点完全正确分开的分离超平面。**
- 数学定义

  感知机针对二分类问题，输入是样本的特征向量$x \in R^n$，输出是样本的类别$y \in \{+1, -1\}$

  感知机函数可表示为：

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

  其中w和b为感知机模型参数，$w \in R^n$叫做权值向量，$b \in R$叫做偏置。后续将通过不断训练更新w和b的值使感知机找到正确的超平面。
- 学习策略

- 感知机将所有误分类点到分离超平面的距离之和作为损失函数。其学习目标是令该和尽可能小，即误分类点的数量尽可能少。

  设误分类点集合M，损失函数可写为：

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

- 使用随机梯度下降法对损失函数进行优化。

  损失函数的梯度为：

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

  选择一个误分类点(x_i, y_i)对参数进行更行，其中η表示学习率：

$$w = w + \eta y_i x_i$$
$$b = b + \eta y_i$$

- 算法步骤

  1. 初始化w和b为0，并设置学习率η。

     （在这里可以采取将权值向量w和偏置b合并为一个n+1的权值向量$w = (w^T, b)^T$，给每个特征向量添加一维常数项1，即$x = (x^T, 1)^T$的方法来矩阵化$w \cdot x + b$的运算）

  2. 遍历所有的样本点，计算每个样本点$y_i(w \cdot x_i + b)$的结果。随机选取一个误分类点(x_i, y_i)，即在该点有$y_i(w \cdot x_i + b) <= 0$，利用该点的$x_i, y_i$对w和b进行更新。

  3. 重复步骤2，直到训练集中没有误分类点。但由于所给数据集可能线性不可分，不可能找到一个模型将其正确划分，在这种情况下我们可以设置一个最大迭代次数，当迭代次数到达预设值时，便停止训练。

  4. 训练结束后我们可以得到一对w和b的值，在验证集上进行评测，以验证集上的预测准确率作为评测指标。

- 对于线性不可分的数据集，参考解决方法：

  1. 设置最大迭代次数，当迭代次数到达预设值时，停止训练
  2. 找到一组参数(w, b)，使得训练集使用该组参数进行划分后，分类的错误样本最少
  3. 引入核函数

划分训练集和验证集

采用实验2中的k-fold交叉验证法。

2. 伪代码

```
function pla(data_mat,label,iteration,rate):
//input:data_mat：特征集合
//      label：标签集合
//      iteration：最大迭代次数
//      rate：学习率
//output:权值向量w，偏置b

```

```
    w，b初始化为0

    for i in range(iteration):
        flag := 1
        for point in data_mat: //遍历样本集合
            judge := label * np.dot(w, point) //判断该点是否误分类

            //若有误分类点
            if judge <= 0 then
                w := w + rate * label * point
                b := b + rate * label
                flag := 0
                break
             end if
         end for
     end for

        //若全部分类正确，返回
        if flag == 1 then
            break
        end if

    return w, b
```

3. 关键代码展示

- 划分数据集和验证集

  采用k-fold交叉验证法划分数据集和验证集

```
def k_fold(dataset, k, i):
    """
    划分训练集和验证集
    :param dataset: 数据集
    :param k: 将数据集分为k个子集
    :param i: 选取第i个子集作为验证集
    :return: 划分出的训练集和验证集
    """
    total = len(dataset)
    step = total // k  # 每一步的步长 向下取整
    start = i * step
    end = start + step
    train_set = np.vstack((dataset[:start], dataset[end:]))
    valid_set = dataset[start:end]
    return train_set, valid_set
```

- 分割出特征集和标签集

  需要注意的是给定数据集的label取值为0和1，但感知机问题处理的label值为-1和1，所以在这里需要注意将为0的label值转换为-1

```
def split(dataset):
    """
    分割出特征集和标签
```

```
    """

    data = []
    label = []
    for row in dataset:
        if row[-1] == 0:
            label.append(-1) #给定数据集的label取值为0和1 这里将0标签转换为-1
        else:
            label.append(1)
        data.append(row[0:39]) #前40列为特征
    # 将特征集和标签集转为array，加快后续的运算
    data_mat = np.array(data)
    label_mat = np.array(label)
    return data_mat, label_mat
```

- 感知机PLA算法

  在本次实验中我实现了两种感知机算法。

  第一种感知机算法为，每次利用第一个误分类点的数据对w和b进行迭代更新。

```
def pla(data_mat, label, iteration, rate):
    """
    该pla方法为每次选择第一个误分类的点，进行更新
    :param data_mat:特征集合
    :param label:label集合
    :param iteration:最大迭代次数
    :param rate:学习率
    :return:训练出的w和b
    """
    w = np.zeros(len(data_mat[0]))  # 初始化w为0向量
    b = 0  # 初始化b为0

    for i in range(iteration):  # 最大迭代次数
        flag = 1  # 标记位 标识是否有误分类的点
        for j in range(len(data_mat)):  # 遍历所有样本点
            judge = label[j] * (np.dot(w, data_mat[j])+b)  # 判断该点是否被误分类
            if judge <= 0:  # 一点发现该点误分类，就立即对w和b进行更新
                w = w + rate * label[j] * data_mat[j]
                b = b + rate * label[j]
                flag = 0
                break
        if flag == 1:  # 若没有误分类的点
            break

    return w, b
```

但是若数据集线性不可分，在遍历样本点时，前面几个点一直都误分类，采用该方法将导致后面的样本点信息丢失。所以在这里我写了第二种感知机算法，从所有误分类的点中随机选取一个点，进行w和b的更新。但是后续验证发现，该方法和原始方法在准确率上并没有什么显著差别。

```
def pla2(data_mat, label, iteration, rate):
    """
    该pla方法为每次从所有误分类的点中随机选取一个，进行更新
```

```
    """
    w = np.zeros(len(data_mat[0]))
    b = 0
    for i in range(iteration):
        f = (np.dot(data_mat, w.T) + b) * label #矩阵运算，直接得到所有点的分类结果
        # 获取误分类点的位置索引
        idx = np.where(f <= 0) # idx为一个元组 第一个元素为误分类点的行索引 第二个元素为列索引
        num = f[idx].size
        if num != 0: # 若有误分类点
            point = np.random.randint((f[idx].shape[0]))  # 随机挑选一个误分类点
            # idx[0][point]为所取点的行索引
            temp_x = data_mat[idx[0][point]] #取出该行对应的x
            temp_y = label[idx[0][point]] #取出该行所对的label
            w = w + rate * temp_x * temp_y
            b = b + rate * temp_y
        else:
            break
    return w, b
```

- 预测结果

```
def predict(w, b, valid_set, label):
    """
    将训练得到的w和b带入验证集 检测正确率
    """
    total = len(valid_set)
    cnt = 0
    for i in range(total):
        temp = np.dot(valid_set[i], w) + b
        if label[i] == np.sign(temp) or temp == 0: #要注意当temp=0时也认为是预测正确
            cnt += 1
    return cnt / total
```

4. 创新点

查阅资料后，我了解到还有一种口袋PLA算法。口袋PLA算法主要是用于数据集线性不可分的情况，尽量找到一条犯错最少的线，来代替最精准分割正例负例的线。这个算法的具体做法是，首先我们手里有一条分割线w_t，发现它在数据点(x_n, y_n)上面犯了错误，那我们就纠正这个分割线得到w_{t+1},我们然后让w_t与w_{t+1}遍历所有的数据，看哪条线犯的错误少。如果w_{t+1}犯的错误少，那么就让w_{t+1}替代w_t，否则w_t不变。

具体代码实现如下。

```
def cal_wrong(w, b, data_mat, label):
    """
    pocket_pla 计算当前该点的所犯错误数
    """
    count = 0
    for i in range(len(data_mat)):
```

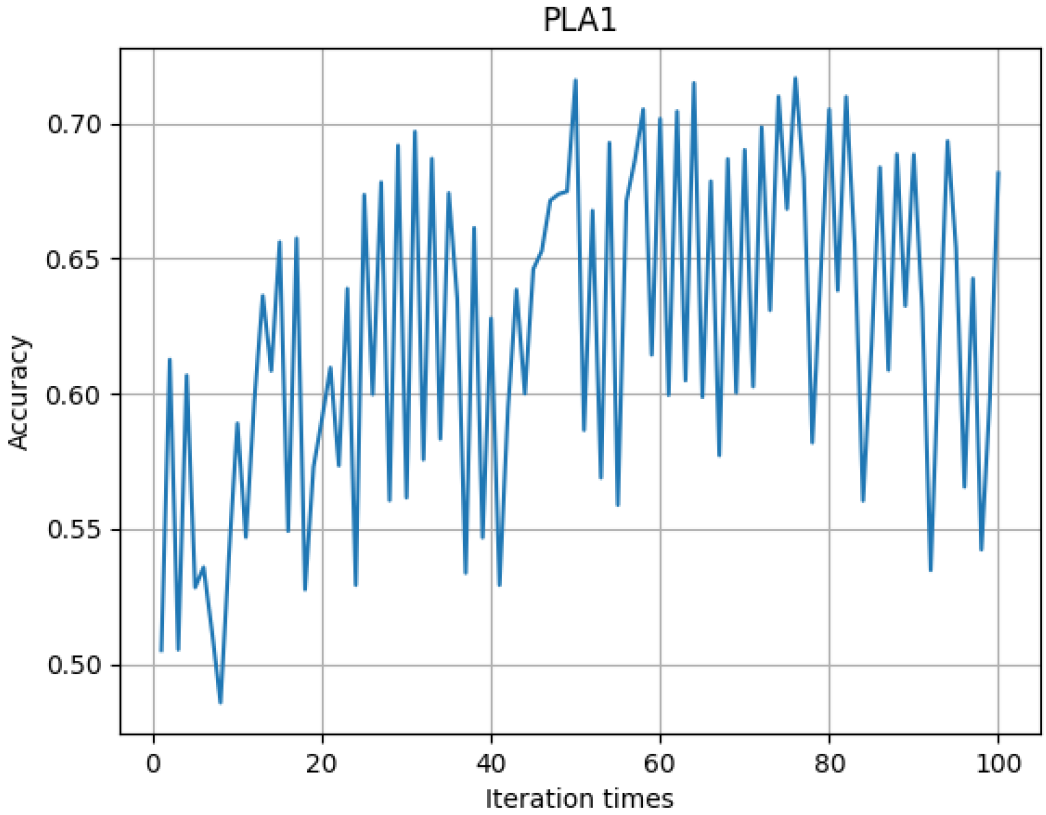
```
        judge = label[j] * (np.dot(w, data_mat[j]) + b)
        if judge <= 0:
            count += 1
    return count

def pocket_pla(data_mat, label, iteration, rate):
    """

    """
    w = np.zeros(len(data_mat[0]))  # 初始化w为0向量
    b = 0  # 初始化b为0
    best_wrong = len(data_mat) + 1
    i = 0

    while i < iteration:
        f = (np.dot(data_mat, w.T) + b) * label  # 矩阵运算，直接得到所有点的分类结果
        # 获取误分类点的位置索引
        idx = np.where(f <= 0)  # idx为一个元组 第一个元素为误分类点的行索引 第二个元素为列索引
        num = f[idx].size
        if num != 0:  # 若有误分类点
            point = np.random.randint((f[idx].shape[0]))  # 随机挑选一个误分类点
            temp_x = data_mat[idx[0][point]]  # 取出该行对应的x
            temp_y = label[idx[0][point]]  # 取出该行所对的label
            temp_w = w + rate * temp_x * temp_y
            temp_b = b + rate * temp_y
            temp_wrong = cal_wrong(temp_w, temp_b, data_mat, label) #计算新w的错误
            if temp_wrong < best_wrong: #若新的w犯错更少则进行更新
                best_wrong = temp_wrong
                w = temp_w
                b = temp_b
            i += 1
        else:break

    return w, b
```

二、 实验结果及分析

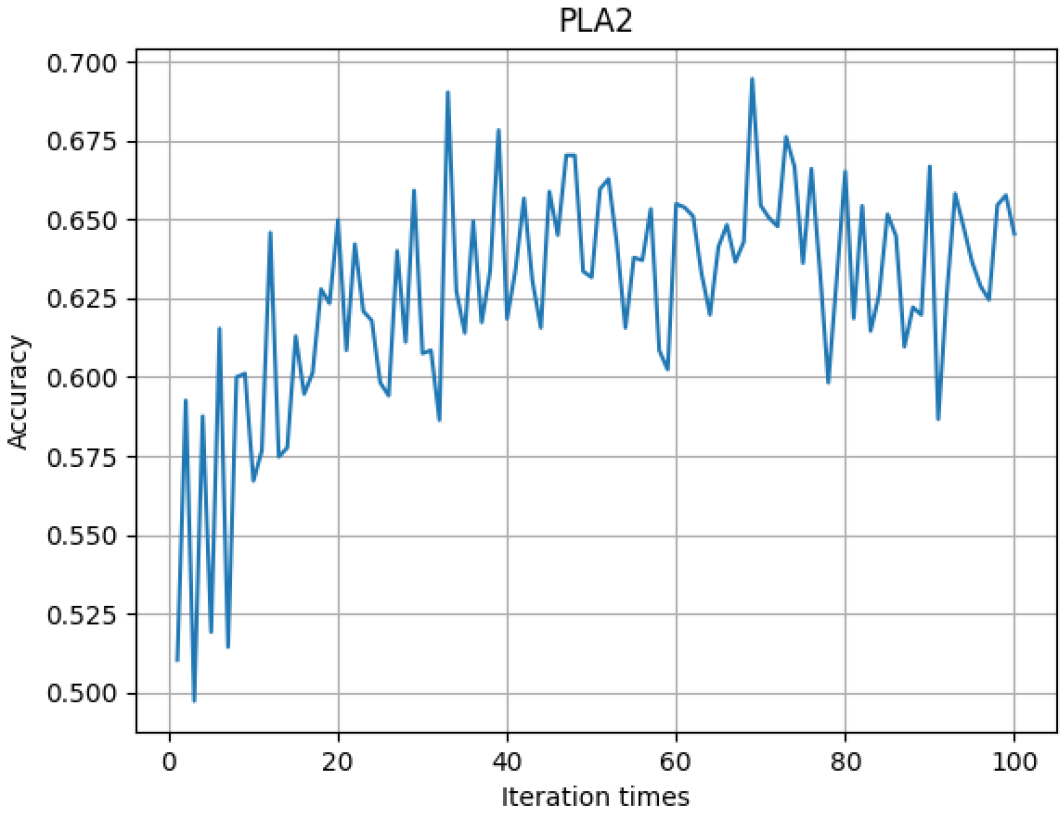
- 不同实现方法对比

取k-fold k值为10，学习率为0.1，训练模型，在验证集上测试准确率。

采用第一种感知机算法时，准确率随固定迭代次数的变化如下图所示。



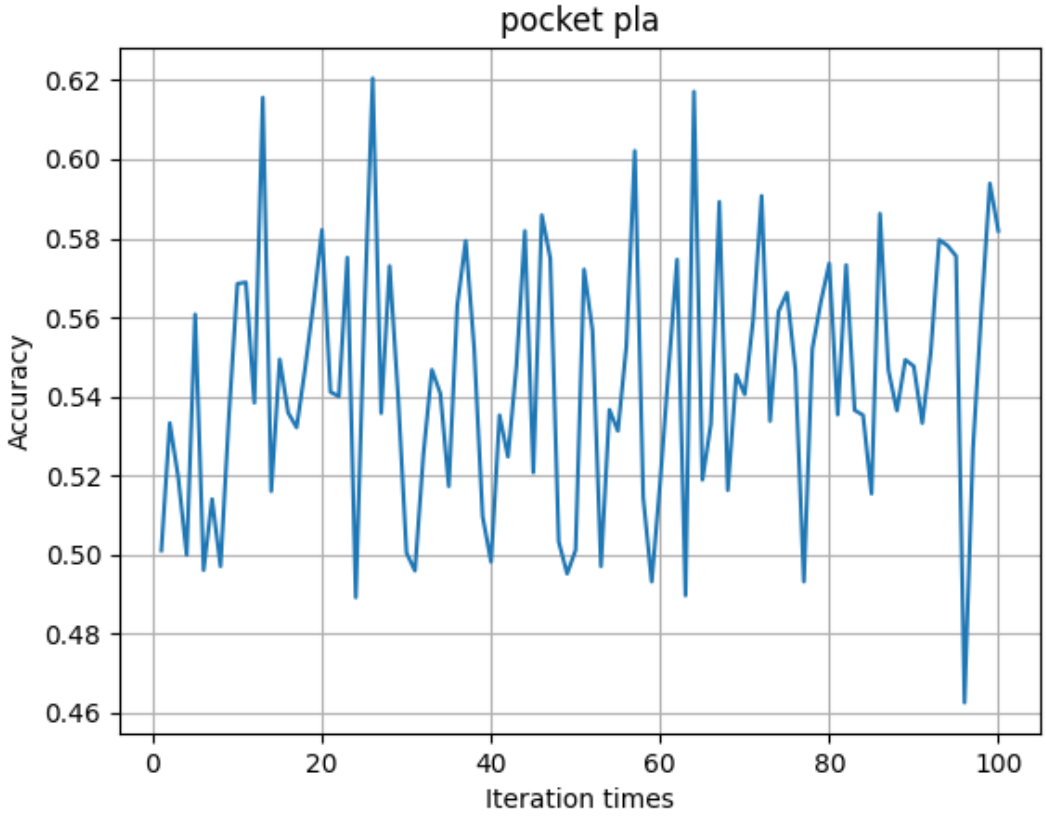
采用第二种感知机算法时，准确率随固定迭代次数的变化如下图所示。



可以发现两种实现方法在趋势和准确率上大体相同，没有显著差异，都是呈现出一个随迭代次数增多，准确率大体上升的一个趋势。最初我以为随机选取一个点更新的方法效果会更好，因为它更符合PLA随机梯度下降的定义。但是会发现，在一些迭代次数上。选择第一个误分类点的准确率会更高，我认为可能是一些随机因素导致的。

同时可见该数据集线性不可分，PLA并不能找到一个最优解，所以每次迭代更新后，准确率就会开始上下波动，可能上升也可能下降。但是会发现，随机选取一个点更新的方法其结果的波动性要小于第一种方法，随机选取点的方法避免了一些特例的产生。

采用口袋pla算法时，准确率随固定迭代次数的变化如下图所示。

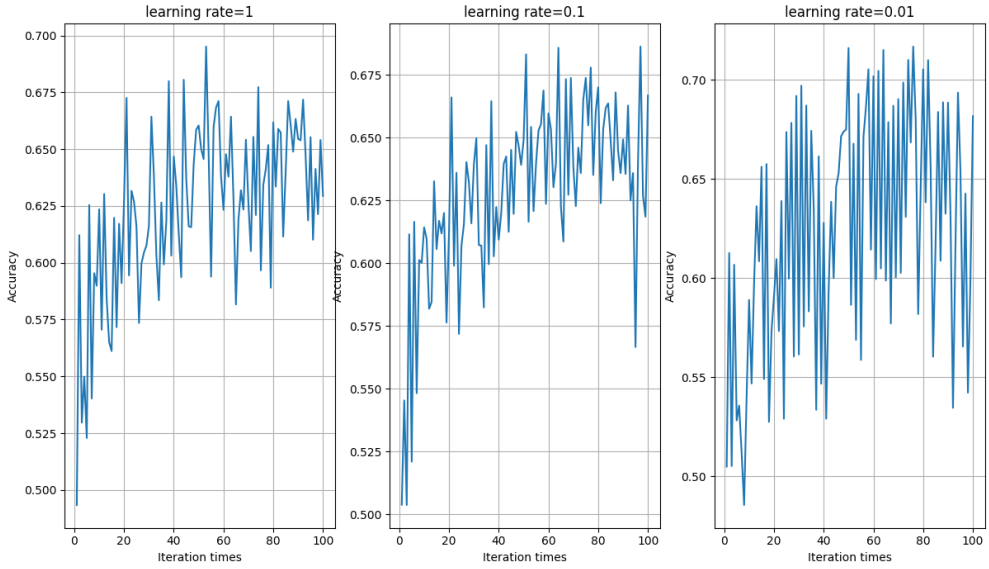


根据算法原理，直觉上我以为口袋PLA算法准确率会高于前两者，但却并没有。可能的原因是因为，每次都是随机选取一个误分类点更新出新线，但随机更新出的新线却不一定会比旧线更优，反而白白浪费了一次迭代机会，导致随机遇到更优线的概率更低了。此外，口袋pla算法每次更新出一条新线，就要计算一遍在所有样本点的错误情况，效率较低，运行时间大大超过前两种方法。可能迭代次数再加大一些会比较合适，但它运行的时间实在是太慢了。。。

- 不同学习率对比

取k-fold k值为10，学习率为分别为1、0.1、0.01，训练模型，在验证集上测试准确率，结果如下图所示。

可以发现这三种学习率在趋势和准确率上大体相同，没有显著差异。但是会发现学习率越小，其结果波动性越强。我个人推测是由于当学习率较小时，其收敛速度较慢，导致结果还没有收敛时就已达到最大迭代次数返回结果了。



任务二、 LR逻辑回归

一、 实验内容

1. 算法原理

- 逻辑回归虽然被称为回归，但实际上还是分类问题。它是一种概率模型，通常表示为概率分布形式，即先算出每个类别的概率，然后根据概率的大小来判断样本的类别。针对二分类问题时，输入的样本的特征向量$x \in R^n$，输出是样本属于某个类别$y \in \{0,1\}$的概率
- `logistic`函数
  - 在先前我们已经学过了线性回归，但是使用线性的函数来拟合规律后取阈值的办法是不准确的,原因在于拟合的函数太直,异常值对结果的影响将很大,所以我们想能不能引入一个非线性的函数用于拟合，便引入了`logistic`函数。
  - 为方便表示，将W表示为$(W^T, b)^T$，将x表示为$x = (x^T, 1)^T$

    在拟合时我们将原先的$y = W^T X$ 换为$\pi(z) = \frac{1}{1+e^{(-z)}}$，其中$z = W^T X$
  - 其输出的结果不再是预测结果，而是某个样本(x, y)，预测为类别y的概率，表示为

    $$f(x) = P(y|x) = \pi(x)^y(1-\pi(x)^{1-y})$$

    其中$f(x)$称为似然函数
- 似然函数

  在整个训练集上考虑似然函数

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

  对数似然函数为（似然函数取对数）

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

- 参数更新

  对$L(w)$取负，将$-L(w)$作为逻辑回归模型的损失函数，并使用批量梯度下降法对损失函数进行优化

  损失函数的梯度为

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

  对参数进行更新

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

- 算法步骤

  1. 将权重向量和阈值合并，即$w = (w^T, b)^T$，并为每个样本特征向量添加一维常数项1，即$x = (x^T, 1)^T$
  2. 设置学习率
  3. 计算当前梯度，并对参数w进行更新
  4. 重复步骤3，知道所有的训练样本都梯度收敛或到达固定最大迭代次数
  5. 训练结束后我们可以得到一个w的值，在验证集上进行评测，以验证集上的预测准确率作为评测指标。

2. 伪代码

```
Function LR(data_mat, label, iteration, rate):
//input:datamat:特征集合
//      label:标签集合
//      iteration:最大迭代次数
//      rate:学习率
//output:训练出的w

    w初始化为n+1维的0向量
    diff := 1e-4    //收敛条件
    data_temp := data_mat多拓展1维常数项1
    for i in range(iteration) do
        sum := 0
        for point in data_temp do
            sum += (label-sigmod(w, point)) * point //sigmod为逻辑回归函数
        end for
        w_new := w + rate * sum //参数更新
        if (distance(w_new,w) <= diff) then//满足收敛条件
            print("梯度下降收敛")
            break
         end if
     end for
        w := w_new
    return w
```

3. 关键代码展示

- 逻辑回归函数

  在这里要注意的是当点乘结果temp小于0时，$\pi(z)=\frac{1}{1+e^{(-temp)}}$，$e^{-temp}$ 结果非常容易上溢，从而导致结果下溢。当temp大于0时$e^{-temp}\in(0,1)$就不会有什么问题。

  所以在这里灵活选择$\pi(z)=\frac{1}{1+e^{(-temp)}}=\frac{e^{temp}}{1+e^{temp}}$这两个表达方式来避免 `exp` 运算带来的溢出问题。

```
def sigmod(w, x):
    """
    logistic函数
    :param w:
    :param x:
    :return:
    """
    temp = w.dot(x)
    #防止溢出
    if temp >= 0:
        temp = 1.0 / (1 + np.exp(-1 * temp))
    else:
        temp = np.exp(temp) / (1 + np.exp(temp))

    return temp

```

- 参数更新

```
def gradient(data_mat, label, iteration, rate):
    """
    计算梯度下降，进行w和b的参数更新
    :param data_mat:特征集合
    :param label:标签集合
    :param iteration:最大迭代次数
    :param rate:学习率
    :return:返回训练的w和b
    """
    w = np.zeros(len(data_mat[0]) + 1)  # 多拓展一维
    diff = 1e-4
    data_temp = np.insert(data_mat, data_mat.shape[1], 1, axis=1) #x多拓展一维
    for i in range(iteration):
        sum = 0
        for j in range(len(data_temp)):
            sum += (label[j]-sigmod(w, data_temp[j])) * data_temp[j]
        w_new = w + rate * sum
        diff1 = np.linalg.norm(w_new - w)  # 以两者的欧式距离为收敛条件判断依据
        if (diff1 <= diff):
            print("梯度下降收敛")
            break
        w = w_new
    return w
```

- 预测结果

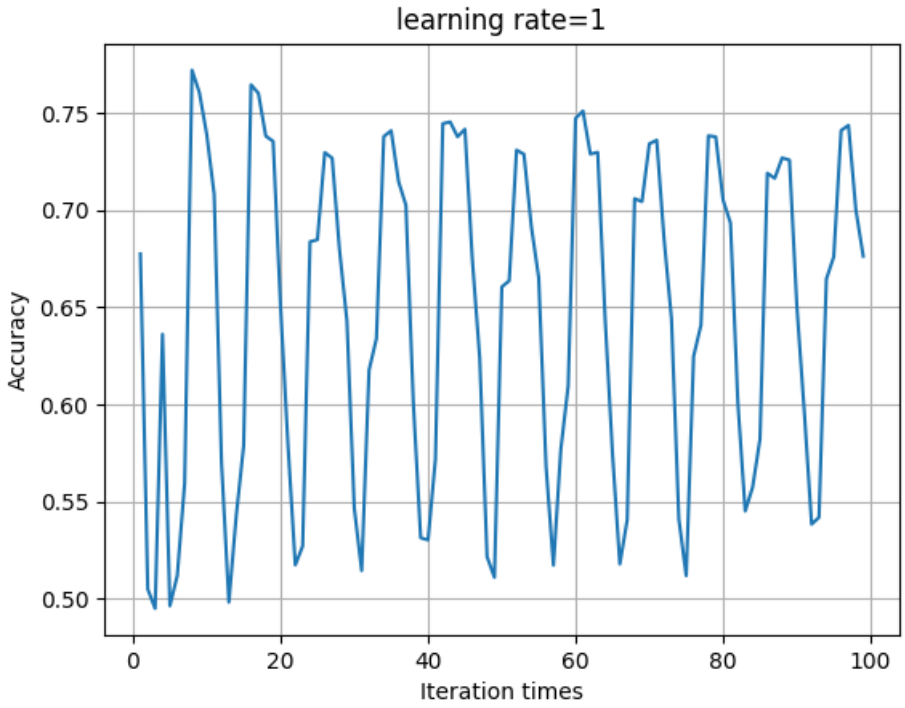
```
def predict(valid_mat, lable, w):
    total = len(valid_mat)
    cnt = 0
    valid_temp = np.insert(valid_mat, valid_mat.shape[1], 1, axis=1)
    for i in range(len(valid_temp)):
        temp=sigmod(w,valid_temp[i])
        if(temp>=0.5 and lable[i]==1) or (temp<0.5 and lable[i]==0):
            cnt+=1
    return cnt / total
```

二、 实验结果及分析

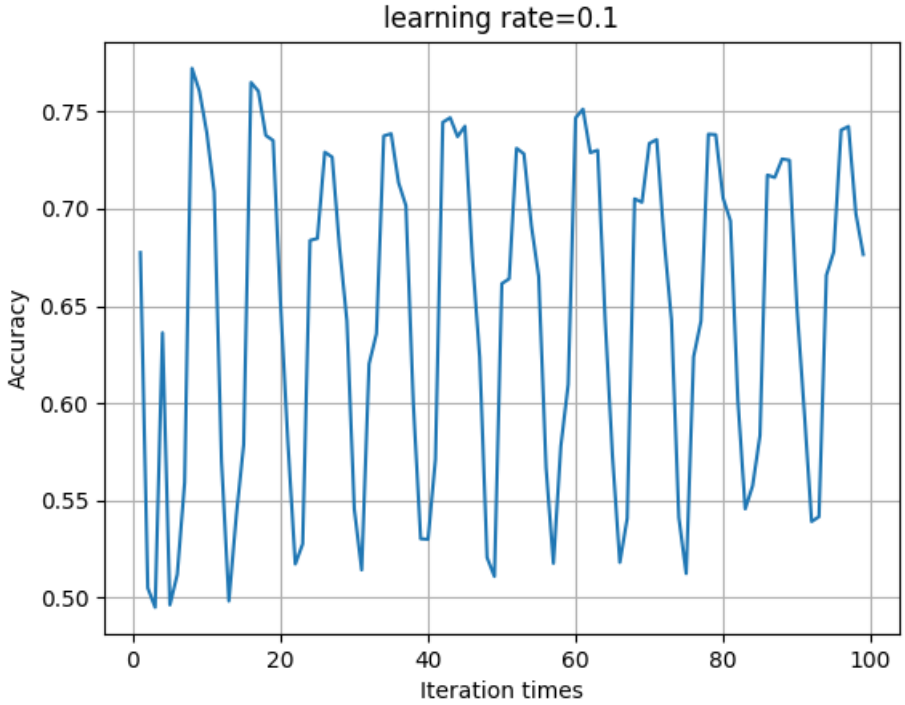
1. 结果展示

取k-fold k值为10，迭代次数为100次

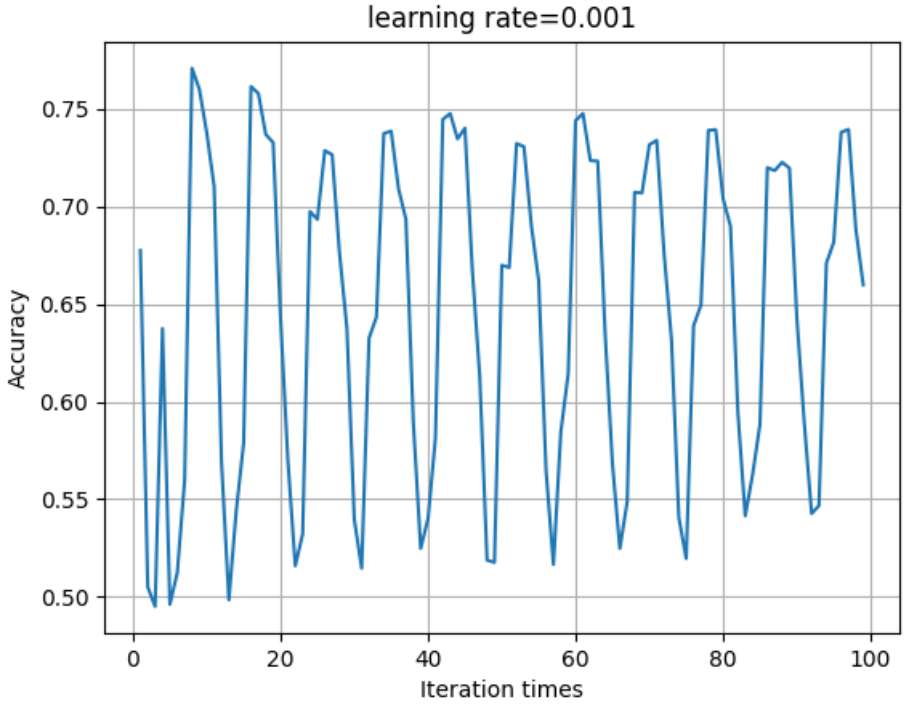
- 学习率为1时，准确率随固定迭代次数的变化如下图所示。



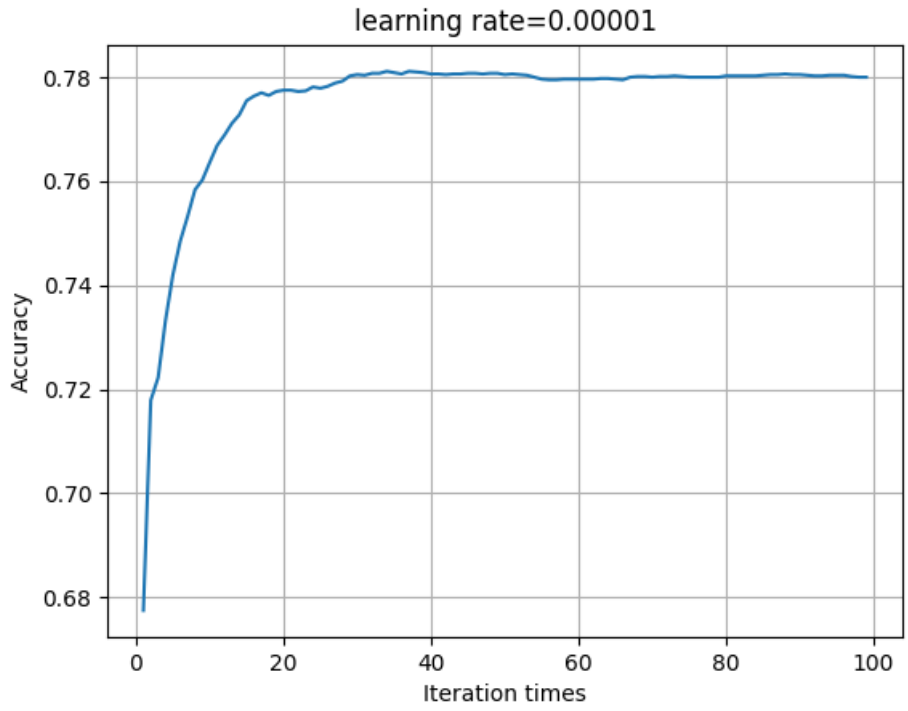
- 学习率为0.1时，准确率随固定迭代次数的变化如下图所示。



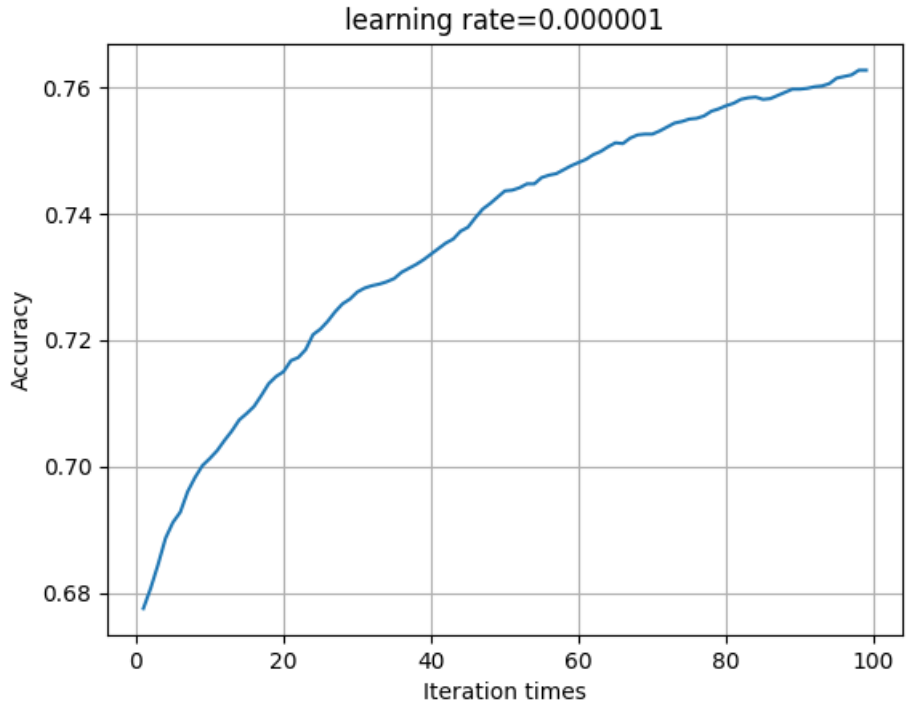
- 学习率为0.001时，准确率随固定迭代次数的变化如下图所示。



- 学习率为0.00001时，准确率随固定迭代次数的变化如下图所示。



- 学习率为0.000001时，准确率随固定迭代次数的变化如下图所示。



2. 结果分析

- 学习率

可以从学习率=1,0.1,0.001的图像结果上看到其准确率一直在波动。当学习率取较大时，会导致训练参数w在最优值附波动，从而导致准确率也在最优值附近震荡，达不到最优值。当把学习率调整到10^{-5}时，调整到了一个较合适的学习率，可以看到准确率先上升后平稳。

- 收敛速度

以学习率为0.001和0.00001对比，可以看到当学习率为0.001花费了更少的迭代次数达到了最优点。但是由于其学习率较大，导致结果波动。

以学习率为10^{-5}和10^{-6}对比，二者都属于可取的学习率，准确率都趋于平稳。但对比之下，10^{-6}的学习率又有点偏小，其收敛速度明显慢于10^{-5}：其迭代100次的准确率，后者在迭代小于20次便可以达到。所以可见，选取一个合适的学习率十分重要。

- 是否收敛

当收敛的阈值设为10^{-4}时，在以上学习率的实验过程中，都没有收敛。（在前面梯度下降的函数中，我设置了若收敛则打印“梯度下降收敛"）

当收敛的阈值设为10^{-3}时，取学习率为10^{-5}时，在第70次迭代左右会梯度收敛。本实验中的目标函数是一个凸函数，所以导致在优化过程中，很难到达最优点。

三、 思考题

- 随机梯度下降与批量梯度下降各自的优缺点?
  - 随机梯度下降

    随机梯度下降是每次迭代只使用一个样本点来对参数进行更新。由于每次迭代中，只是随机优化一条训练数据上的损失函数，这样每一轮参数更新的速度大大加快。

    但其缺点是准确率下降。即使目标函数是强凸函数的情况下，随机梯度下降的方法仍旧无法做到线性收敛。此外就是可能会收敛到局部最优解，但单个样本并不能代表全体样本的趋势，从而不一定会达到全局最优解。
  - 批量梯度下降

    批量梯度下降是每次迭代使用所有样本点来对参数进行更新。由全数据集确定的方向能够更好地代表样本总体，从而更准确地朝向极值所在的方向。当目标函数为凸函数时，批量梯度下降一定能够得到全局最优。此外，由于一次迭代是对所有样本进行计算，可以灵活运用矩阵进行操作，比那与实现并行。

    但其缺点是，当样本数目很大时由于每轮迭代都需要计算所有样本的损失函数，从而会导致训练消耗的时间较慢。

- 不同学习率η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

学习率较小时，权重向量更新较慢，收敛过程较为缓慢，可能需要更多的迭代次数来更新权重向量。当学习率较大时，权重向量更新快，能更快收敛，达到最优解。但是若学习率设置的太高，梯度可能在最小值附近来回震荡，甚至可能最终无法收敛。

- 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。

若逻辑回归的目标函数来是一个严格的凸函数，在优化过程中，一般很难到达最优点。在最优点附近的时候梯度很低，导致参数更新速度越来越慢，甚至可能会无限靠近最优点却无法到达最优点的情况发生。此时若学习率设置的不合理，可能又会出现不断震荡的情况。因此，将模长作为0作为收敛终止条件不合适。

一般以两次权重向量差值的二范数低于某个阈值作为收敛终止条件。