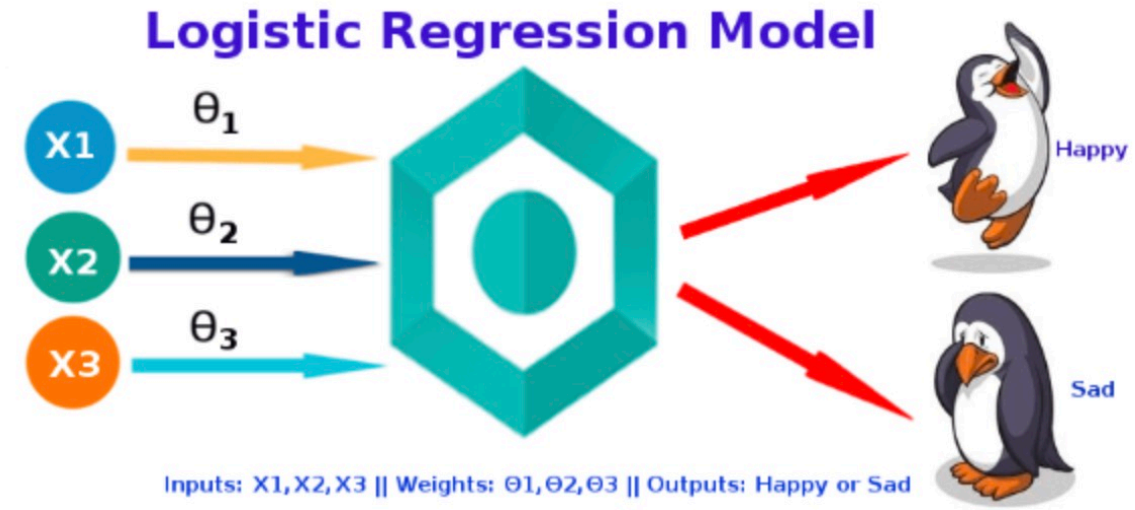
# 逻辑回归

# 一、概述

分类技术是机器学习和数据挖掘应用中的重要组成部分。在数据科学中，大约70%的问题属于分类问题。解决分类的算法也有很多种。比如：KNN，使用距离计算来实现分类；决策树，通过构建直观易懂的树来实现分类。这里我们要展开的是Logistic回归，它是一种很常见的用来解决二元分类问题的回归方法，它主要是通过寻找最优参数来正确地分类原始数据。

# 二、基本原理

逻辑回归（Logistic Regression，简称LR），其实是一个很有误导性的概念，虽然它的名字中带有 "回归" 两个字，但是它最擅长处理的却是分类问题。LR分类器适用于各项广义上的分类任务，例如：评论信息的正负情感分析（二分类）、用户点击率（二分类）、用户违约信息预测（二分类）、垃圾邮件预测（二分类）、疾病预测（二分类）、用户等级分类（多分类）等场景。我们这里主要讨论的是二分类问题。



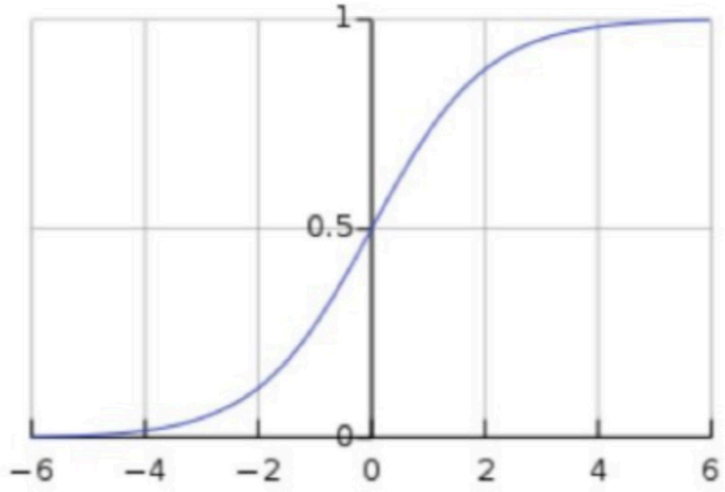

之前我们介绍了线性回归的算法基本原理、使用方法、在最小二乘法求最优解以及条件不满足的情况系原始算法的修正方法。但实际上，线性回归只是机器学习类算法中最简单的用特征预测标签数值的回归算法，满足线性规律的真实场景并不是很多，因此标准线性回归应用面有限。为了解决该问题，线性回归在实际应用中引入了诸多变化形式，而这些变化形式可统一规整为如下形式：

$$y = g^{-1}(w^T x + b)$$

其中 $g(*)$ 为可微函数。而这类模型也被称为广义线性模型（generalized linear model），其中函数 $g(*)$ 被称为联系函数（link function），现如今被广为人知的逻辑回归就是诸多广义回归算法的其中一种。在逻辑回归中，我们使用对数几率函数（Logistic function）作为 $g^{-1}(*)$，对数几率函数表示形式如下：

$$y = g(z) = \frac{1}{1 + e^{-z}}$$

能够看出，对数几率函数是一个Sigmoid函数。Sigmoid函数是形似S的函数，对率函数是Sigmoid函数的重要代表，在感知机理论中也发挥着重大作用。




利用对率函数，我们可将 $z$ 转化为一个（0，1）区间内的值，除此之外，$g(z)$ 还有一个很好的导数性质：

$$g'(z) = g(z)(1 - g(z))$$

带入对率函数，得到逻辑回归表达式为：

$$y = \frac{1}{1 + e^{-(w^T x + b)}}$$

进一步可得：

$$ln\frac{y}{1 - y} = w^T x + b$$

由于 $y$ 和 $1 - y$ 的和为1，因此可将 $y$ 和 $1 - y$ 视为一对正反例的可能性，即 $y$ 视作样本 $x$ 为正例的可能性，则 $1 - y$ 为 $x$ 为反例的可能性，二者比例：

$$\frac{y}{1 - y}$$

被称为 "几率"（odds），反映了样本 $x$ 为正例的相对可能性。对几率取对数则得到 "对数几率" （log odds，亦称logit）：

$$ln\frac{y}{1 - y}$$

由此看出，上式实际上是在用线性回归模型的预测结果取逼近真实标记的对数几率。因此，其对应的模型被称为 "对数几率回归"（logistic regression）。需要注意的是，虽然其名字包含回归二字，但本质上是一种分类学习方法。这种方法有很多优点，例如它是直接对分类可能性进行建模，因此它不仅预测出 "类别"，而且得到的是近似概率预测，这对许多需要利用概率辅助决策的任务很有用。同时，对率函数是任意阶可导的凸函数，有很多的数学性质，现有的很多数值优化算法都可直接用于求取最优解。接下来我们将采用梯度下降的方法对其进行求解，首先我们将对梯度下降理论进行简单介绍。

下面给出逻辑回归的推导公式

$$
\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j} \frac{-1}{m} \sum_{i=1}^{m} \left[ y^{(i)} \log\left(h_\theta\left(x^{(i)}\right)\right) + \left(1-y^{(i)}\right) \log\left(1-h_\theta\left(x^{(i)}\right)\right) \right] \\
&\underset{\text{linearity}}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[ y^{(i)} \frac{\partial}{\partial \theta_j} \log\left(h_\theta\left(x^{(i)}\right)\right) + \left(1-y^{(i)}\right) \frac{\partial}{\partial \theta_j} \log\left(1-h_\theta\left(x^{(i)}\right)\right) \right] \\
&\underset{\text{chain rule}}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[ y^{(i)} \frac{\frac{\partial}{\partial \theta_j} h_\theta\left(x^{(i)}\right)}{h_\theta\left(x^{(i)}\right)} + \left(1-y^{(i)}\right) \frac{\frac{\partial}{\partial \theta_j}\left(1-h_\theta\left(x^{(i)}\right)\right)}{1-h_\theta\left(x^{(i)}\right)} \right] \\
&\underset{h_\theta(x)=\sigma\left(\theta^\top x\right)}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[ y^{(i)} \frac{\frac{\partial}{\partial \theta_j} \sigma\left(\theta^\top x^{(i)}\right)}{h_\theta\left(x^{(i)}\right)} + \left(1-y^{(i)}\right) \frac{\frac{\partial}{\partial \theta_j}\left(1-\sigma\left(\theta^\top x^{(i)}\right)\right)}{1-h_\theta\left(x^{(i)}\right)} \right] \\
&\underset{\sigma'}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[ y^{(i)} \frac{\sigma\left(\theta^\top x^{(i)}\right)\left(1-\sigma\left(\theta^\top x^{(i)}\right)\right) \frac{\partial}{\partial \theta_j}\left(\theta^\top x^{(i)}\right)}{h_\theta\left(x^{(i)}\right)} - \left(1-y^{(i)}\right) \frac{\sigma\left(\theta^\top x^{(i)}\right)\left(1-\sigma\left(\theta^\top x^{(i)}\right)\right) \frac{\partial}{\partial \theta_j}\left(\theta^\top x^{(i)}\right)}{1-h_\theta\left(x^{(i)}\right)} \right] \\
&\underset{\sigma\left(\theta^\top x\right)=h_\theta(x)}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[ y^{(i)} \frac{h_\theta\left(x^{(i)}\right)\left(1-h_\theta\left(x^{(i)}\right)\right) \frac{\partial}{\partial \theta_j}\left(\theta^\top x^{(i)}\right)}{h_\theta\left(x^{(i)}\right)} - \left(1-y^{(i)}\right) \frac{h_\theta\left(x^{(i)}\right)\left(1-h_\theta\left(x^{(i)}\right)\right) \frac{\partial}{\partial \theta_j}\left(\theta^\top x^{(i)}\right)}{1-h_\theta\left(x^{(i)}\right)} \right] \\
&\underset{\frac{\partial}{\partial \theta_j}\left(\theta^\top x^{(i)}\right)=x_j^{(i)}}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[ y^{(i)} \left(1-h_\theta\left(x^{(i)}\right)\right) x_j^{(i)} - \left(1-y^{i}\right)\ h_\theta\left(x^{(i)}\right) x_j^{(i)} \right] \\
&\underset{\text{distribute}}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[ y^{i} - y^{i} h_\theta\left(x^{(i)}\right) - h_\theta\left(x^{(i)}\right) + y^{(i)} h_\theta\left(x^{(i)}\right) \right] x_j^{(i)} \\
&\underset{\text{cancel}}{=} \frac{-1}{m} \sum_{i=1}^{m} \left[ y^{(i)} - h_\theta\left(x^{(i)}\right) \right] x_j^{(i)} \\
&= \frac{1}{m} \sum_{i=1}^{m} \left[ h_\theta\left(x^{(i)}\right) - y^{(i)} \right] x_j^{(i)}
\end{aligned}
$$

# 三、梯度下降（Gradient Descent）

在求解机器学习算法的模型参数，即无约束优化问题时，梯度下降（Gradient Descent）是最常用的方法之一，接下来就对梯度下降进行介绍。

## 1. 梯度

在微积分里面，对多元函数的参数求 $\partial$ 偏导数，把求得的各个参数的偏导数以向量的形式写出来，就是梯度。比如函数 $f(x,y)$ ，分别对 $x,y$ 求偏导数，求得的梯度下向量就是 $(\partial f/\partial x, \partial f/\partial y)^T$ ，简称 $grad\ \ f(x,y)$ 或者 $\nabla f(x,y)$ 。对于在点 $(x_0,y_0)$ 的具体梯度向量就是 $(\partial f/\partial x0, \partial f/\partial y0)^T$ 或者 $\nabla f(x_0,y_0)$ ，如果是三个参数的向量梯度，就是 $(\partial f/\partial x, \partial f/\partial y, \partial f/\partial z)^T$ ，以此类推。

那么这个梯度向量求出来有什么意义呢？它的意义从几何意义上讲，就是函数变化增加最快的地方。具体来说，对于函数 $f(x,y)$ 在点 $(x_0,y_0)$ ，沿着梯度向量的方向就是 $(\partial f/\partial x0, \partial f/\partial y0)^T$ 的方向是 $f(x,y)$ 增加最快的方向。或者说，沿着梯度向量的方向，更加容易找到函数的最大值。反过来说，沿着梯度向量相反的方向，也就是 $-(\partial f/\partial x0, \partial f/\partial y0)^T$ 的方向，梯度减少最快，也就是更加容易找到函数的最小值。

## 2. 梯度下降和梯度上升

在机器学习算法中，在最小化损失函数时，可以通过梯度下降法来一步步的迭代求解，得到最小化的损失函数，和模型参数值。反过来，如果我们需要求解损失函数的最大值，这时就需要用梯度上升法来迭代了。梯度下降和梯度上升法是可以互相转化的。比如我们需要求解损失函数 $f(\theta)$ 的最小值，这时我们需要用梯度下降来迭代求解。但是实际上，我们可以反过来求解损失函数 $-f(\theta)$ 的最大值，这时梯度上升法就派上用场了。

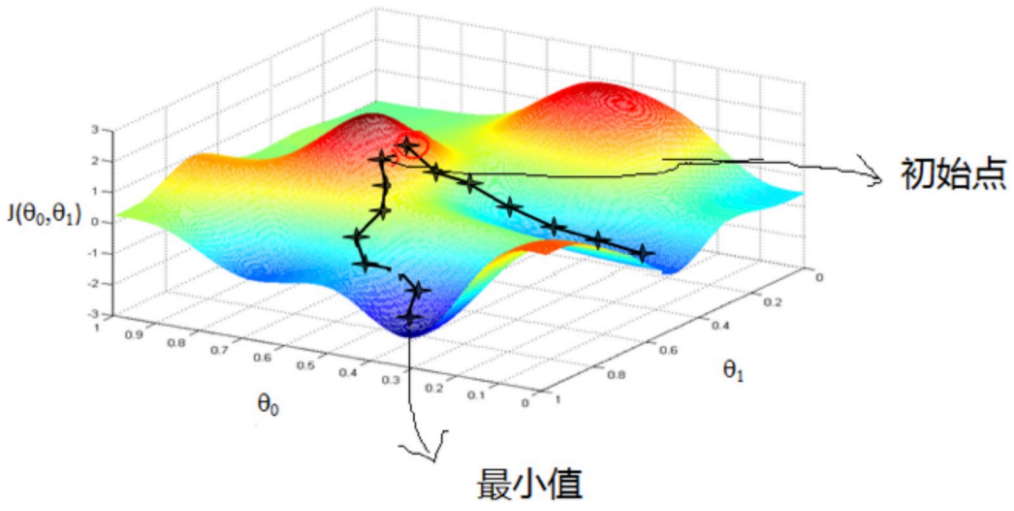
下面来详细总结下梯度下降法。

## 3. 梯度下降算法详解

### 3.1 梯度下降的直观解释

首先来看看梯度下降的一个直观的解释。比如我们在一座大山上的某处位置，由于我们不知道怎么下山，于是决定走一步算一步，也就是在每走到一个位置的时候，求解当前位置的梯度，沿着梯度的负方向，也就是当前最陡峭的位置向下走一步，然后继续求解当前位置梯度，向这一步所在位置沿着最陡峭最易下山的位置走一步。这样一步步的走下去，一直走到觉得我们已经到了山脚。当然这样走下去，有可能我们不能走到山脚，而是到了某一个局部的山峰低处。

从上面的解释可以看出，梯度下降不一定能够找到全局的最优解，有可能是一个局部最优解。当然，如果损失函数是凸函数，梯度下降法得到的解就一定是全局最优解。




### 3.2 梯度下降的相关概念

在详细了解梯度下降的算法之前，我们先看看相关的一些概念。

- 步长（Learning rate）：步长决定了在梯度下降迭代的过程中，每一步沿梯度负方向前进的长度。用上面下山的例子，步长就是在当前这一步所在位置沿着最陡峭最易下山的位置走的那一步的长度。

- 假设函数（hypothesis function）：在监督学习中，为了拟合输入样本，而使用的假设函数记为 $\hat{y}$ 。比如对于单个特征的m个样本，可以采用拟合函数为：$\hat{y} = w_0 + w_1 x$ 。
- 损失函数（loss function）：为了评估模型拟合的好坏，通常用损失函数来度量拟合的程度。损失函数极小化，意味着拟合程度最好，对应的模型的参数即为最优参数。在线性回归中，损失函数通常为样本输出和假设函数的差去平方。比如对于m个样本，采用线性回归，损失函数为：

$$J(w_0, w_1) = \sum_{i=1}^{m} (\hat{y_i} - y_i)^2$$

### 3.3 梯度下降的详细算法

梯度下降的算法用矩阵法来表示，更加简洁，且由于使用了矩阵，实现逻辑更加的一目了然。这一节要求有一定的矩阵分析的基础知识，尤其是矩阵求导的知识。

1. 先决条件：需要确认优化模型的假设函数和损失函数。对于线性回归，假设函数 $\hat{y} = w_0 + w_1 x_1 + \ldots + w_n x_n$ 矩阵表达方式为：

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w}$$

   其中，假设函数为$m \times 1$ 的向量，$w$ 是$( n+1) \times 1$ 的向量，里面有$n$ 个模型参数。$\boldsymbol{X}$ 为 $m \times (n+1)$ 维的矩阵。$m$ 代表样本的个数，$n+1$ 代表样本的特征数。损失函数的表达式为：

$$J(w) = \frac{1}{2m}(Xw - Y)^T (Xw - Y)$$

   其中 $Y$ 是样本的输出向量，维度为 $m \times 1$ 。

2. 算法相关参数初始化：$w$ 向量可以初始化为默认值，或者调优后的值。算法终止距离 $\varepsilon$ ，步长 $\alpha$ 初始化为1，在调优时再进行优化。
3. 算法过程：

   1）确定当前位置的损失函数的梯度，对于 $w$ 向量，其梯度表达式如下：

$$\frac{\partial}{\partial w} J(w)$$

   2）用步长乘以损失函数的梯度，得到当前位置下降的距离，即

$$\alpha \frac{\partial}{\partial w} J(w)$$

对应于前面登山例子中的某一步。

3）确定 $w$ 向量里面的每个值，梯度下降的距离都小于 $\varepsilon$ ，如果小于 $\varepsilon$ 则算法 终止，当前 $w$ 向量即为最终结果。否则进入步骤4。

4）更新 $w$ 向量，其更新表达式如下。更新完毕后继续转入后续步骤1。

$$w = w - \alpha \frac{\partial}{\partial w} J(w)$$

然后我们还是用线性回归的例子来描述具体的算法过程，损失函数对于 $w$ 向量的偏导数计算如下：

$$\frac{\partial}{\partial w} J(w) = \frac{1}{m} X^T (Xw - Y)$$

步骤4中 $w$ 向量的更新表达式如下：

$$w = w - \alpha X^T (Xw - Y)/m$$

可以看出矩阵法非常简洁。这里用到了矩阵求导链式法则，和两个矩阵求导的公式。

$$公式1: \frac{\partial}{\partial X}(XX^T) = 2X$$
$$公式2: \frac{\partial}{\partial w}(Xw) = X^T$$

## 3.4 梯度下降的算法调优

在使用梯度下降时，需要进行调优。哪些地方需要调优呢?

- **算法的步长选择。** 在前面的算法描述中，提到取步长为1，但是实际上取值取决于数据样本，可以多取一些值，从大到小，分别运行算法，看看迭代效果，如果损失函数在变小，说明取值有效，否则要增大步长。
  - 步长太大，会导致迭代过快，甚至有可能错过最优解。
  - 步长太小，迭代速度太慢，很长时间算法都不能结束。

  所以算法的步长需要多次运行后才能得到一个较为优的值。
- **算法参数的初始值选择。**初始值不同，获得的最小值也有可能不同，因此梯度下降求得的只是局部最小值；当然如果损失函数是凸函数则一定是最优解。由于有局部最优解的风险，需要多次用不同的初始值运行算法，观测损失函数的最小值，选择损失函数最小化的初始值。
- **标准化。**由于样本不同特征的取值范围不一样，可能导致迭代很慢，为了减少特征取值的影响，可以对特征数据标准化，也就是对于每个特征 $x$ ，求出它的期望 $\bar{x}$ 和标准差 $std(x)$ ，然后转化为：

$$\frac{x-\bar{x}}{std(x)}$$

这样特征的新期望为0，新方差为1，且无量纲，收敛速度可以大大加快。

# 四、逻辑回归的Scikit-Learn实现

## 1. 参数讲解

*class* `sklearn.linear_model.LogisticRegression`(*penalty='l2'*, *dual=False*, *tol=0.0001*, *C=1.0*, *fit_intercept=True*, *intercept_scaling=1*, *class_weight=None*, *random_state=None*, *solver='warn'*, *max_iter=100*, *multi_class='warn'*, *verbose=0*, *warm_start=False*, *n_jobs=None*)

### 1.1 正则化参数penalty

LogisticRegression默认就带了正则化项。penalty参数可选择的值为 "l1"和 "l2"，分别对应L1的正则化和L2的正则化，默认是L2的正则化。

在调参时如果我们主要的目的只是为了解决过拟合，一般penalty选择L2正则化就够了。但是如果选择L2正则化发现还是过拟合，即预测效果差的时候，就可以考虑L1正则化。另外，如果模型的特征非常多，我们希望一些不重要的特征系数归零，从而让模型系数稀疏化的话，也可以使用L1正则化。

#### 1.1.1 L1和L2正则化的直观解释

**为什么L1正则化可以产生稀疏模型（L1是怎么让系数等于零的），以及为什么L2正则化可以防止过拟合?**

- L1正则化和特征选择

  假设有如下带L1正则化的损失函数：

$$J = J_0 + \alpha \sum_w |w|$$

  其中$J_0$ 是原始的损失函数，第二项是L1正则化项，$\alpha$ 是正则化系数。我们注意到L1正则化是权值的绝对值之和，$J$ 是带有绝对值符号的函数，因此 $J$ 是不完全可微的。我们的任务是通过一些方法（比如梯度下降）求出损失函数的最小值。当我们在原始的损失函数后添加L1正则化项时，相当于对 $J_0$ 做了一个约束。令$L = \alpha \sum_w |w|$，则$J = J_0 + L$ ，此时我们的任务变成在L约束条件下求出$J_0$ 取最小值的解。考虑二维的情况，即只有两个权值 $w_1$ 和$w_2$ ，此时 $L = |w_1| + |w_2|$ 对于梯度下降法，求解 $J_0$ 的过程可以画出等值线，同时L1正则化的函数L也可以在 $w_1, w_2$ 的二维平

面上画出来。

图中等值线是$J_0$ 的等值线，黑色方形是L函数的图形。在图中，当等值线与L图形首次相交的地方就是最优解。上图中$J_0$ 与L在L的一个顶点处相交，这个顶点就是最优解。注意这个顶点的值是$(w_1, w_2) = (0, w)$ 。可以直观想到，因为L函数有很多“突出的角”（二维情况下有四个，多维情况下更多），$J_0$ 与这些角接触的几率会远大于与L其他部位接触的几率，而在这些角上，会有很多权值等于0，这就是为什么L1正则化可以产生稀疏模型，进而可以用于特征选择。

而正则化的系数$\alpha$ ，可以控制L图形的大小。$\alpha$ 越小，L的图形越大（黑色方框）；$\alpha$ 越大，L的图形就越小，可以小到黑色方框值超出原点一点点的范围，这时最优点的值可以取到很小的值。

- L2正则化

  类似的，假设有如下带L2正则化的损失函数：

$$J = J_0 + \alpha \sum_w w^2$$

同样可以画出其在二维平面上的图形，如下：

二维平面下L2正则化的函数图形是个圆，与方形相比，被磨去了棱角。因此$J_0$ 与L相交时使得$w_1, w_2$ 等于零　　的几率小了许多，这就是为什么L2正则化不具有稀疏性的原因。

#### 1.1.2 参数选择

penalty参数的选择会影响我们损失函数优化算法的选择。即参数solver的选择，如果是L2正则化，那么4种可选的算法{‘newton-cg’, ‘lbfgs’, ‘liblinear’, ‘sag’}都可以选择。但是**如果penalty是L1正则化的话，就只能选择‘liblinear’了**。这是因为L1正则化的损失函数不是连续可导的，而{‘newton-cg’, ‘lbfgs’,‘sag’}这三种优化算法时都需要损失函数的一阶或者二阶连续导数。 而‘liblinear’并没有这个依赖。

而两种正则化下C的取值，都可以通过学习曲线来进行调整。

建立两个逻辑回归，L1正则化和L2正则化的差别就一目了然了：

```
from sklearn.linear_model import LogisticRegression as LR
from sklearn.datasets import load_breast_cancer
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

data = load_breast_cancer()
X = data.data
```

```
y = data.target

data.data.shape

lrl1 = LR(penalty="l1",solver="liblinear",C=0.5,max_iter=1000)

lrl2 = LR(penalty="l2",solver="liblinear",C=0.5,max_iter=1000)

#逻辑回归的重要属性coef_，查看每个特征所对应的参数
lrl1 = lrl1.fit(X,y)
lrl1.coef_

(lrl1.coef_ != 0).sum(axis=1)

lrl2 = lrl2.fit(X,y)
lrl2.coef_
```

可以看见，当我们选择L1正则化的时候，许多特征的参数都被设置为了0，这些特征在真正建模的时候，就不会出现在我们的模型当中了，而L2正则化则是对所有的特征都给出了参数。

究竟哪个正则化的效果更好呢？还是都差不多？

```
l1 = []
l2 = []
l1test = []
l2test = []

Xtrain, Xtest, Ytrain, Ytest =
train_test_split(X,y,test_size=0.3,random_state=420)

for i in np.linspace(0.05,1,19):
    lrl1 = LR(penalty="l1",solver="liblinear",C=i,max_iter=1000)
    lrl2 = LR(penalty="l2",solver="liblinear",C=i,max_iter=1000)

    lrl1 = lrl1.fit(Xtrain,Ytrain)
    l1.append(accuracy_score(lrl1.predict(Xtrain),Ytrain))
    l1test.append(accuracy_score(lrl1.predict(Xtest),Ytest))

    lrl2 = lrl2.fit(Xtrain,Ytrain)
    l2.append(accuracy_score(lrl2.predict(Xtrain),Ytrain))
    l2test.append(accuracy_score(lrl2.predict(Xtest),Ytest))

graph = [l1,l2,l1test,l2test]
color = ["green","black","lightgreen","gray"]
label = ["L1","L2","L1test","L2test"]

plt.figure(figsize=(6,6))
for i in range(len(graph)):
    plt.plot(np.linspace(0.05,1,19),graph[i],color[i],label=label[i])
```

```
plt.legend(loc=4) #图例的位置在哪里?4表示，右下角
plt.show()
```

可见，至少在我们的乳腺癌数据集下，两种正则化的结果区别不大。但随着C的逐渐变大，正则化的强度越来越小，模型在训练集和测试集上的表现都呈上升趋势，直到C=0.8左右，训练集上的表现依然在走高，但模型在未知数据集上的表现开始下跌，这时候就是出现了过拟合。我们可以认为，C设定为0.8会比较好。在实际使用时，基本就默认使用L2正则化，如果感觉到模型的效果不好，那就换L1试试看。

## 1.2 算法优化参数solver

solver参数决定了我们对逻辑回归损失函数的优化方法，有4种算法可以选择，分别是:

- liblinear：使用了开源的liblinear库实现，内部使用了坐标轴下降法来迭代优化损失函数。
- lbfgs：拟牛顿法的一种，利用损失函数二阶导数矩阵即海森矩阵来迭代优化损失函数。
- newton-cg：也是牛顿法家族的一种，利用损失函数二阶导数矩阵即海森矩阵来迭代优化损失函数。
- sag：即随机平均梯度下降，是梯度下降法的变种，和普通梯度下降法的区别是每次迭代仅用一部分的样本来计算梯度，适合于样本数据多的时候。

从上面面的描述可以看出，newton-cg, lbfgs和sag这三种优化算法时都需要损失函数的一阶或者二阶连续导数，因此不能用于没有连续导数的L1正则化，只能用于L2正则化。而liblinear通吃L1正则化和L2正则化。

同时，sag每次仅使用了部分样本进行梯度迭代，所以当样本量少的时候不要选择它，而如果样本量非常大，比如大于10万，sag是第一选择。但是sag不能用于L1正则化，所以当你有大量的样本，又需要L1正则化的话就要自己做取舍了。要么通过对样本采样来降低样本量量，要么回到L2正则化。

从上面的描述，大家可能觉得，既然newton-cg, lbfgs和sag这么多限制，如果不是大样本，我们选择liblinear不就行了嘛！错，因为liblinear也有自己的弱点！我们知道，逻辑回归有二元逻辑回归和多元逻辑回归。对于多元逻辑回归常见的有one-vs-rest(OvR)和many-vs-many(MvM)两种，而MvM一般比OvR分类相对准确一些。郁闷的是liblinear只支持OvR，不支持MvM，这样如果我们需要相对精确的多元逻辑回归时，就不能选择liblinear了。也意味着如果我们需要相对精确的多元逻辑回归不能使用L1正则化了。

## 1.3 梯度下降：重要参数max_iter

逻辑回归的数学目的是求解能够让模型最优化，拟合程度最好的参数$w$的值，即求解能够让损失函数$J(w)$最小化的$w$值。对于二元逻辑回归来说，有多种方法可以用来求解参数$w$，最常见的有梯度下降法(Gradient Descent)，坐标下降法(Coordinate Descent)，牛顿法(Newton-Raphson method)等，其中又以梯度下降法最为著名。每种方法都涉及复杂的数学原理，但这些计算在执行的任务其实是类似的。

来看看乳腺癌数据集下，max_iter的学习曲线：

```
l2 = []
l2test = []

Xtrain, Xtest, Ytrain, Ytest =
train_test_split(X,y,test_size=0.3,random_state=420)

for i in np.arange(1,201,10):
    lrl2 = LR(penalty="l2",solver="liblinear",C=0.9,max_iter=i)
    lrl2 = lrl2.fit(Xtrain,Ytrain)
    l2.append(accuracy_score(lrl2.predict(Xtrain),Ytrain))
    l2test.append(accuracy_score(lrl2.predict(Xtest),Ytest))

graph = [l2,l2test]
color = ["black","gray"]
label = ["L2","L2test"]

plt.figure(figsize=(20,5))
for i in range(len(graph)):
    plt.plot(np.arange(1,201,10),graph[i],color[i],label=label[i])
plt.legend(loc=4)
plt.xticks(np.arange(1,201,10))
plt.show()

#我们可以使用属性.n_iter_来调用本次求解中真正实现的迭代次数

lr = LR(penalty="l2",solver="liblinear",C=0.9,max_iter=300).fit(Xtrain,Ytrain)
lr.n_iter_
```

当max_iter中限制的步数已经走完了，逻辑回归却还没有找到损失函数的最小值，参数$w$的值还没有被收敛，sklearn就会弹出这样的红色警告：

当参数solver="liblinear"：

```
C:\Python\lib\site-packages\sklearn\svm\base.py:922: ConvergenceWarning: L
iblinear failed to converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
```

当参数solver="sag"：

```
C:\Python\lib\site-packages\sklearn\linear_model\sag.py:334: ConvergenceWa
rning: The max_iter was reached which means the coef_ did not converge
  "the coef_ did not converge", ConvergenceWarning)
```

虽然写法看起来略有不同，但其实都是一个含义，这是在提醒我们：参数没有收敛，请增大max_iter中输入的数字。但我们不一定要听sklearn的。max_iter很大，意味着步长小，模型运行得会更加缓慢。虽然我们在梯度下降中追求的是损失函数的最小值，但这也可能意味着我们的模型会过拟合（在训练集上表现得太好，在测试集上却不一定），因此，如果在max_iter报红条的情况下，模型的训练和预测效果都已经不错了，那我们就不需要再增大max_iter中的数目了，毕竟一切都以模型的预测效果为基准——只要最终的预测效果好，运行又快，那就一切都好，无所谓是否报红色警告了。

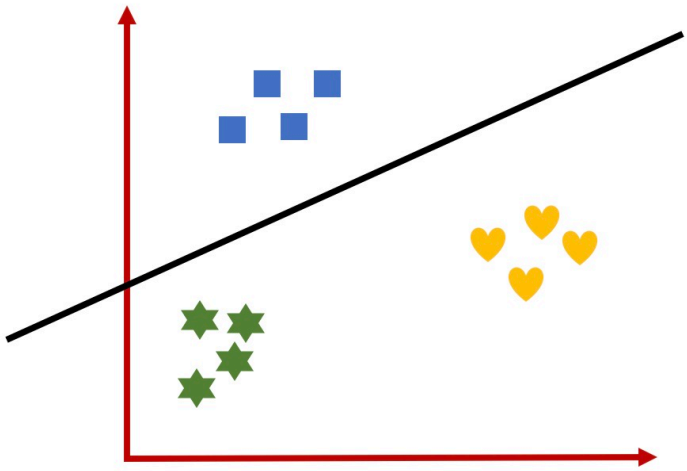
## 1.4 分类方式选择参数

multi_class参数决定了我们分类方式的选择，有ovr和multinomial两个值可以选择，默认是ovr。

ovr即前面提到的one-vs-rest(OvR)，而multinomial即前面提到的many-vs-many(MvM)。如果是二元逻辑回归，ovr和multinomial并没有任何区别，区别主要在多元逻辑回归上。

OvR的思想很简单，无论你是多少元逻辑回归，我们都可以看做二元逻辑回归。具体做法是，对于第K类的分类决策，我们把所有第K类的样本作为正例，除了第K类样本以外的所有样本都作为负例，然后在上面做二元逻辑回归，得到第K类的分类模型。其他类的分类模型获得以此类推。

- **生成三个假的数据集：**

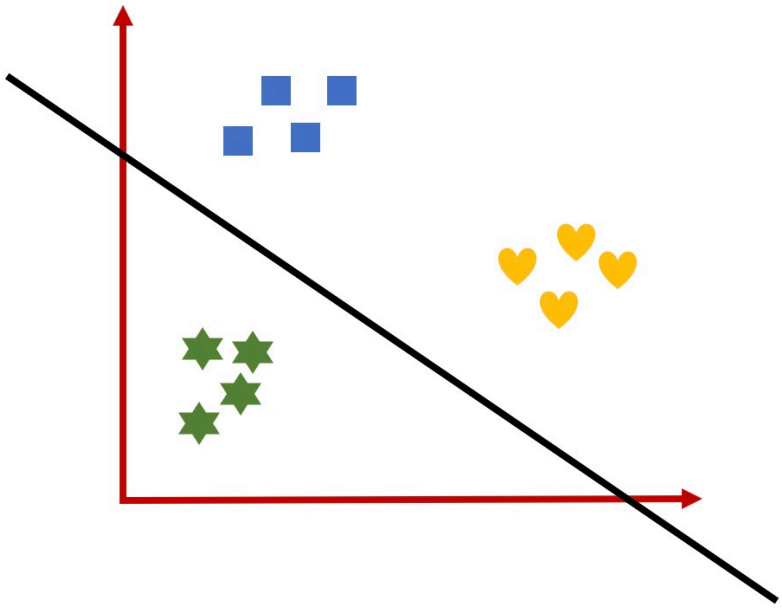


定义一个函数：

$$\hat{y}^{(1)} = P(y = 1|x; w)$$

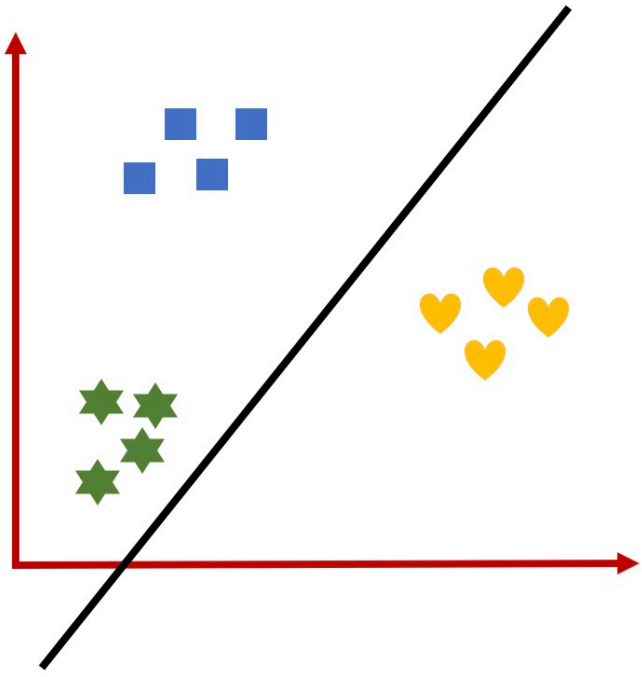
处理过的数据集就是二分类问题，通过逻辑回归可能得到黑线区分不同类别。

同理，



定义函数：

$$\hat{y}^{(2)} = P(y = 2|x; w)$$



定义函数：

$$\hat{y}^{(3)} = P(y = 3|x; w)$$

将公式总结在一起：

$$\hat{y}^{(i)} = P(y = i|x; w)$$

---

当需要预测新的数据的类别时，使用如下公式：

$$\max_i \hat{y}^{(i)}$$

使用不同的函数去预测输入$x$，分别计算不同$\hat{y}$的值，然后取其中的最大值。哪个类别 $i$ 对应的 $\hat{y}$ 越大，就认为属于哪个类。

而MvM则相对复杂，这里举MvM的特例one-vs-one(OvO)作讲解。如果模型有T类，我们每次在所有的T类样本里面选择两类样本出来，不妨记为T1类和T2类，把所有的输出为T1和T2的样本放在一起，把T1作为正例，T2作为负例，进行二元逻辑回归，得到模型参数。我们一共需要T(T- 1)/2次分类。

从上面的描述可以看出OvR相对简单，但分类效果相对略差(这里指大多数样本分布情况，某些样本分布下OvR可能更好)。而MvM分类相对精确，但是分类速度没有OvR快。

如果选择了ovr，则4种损失函数的优化方方法liblinear，newton-cg, lbfgs和sag都可以选择。但是如果选择了multinomial，则只能选择newton-cg, lbfgs和sag了。

这里使用鸢尾花多分类数据集进行演示：

```
from sklearn.linear_model import LogisticRegression as LR
from sklearn.datasets import load_iris
iris = load_iris()

for multi_class in ('multinomial', 'ovr'):
    clf = LR(solver='sag', max_iter=100, random_state=42,
                             multi_class=multi_class).fit(iris.data,
iris.target)

#打印两种multi_class模式下的训练分数
#%的用法，用%来代替打印的字符串中，想由变量替换的部分。%.3f表示，保留三位小数的浮点数。%s表示，字符串。
#字符串后的%后使用元祖来容纳变量，字符串中有几个%，元组中就需要有几个变量

    print("training score : %.3f (%s)" % (clf.score(iris.data, iris.target),
multi_class))
```