

菊安酱的机器学习第4期

菊安酱的直播间: <https://live.bilibili.com/14988341>

每周一晚8:00 菊安酱和你不见不散哦~(^o^)/~

更新日期: 2018-11-26

作者: 菊安酱

课件内容说明:

- 本文为作者原创内容, 转载请注明作者和出处
- 如果想获得此课件及录播视频, 可扫描左边二维码, 回复"k"进群
- 如果想获得2小时完整版视频, 可扫描右边二维码或点击如下链接
- 若有任何疑问, 请给作者留言。



交流群二维码



完整版视频及课件

直播视频及课件: <http://www.peixun.net/view/1278.html>

完整版视频及课件: <http://edu.cda.cn/course/966>

12期完整版课纲

直播时间: 每周一晚8:00

直播内容:

时间	期数	算法
2018/11/05	第1期	k-近邻算法
2018/11/12	第2期	决策树
2018/11/19	第3期	朴素贝叶斯
2018/11/26	第4期	Logistic回归
2018/12/03	第5期	支持向量机
2018/12/10	第6期	AdaBoost 算法
2018/12/17	第7期	线性回归
2018/12/24	第8期	树回归
2018/12/31	第9期	K-均值聚类算法
2019/01/07	第10期	Apriori 算法
2019/01/14	第11期	FP-growth 算法
2019/01/21	第12期	奇异值分解SVD

Logistic回归

菊安酱的机器学习第4期

12期完整版课纲

Logistic回归

一、概述

1. Logistic Regression
 - 1.1 线性回归
 - 1.2 Sigmoid函数
 - 1.3 逻辑回归
 - 1.4 LR 与线性回归的区别
2. LR的损失函数
3. LR 正则化
 - 3.1 L1 正则化
 - 3.2 L2 正则化
 - 3.3 L1正则化和L2正则化的区别
4. RL 损失函数求解
 - 4.1 基于对数似然损失函数
 - 4.2 基于极大似然估计

二、梯度下降法

1. 梯度
2. 梯度下降的直观解释
3. 梯度下降的详细算法
 - 3.1 梯度下降法的代数方式描述
 - 3.2 梯度下降法的矩阵方式描述
4. 梯度下降的种类
 - 4.1 批量梯度下降法BGD
 - 4.2 随机梯度下降法SGD
 - 4.3 小批量梯度下降法MBGD
5. 梯度下降的算法调优

三、使用梯度下降求解逻辑回归

1. 使用BGD求解逻辑回归
 - 1.1 导入数据集
 - 1.2 定义辅助函数
 - 1.3 BGD算法python实现
 - 1.4 准确率计算函数
2. 使用SGD求解逻辑回归
 - 2.1 SGD算法python实现

四、从疝气病症预测病马的死亡率

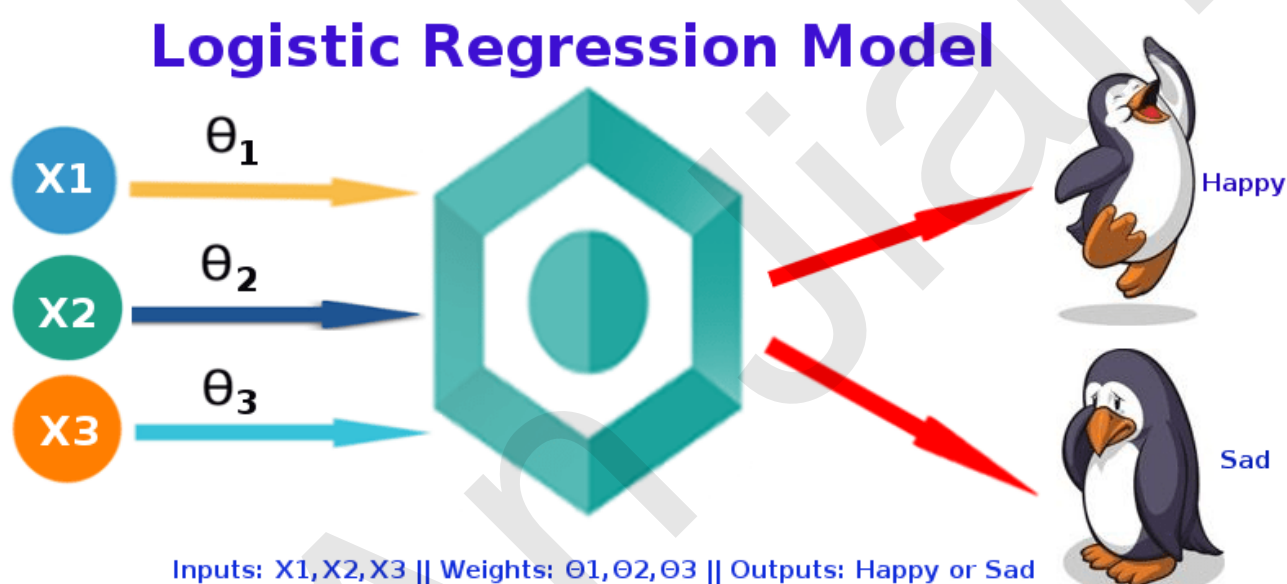
1. 准备数据
2. logistic回归分类函数

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。解决分类问题的算法也有很多种，比如：k-近邻算法，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；朴素贝叶斯，使用概率论构建分类器。这里我们要讲的是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。

1. Logistic Regression

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）、垃圾邮件检测（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。



1.1 线性回归

提到逻辑回归我们不得不提一下线性回归，逻辑回归和线性回归同属于广义线性模型，逻辑回归就是用线性回归模型的预测值去拟合真实标签的**对数几率**（一个事件的几率（odds）是指该事件发生的概率与不发生的概率之比，如果该事件发生的概率是 P ，那么该事件的几率是 $\frac{P}{1-P}$ ，对数几率就是 $\log \frac{P}{1-P}$ ）。

逻辑回归和线性回归本质上都是得到一条直线，不同的是，线性回归的直线是尽可能去拟合输入变量 X 的分布，使得训练集中所有样本点到直线的距离最短；而逻辑回归的直线是尽可能去拟合**决策边界**，使得训练集样本中的样本点尽可能分离开。因此，两者的目的是不同的。

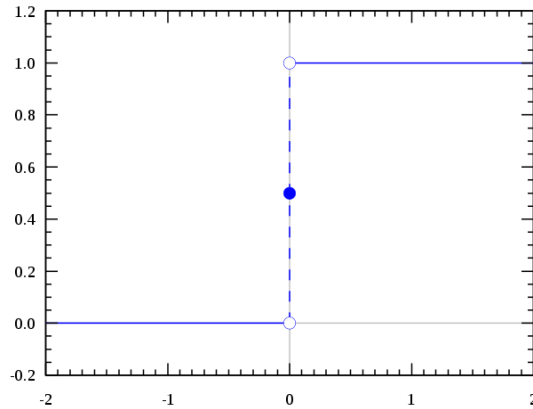
线性回归方程：

$$y = wx + b$$

此处， y 为因变量， x 为自变量。在机器学习中 y 是标签， x 是特征。

1.2 Sigmoid函数

我们想要的函数应该是，能接受所有的输入然后预测出类别。例如在二分类的情况下，函数能输出0或1。那拥有这类性质的函数称为海维赛德阶跃函数（Heaviside step function），又称之为单位阶跃函数（如下图所示）



单位阶跃函数的问题在于：在0点位置该函数从0瞬间跳跃到1，这个瞬间跳跃过程很难处理（不好求导）。幸运的是，Sigmoid函数也有类似的性质，且数学上更容易处理。

Sigmoid函数公式：

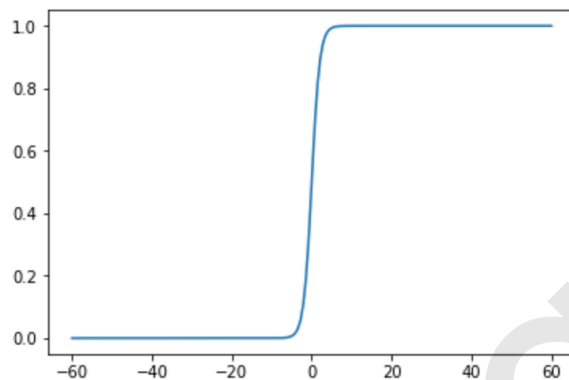
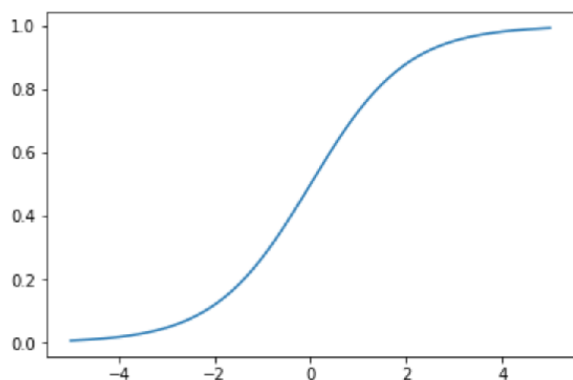
$$f(x) = \frac{1}{1 + e^{-(x)}}$$

```
import numpy as np
import math
import matplotlib.pyplot as plt
%matplotlib inline

x = np.linspace(-5,5,200)
y = [1/(1+math.e**(-x)) for x in x]
plt.plot(X,y)
plt.show()

x = np.linspace(-60,60,200)
y = [1/(1+math.e**(-x)) for x in x]
plt.plot(X,y)
plt.show()
```

下图给出了Sigmoid函数在不同坐标尺度下的两条曲线。当x为0时，Sigmoid函数值为0.5。随着x的增大，对应的函数值将逼近于1；而随着x的减小，函数值逼近于0。所以Sigmoid函数值域为（0,1），注意这是开区间，它仅无限接近0和1。如果横坐标刻度足够大，Sigmoid函数看起来就很像一个阶跃函数了。



1.3 逻辑回归

通过将线性模型和Sigmoid函数结合, 我们可以得到逻辑回归的公式:

$$y = \frac{1}{1 + e^{-(wx+b)}}$$

这样y就是 (0,1) 的取值。

对式子进行变换, 可得:

$$\log \frac{y}{1-y} = wx + b$$

这个其实就是一个对数几率公式

二项Logistic回归:

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

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

多项Logistic回归:

$$P(y=k|x) = \frac{e^{w_k \cdot x}}{1 + \sum_{k=1}^{K-1} e^{w_k \cdot x}}$$

$$P(y=K|x) = \frac{1}{1 + \sum_{k=1}^{K-1} e^{w_k \cdot x}}$$

1.4 LR 与线性回归的区别

逻辑回归和线性回归是两类模型, 逻辑回归是分类模型, 线性回归是回归模型

(看到这里的小伙伴, 请忽略视频讲解的内容~, 因为这两个模型是不同类的模型不可这样比较~~)

2. LR的损失函数

在机器学习算法中, 我们常常使用损失函数来衡量模型预测的好坏。损失函数, 通俗讲, 就是衡量**真实值和预测值之间差距**的函数。所以, 损失函数越小, 模型就越好。在这里, 最小损失是0。

LR损失函数为:

$$\begin{cases} -\log(x), y = 1 \\ -\log(1-x), y = 0 \end{cases}$$

看一下这个函数的图像:

```
x = np.linspace(0.0001,1,200)
y = [(-np.log(x)) for x in x]
plt.plot(X,y)
plt.show()

x = np.linspace(0,0.99999,200)
y = [(-np.log(1-x)) for x in x]
plt.plot(X,y)
plt.show()
```

我们把这两个损失函数综合起来:

$$-[y\log(x) + (1-y)\log(1-x)]$$

y就是标签, 分别取0, 1

对于m个样本, 总的损失函数为:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y_i \log(p(x_i)) + (1 - y_i) \log(1 - p(x_i))]$$

这个式子中, m是样本数, y是标签, 取值0或1, i表示第i个样本, p(x)表示预测的输出。

不过当损失过于小的时候, 也就是模型能够拟合绝大部分的数据, 这时候就容易出现过拟合。为了防止过拟合, 我们会引入正则化。

3. LR 正则化

3.1 L1 正则化

Lasso 回归, 相当于为模型添加了这样一个先验知识: w 服从零均值拉普拉斯分布。

拉普拉斯分布:

$$f(w|\mu, b) = \frac{1}{2b} \exp\left(-\frac{|w - \mu|}{b}\right)$$

其中 μ, b 为常数, 且 $\mu > 0$ 。

下面证明这一点, 由于引入了先验知识, 所以似然函数这样写:

$$L(w) = P(y|w, x)P(w) = \prod_{i=1}^N p(x_i)^{y_i} (1 - p(x_i))^{1-y_i} \prod_{j=1}^d \frac{1}{2b} \exp\left(-\frac{|w_j|}{b}\right)$$

取log再取负, 得到目标函数:

$$-\log L(w) = -\sum_i [y_i \log p(x_i) + (1 - y_i) \log(1 - p(x_i))] + \frac{1}{2b^2} \sum_j |w_j|$$

等价于原始的cross-entropy后面加上了L1正则，因此L1正则的本质其实是为模型增加了“模型参数服从零均值拉普拉斯分布”这一先验知识。

3.2 L2 正则化

Ridge 回归，相当于为模型添加了这样一个先验知识： w 服从零均值正态分布。

正态分布公式：

$$f(w|\mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(w - \mu)^2}{2\sigma^2}\right)$$

下面证明这一点，由于引入了先验知识，所以似然函数这样写：

$$\begin{aligned} L(w) &= P(y|w, x)P(w) = \prod_{i=1}^N p(x_i)^{y_i} (1 - p(x_i))^{1-y_i} \prod_{j=1}^d \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{w_j^2}{2\sigma^2}\right) \\ &= \prod_{i=1}^N p(x_i)^{y_i} (1 - p(x_i))^{1-y_i} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{w^T w}{2\sigma^2}\right) \end{aligned}$$

取log再取负，得到目标函数：

$$-\log L(w) = -\sum_i [y_i \log p(x_i) + (1 - y_i) \log(1 - p(x_i))] + \frac{w^T w}{2\sigma^2} + \text{const}$$

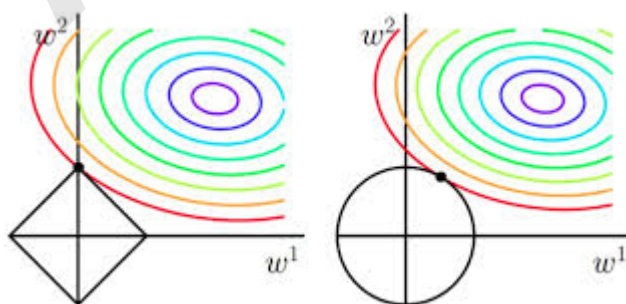
等价于原始的cross-entropy后面加上了L2正则，因此L2正则的本质其实是为模型增加了“模型参数服从零均值正态分布”这一先验知识。

3.3 L1正则化和L2正则化的区别

1. 两者引入的关于模型参数的先验知识不一样，L1是拉普拉斯分布，L2是正态分布
2. L1偏向于使模型参数变得稀疏(但实际上并不那么容易)，L2偏向于使模型每一个参数都很小，但是更加稠密，从而防止过拟合。

为什么L1偏向于稀疏，L2偏向于稠密呢？

看下面两张图，每一个圆表示loss的等高线，即在该圆上loss都是相同的，可以看到L1更容易在坐标轴上达到，而L2则容易在象限里达到。



4. RL 损失函数求解

4.1 基于对数似然损失函数

对数似然损失函数为:

$$L(Y, P(Y|X)) = -\log P(Y|X)$$

对于LR来说, 单个样本的对数似然损失函数可以写成如下形式:

$$L(y_i, p(y_i|x_i)) = \begin{cases} -\log \frac{\exp(w \cdot x)}{1 + \exp(w \cdot x)} = \log(1 + e^{-w \cdot x}) = \log(1 + e^{-\hat{y}_i}), y_i = 1 \\ -\log \frac{1}{1 + \exp(w \cdot x)} = \log(1 + e^{w \cdot x}) = \log(1 + e^{\hat{y}_i}), y_i = 0 \end{cases}$$

综合起来, 写成同一个式子:

$$L(y_i, p(y_i|x_i)) = y_i \log(1 + e^{-w \cdot x}) + (1 - y_i) \log(1 + e^{w \cdot x})$$

于是对整个训练样本集而言, 对数似然损失函数是:

$$J(w) = \frac{1}{N} \sum_{i=1}^N \{y_i \log(1 + e^{-w \cdot x}) + (1 - y_i) \log(1 + e^{w \cdot x})\}$$

4.2 基于极大似然估计

设 $p(y = 1|x) = p(x)$

$p(y = 0|x) = 1 - p(x)$,

假设样本是独立同分布生成的, 它们的似然函数就是各样本后验概率连乘:

$$L(w) = P(y|w, x) = \prod_{i=1}^N p(x_i)^{y_i} [1 - p(x_i)]^{1-y_i}$$

为了防止数据下溢, 写成对数似然函数形式:

$$\begin{aligned} \log L(w) &= \sum_{i=1}^N [y_i \log p(x_i) + (1 - y_i) \log(1 - p(x_i))] \\ -\log L(w) &= -\sum_{i=1}^N [y_i (w \cdot x_i) - \log(1 + e^{w \cdot x})] \end{aligned}$$

可以看出实际上 $J(w) = -\frac{1}{N} \log L(w)$, $J(w)$ 要最小化, 而 $\log L(w)$ 要最大化, 实际上是等价的。

讨论:

1. 损失函数为什么是log损失函数 (交叉熵) ,而不是MSE?

假设目标函数是MSE而不是交叉熵, 即:

$$\begin{aligned} L &= \frac{(y - \hat{y})^2}{2} \\ \frac{\partial L}{\partial w} &= (\hat{y} - y) \sigma'(w \cdot x) x \end{aligned}$$

这里sigmoid的导数项:

$$\sigma'(w \cdot x) = w \cdot x(1 - w \cdot x)$$

根据 w 的初始化, 导数值可能很小(想象一下sigmoid函数在输入较大时的梯度)而导致收敛变慢, 而训练途中也可能因为该值过小而提早终止训练。

另一方面, logloss的梯度如下, 当模型输出概率偏离于真实概率时, 梯度较大, 加快训练速度, 当拟合值接近于真实概率时训练速度变缓慢, 没有MSE的问题。

$$g = \sum_{i=1}^N x_i(y_i - p(x_i))$$

二、梯度下降法

由于极大似然函数无法直接求解, 所以在机器学习算法中, 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。

1. 梯度

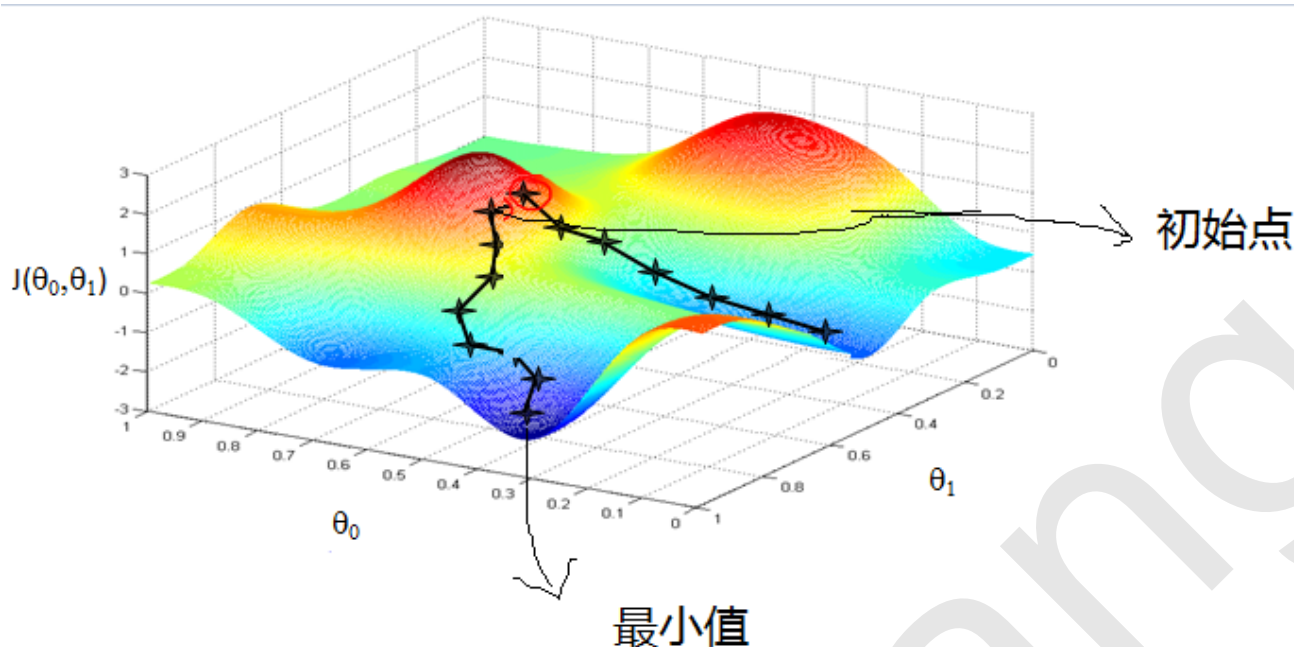
在微积分里面, 对多元函数的参数求 ∂ 偏导数, 把求得的各个参数的偏导数以向量的形式写出来, 就是梯度。比如函数 $f(x,y)$, 分别对 x,y 求偏导数, 求得的梯度向量就是 $(\partial f/\partial x, \partial f/\partial y)^T$, 简称 $\text{grad } f(x,y)$ 或者 $\nabla f(x,y)$ 。对于在点 (x_0, y_0) 的具体梯度向量就是 $(\partial f/\partial x_0, \partial f/\partial y_0)^T$ 或者 $\nabla f(x_0, y_0)$, 如果是3个参数的向量梯度, 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? 他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 $(\partial f/\partial x_0, \partial f/\partial y_0)^T$ 的方向是 $f(x,y)$ 增加最快的地方。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 $-(\partial f/\partial x_0, \partial f/\partial y_0)^T$ 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. 梯度下降的详细算法

梯度下降法的算法可以有代数法和矩阵法（也称向量法）两种表示，如果对矩阵分析不熟悉，则代数法更加容易理解。不过矩阵法更加的简洁，且由于使用了矩阵，实现逻辑更加的一目了然。这里先介绍代数法，后介绍矩阵法。

3.1 梯度下降法的代数方式描述

1. 先决条件：确认优化模型的假设函数和损失函数。

比如对于线性回归，假设函数表示为 $h_{\theta}(x_1, x_2, \dots, x_n) = \theta_0 + \theta_1 x_1 + \dots + \theta_n x_n$ ，其中 $\theta_i (i = 0, 1, 2, \dots, n)$ 为模型参数， $x_i (i = 0, 1, 2, \dots, n)$ 为每个样本的 n 个特征值。这个表示可以简化，我们增加一个特征 $x_0 = 1$ ，这样

$h_{\theta}(x_0, x_1, \dots, x_n) = \sum_{I=0}^n \theta_I x_I$ 。同样是线性回归，对应于上面的假设函数，损失函数为（此处损失函数之前加上 $\frac{1}{2m}$ ，主要是为了修正SSE让计算公式结果更加美观，实际上损失函数取MSE或SSE均可，二者对于一个给定样本而言只相差一个固定数值）：

$$J(\theta_0, \theta_1, \dots, \theta_n) = \frac{1}{2m} \sum_{j=0}^m (h_{\theta}(x_0^{(j)}, x_1^{(j)}, \dots, x_n^{(j)}) - y_j)^2$$

2. 算法相关参数初始化：主要是初始化 $\theta_0, \theta_1, \dots, \theta_n$ ，算法终止距离 ϵ 以及步长 α 。在没有任何先验知识的时候，我们比较倾向于将所有的 θ 初始化为0，将步长初始化为1。在调优的时候再进行优化。

3. 算法过程：

(1). 确定当前位置的损失函数的梯度，对于 θ_i ，其梯度表达式如下：

$$\frac{\partial}{\partial \theta_i} J(\theta_0, \theta_1, \dots, \theta_n)$$

(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 $\alpha \frac{\partial}{\partial \theta_i} J(\theta_0, \theta_1, \dots, \theta_n)$ ，对应于前面登山例子中的某一步。

(3). 确定是否所有的 θ_i ，梯度下降的距离都小于 ϵ ，如果小于 ϵ 则算法终止，当前所有的 $\theta_i (i = 0, 1, \dots, n)$ 即为最终结果。否则进入步骤4。

(4). 更新所有的 θ ，对于 θ_i ，其更新表达式如下。更新完毕后继续转入步骤1。

$$\theta_i = \theta_i - \alpha \frac{\partial}{\partial \theta_i} J(\theta_0, \theta_1, \dots, \theta_n)$$

下面用线性回归的例子来具体描述梯度下降。假设我们的样本是

$(x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)}, y_0), (x_1^{(1)}, x_2^{(1)}, \dots, x_n^{(1)}, y_1), \dots, (x_1^{(m)}, x_2^{(m)}, \dots, x_n^{(m)}, y_m)$, 损失函数如前面先决条件所述:

$$J(\theta_0, \theta_1, \dots, \theta_n) = \frac{1}{2m} \sum_{j=0}^m (h_{\theta}(x_0^{(j)}, x_1^{(j)}, \dots, x_n^{(j)}) - y_j)^2$$

则在算法过程步骤1中对于 θ_i 的偏导数计算如下:

$$\frac{\partial}{\partial \theta_i} J(\theta_0, \theta_1, \dots, \theta_n) = \frac{1}{m} \sum_{j=0}^m (h_{\theta}(x_0^{(j)}, x_1^{(j)}, \dots, x_n^{(j)}) - y_j) x_i^{(j)}$$

由于样本中没有 x_0 上式中令所有的 x_0^j 为1. 步骤4中 θ_i 的更新表达式如下:

$$\theta_i = \theta_i - \alpha \frac{1}{m} \sum_{j=0}^m (h_{\theta}(x_0^{(j)}, x_1^{(j)}, \dots, x_n^{(j)}) - y_j) x_i^{(j)}$$

从这个例子可以看出当前点的梯度方向是由所有的样本决定的, 加 $\frac{1}{m}$ 是为了好理解。由于步长也为常数, 他们的乘积也为常数, 所以这里 $\alpha \frac{1}{m}$ 可以用一个常数表示。在下面会详细讲到的梯度下降法的变种, 他们主要的区别就是对样本的采用方法不同。这里我们采用的是用所有样本。

3.2 梯度下降法的矩阵方式描述

这一部分主要讲解梯度下降法的矩阵方式表述, 相对于上面的代数法, 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

$h_{\theta}(x_1, x_2, \dots, x_n) = \theta_0 + \theta_1 x_1 + \dots + \theta_n x_n$ 的矩阵表达方式为:

$$h_{\theta}(\mathbf{x}) = \mathbf{X}\theta$$

其中, 假设函数 $h_{\theta}(\mathbf{X})$ 为 $m \times 1$ 的向量, θ 为 $(n+1) \times 1$ 的向量, 里面有 n 个代数法的模型参数。 \mathbf{X} 为 $m \times (n+1)$ 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: $J(\theta) = \frac{1}{2m} (\mathbf{X}\theta - \mathbf{Y})^T (\mathbf{X}\theta - \mathbf{Y})$, 其中 \mathbf{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: θ 向量可以初始化为默认值, 或者调优后的值。算法终止距离 ϵ , 步长 α 和3.1比没有变化。
3. 算法过程:

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

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

(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 $\alpha \frac{\partial}{\partial \theta} J(\theta)$ 对应于前面登山例子中的某一步。

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

(4). 更新 $\mathbf{\theta}$ 向量, 其更新表达式如下。更新完毕后继续转入步骤1。

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

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 θ 向量的偏导数计算如下:

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

步骤4中 θ 向量的更新表达式如下:

$$\theta = \theta - \alpha \mathbf{X}^T (\mathbf{X}\theta - \mathbf{Y}) / m$$

可以看到矩阵法要简洁很多。这里面用到了矩阵求导链式法则, 和两个矩阵求导的公式。

$$\text{公式1: } \frac{\partial}{\partial \mathbf{X}} (\mathbf{X}\mathbf{X}^T) = 2\mathbf{X}$$

$$\text{公式2: } \frac{\partial}{\partial \theta} (\mathbf{X}\theta) = \mathbf{X}^T$$

4. 梯度下降的种类

4.1 批量梯度下降法BGD

批量梯度下降法 (Batch Gradient Descent, BGD) 是梯度下降法最常用的形式, 具体做法也就是在更新参数时使用所有的样本来进行更新。

$$\theta_i = \theta_i - \alpha \sum_{j=1}^m (h_{\theta}(x_0^{(j)}, x_1^{(j)}, \dots, x_n^{(j)}) - y_j) x_i^{(j)}$$

由于我们有 m 个样本, 这里求梯度的时候就用了所有样本的梯度数据。

4.2 随机梯度下降法SGD

随机梯度下降法, 其实和批量梯度下降法原理类似, 区别在与求梯度时没有用所有的 m 个样本的数据, 而是仅仅选取一个样本 j 来求梯度。对应的更新公式是:

$$\theta_i = \theta_i - \alpha (h_{\theta}(x_0^{(j)}, x_1^{(j)}, \dots, x_n^{(j)}) - y_j) x_i^{(j)}$$

随机梯度下降法和批量梯度下降法是两个极端, 一个采用所有数据来梯度下降, 一个用一个样本来梯度下降。自然各自的优缺点都非常突出。对于训练速度来说, 随机梯度下降法由于每次仅仅采用一个样本来迭代, 训练速度很快, 而批量梯度下降法在样本量很大的时候, 训练速度不能让人满意。对于准确度来说, 随机梯度下降法用于仅仅用一个样本决定梯度方向, 导致解很有可能不是最优。对于收敛速度来说, 由于随机梯度下降法一次迭代一个样本, 导致迭代方向变化很大, 不能很快的收敛到局部最优解。但值得一提的是, 随机梯度下降法在处理非凸函数优化的过程当中有非常好的表现, 由于其下降方向具有一定随机性, 因此能很好的绕开局部最优解, 从而逼近全局最优解。

那么, 有没有一个中庸的办法能够结合两种方法的优点呢? 有! 这就是下面的小批量梯度下降法。

4.3 小批量梯度下降法MBGD

小批量梯度下降法是批量梯度下降法和随机梯度下降法的折衷, 也就是对于 m 个样本, 我们采用 x 个子样本来迭代, $1 < x < m$ 。一般可以取 $x=10$, 当然根据样本的数据, 可以调整这个 x 的值。对应的更新公式是:

$$\theta_i = \theta_i - \alpha \sum_{j=t}^{t+x-1} (h_{\theta}(x_0^{(j)}, x_1^{(j)}, \dots, x_n^{(j)}) - y_j) x_i^{(j)}$$

总结:

BGD会获得全局最优解, 缺点是在更新每个参数的时候需要遍历所有的数据, 计算量会很大, 并且会有很多的冗余计算, 导致的结果是当数据量大的时候, 每个参数的更新都会很慢。

SGD以高方差频繁更新, 优点是使得SGD会跳到新的和潜在更好的局部最优解, 缺点是使得收敛到局部最优解的过程更加的复杂。

MBGD降结合了BGD和SGD的优点, 每次更新的时候使用n个样本。减少了参数更新的次数, 可以达到更加稳定收敛结果, 一般在深度学习当中可以采用这种方法, 将数据一个batch一个batch的送进去训练。

不过在使用上述三种方法时有两个问题是不可避免的:

- 1、如何选择合适的学习率 (learning_rate)。自始至终保持同样的学习率显然是不太合适的, 开始学习参数的时候, 距离最优解比较远, 需要一个较大的学习率能够快速的逼近最优解。当参数接近最优解时, 继续保持最初的学习率, 容易越过最优点, 在最优点附近震荡。
- 2、如何对参数选择合适的学习率。对每个参数都保持的同样的学习率也是很很不合理的。有些参数更新频繁, 那么学习率可以适当小一点。有些参数更新缓慢, 那么学习率就应该大一点。

针对以上问题, 就提出了诸如Adam, 动量法等优化方法, 感兴趣的小伙伴可以自行研究。

5. 梯度下降的算法调优

1. 算法的**步长**选择。步长的选择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。前面说了。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。
2. 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, 需要多次用不同初始值运行算法, 关键损失函数的最小值, 选择损失函数最小化的初值。
3. **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, 收敛速度可以大大加快。

三、使用梯度下降求解逻辑回归

testSet数据集中一共有100个点, 每个点包含两个数值型特征: X1和X2。因此可以将数据在一个二维平面上展示出来。我们可以将第一列数据(X1)看作x轴上的值, 第二列数据(X2)看作y轴上的值。而最后一列数据即为分类标签。根据标签的不同, 对这些点进行分类。

在此数据集上, 我们将通过批量梯度下降法和随机梯度下降法找到最佳回归系数。

1. 使用BGD求解逻辑回归

批量梯度下降法的伪代码:

```
每个回归系数初始化为1
重复下面步骤直至收敛:
    计算整个数据集的梯度
    使用alpha*gradient更新回归系数的向量
返回回归系数
```

1.1 导入数据集

```
import pandas as pd
import numpy as np

dataSet = pd.read_table('testSet.txt', header = None)
dataSet.columns = ['x1', 'x2', 'labels']
dataSet
```

1.2 定义辅助函数

Sigmoid函数

```
"""
函数功能: 计算sigmoid函数值
参数说明:
    inX: 数值型数据
返回:
    s: 经过sigmoid函数计算后的函数值
"""
def sigmoid(inX):
    s = 1/(1+np.exp(-inX))
    return s
```

标准化函数

```

"""
函数功能: 标准化 (期望为0, 方差为1)
参数说明:
    xMat: 特征矩阵
返回:
    inMat: 标准化之后的特征矩阵
"""

```

```

def regularize(xMat):
    inMat = xMat.copy()
    inMeans = np.mean(inMat,axis = 0)
    inVar = np.std(inMat,axis = 0)
    inMat = (inMat - inMeans)/inVar
    return inMat

```

1.3 BGD算法python实现

```

"""
函数功能: 使用BGD求解逻辑回归
参数说明:
    dataSet: DF数据集
    alpha: 步长
    maxCycles: 最大迭代次数
返回:
    weights: 各特征权重值
"""

```

```

def BGD_LR(dataSet,alpha=0.001,maxCycles=500):
    xMat = np.mat(dataSet.iloc[:, :-1].values)
    yMat = np.mat(dataSet.iloc[:, -1].values).T
    xMat = regularize(xMat)
    m,n = xMat.shape
    weights = np.zeros((n,1))
    for i in range(maxCycles):
        grad = xMat.T*(xMat * weights-yMat)/m
        weights = weights -alpha*grad
    return weights

```

```

ws=BGD_LR(dataSet,alpha=0.01,maxCycles=500)
xMat = np.mat(dataSet.iloc[:, :-1].values)
yMat = np.mat(dataSet.iloc[:, -1].values).T
xMat = regularize(xMat)
(xMat * ws).A.flatten()

```

```

p = sigmoid(xMat * ws).A.flatten()
for i, j in enumerate(p):
    if j < 0.5:
        p[i] = 0
    else:
        p[i] = 1

```



```
train_error = (np.fabs(yMat.A.flatten() - p)).sum()
train_error_rate = train_error / yMat.shape[0]
train_error_rate
```

1.4 准确率计算函数

将上述过程封装为函数，方便后续调用。

```
"""
函数功能: 计算准确率
参数说明:
    dataSet: DF数据集
    method: 计算权重函数
    alpha: 步长
    maxCycles: 最大迭代次数
返回:
    trainAcc: 模型预测准确率
"""
def logisticAcc(dataSet, method, alpha=0.01, maxCycles=500):
    weights = method(dataSet, alpha=alpha, maxCycles=maxCycles)
    p = sigmoid(xMat * ws).A.flatten()
    for i, j in enumerate(p):
        if j < 0.5:
            p[i] = 0
        else:
            p[i] = 1
    train_error = (np.fabs(yMat.A.flatten() - p)).sum()
    trainAcc = 1 - train_error / yMat.shape[0]
    return trainAcc
```

测试函数运行效果

```
logisticAcc(dataSet, BGD_LR, alpha=0.01, maxCycles=500)
```

2. 使用SGD求解逻辑回归

随机梯度下降法的伪代码:

```
每个回归系数初始化为1
对数据集中每个样本:
    计算该样本的梯度
    使用alpha*gradient更新回归系数值
返回回归系数值
```

2.1 SGD算法python实现

```
"""
函数功能: 使用SGD求解逻辑回归
```

参数说明:

dataSet: DF数据集
alpha: 步长
maxCycles: 最大迭代次数

返回:

weights: 各特征权重值

"""

```
def SGD_LR(dataSet,alpha=0.001,maxCycles=500):
    dataSet = dataSet.sample(maxCycles, replace=True)
    dataSet.index = range(dataSet.shape[0])
    xMat = np.mat(dataSet.iloc[:, :-1].values)
    yMat = np.mat(dataSet.iloc[:, -1].values).T
    xMat = regularize(xMat)
    m, n = xMat.shape
    weights = np.zeros((n,1))
    for i in range(m):
        grad = xMat[i].T * (xMat[i] * weights - yMat[i])
        weights = weights - alpha * grad
    return weights
```

```
SGD_LR(dataSet,alpha=0.001,maxCycles=500)
```

计算准确率

```
logisticAcc(dataSet, SGD_LR, alpha=0.001, maxCycles=5000)
```

四、从疝气病症预测病马的死亡率

将使用Logistic回归来预测患疝气病的马的存活问题。原始数据集下载地址:

<http://archive.ics.uci.edu/ml/datasets/Horse+Colic>

这里的数据包含了368个样本和28个特征。这种病不一定源自马的肠胃问题, 其他问题也可能引发马疝病。该数据集中包含了医院检测马疝病的一些指标, 有的指标比较主观, 有的指标难以测量, 例如马的疼痛级别。另外需要说明的是, 除了部分指标主观和难以测量外, 该数据还存在一个问题, 数据集中有30%的值是缺失的。下面将首先介绍如何处理数据集中的数据缺失问题, 然后再利用Logistic回归和随机梯度上升算法来预测病马的生死。

1. 准备数据

数据中的缺失值是一个非常棘手的问题, 很多文献都致力于解决这个问题。那么, 数据缺失究竟带来了什么问题? 假设有100个样本和20个特征, 这些数据都是机器收集回来的。若机器上的某个传感器损坏导致一个特征无效时该怎么办? 它们是否还可用? 答案是肯定的。因为有时候数据相当昂贵, 扔掉和重新获取都是不可取的, 所以必须采用一些方法来解决这个问题。下面给出了一些可选的做法:

- 使用可用特征的均值来填补缺失值;
- 使用特殊值来填补缺失值, 如-1;
- 忽略有缺失值的样本;
- 使用相似样本的均值添补缺失值;
- 使用另外的机器学习算法预测缺失值。

预处理数据做两件事:

- 如果测试集中一条数据的特征值已经确实, 那么我们选择实数0来替换所有缺失值, 因为本文使用Logistic回归。因此这样做不会影响回归系数的值。sigmoid(0)=0.5, 即它对结果的预测不具有任何倾向性。
- 如果测试集中一条数据的类别标签已经缺失, 那么我们将该类别数据丢弃, 因为类别标签与特征不同, 很难确定采用某个合适的值来替换。

原始的数据集经过处理, 保存为两个文件: horseColicTest.txt和horseColicTraining.txt。

```
train = pd.read_table('horseColicTraining.txt', header=None)
train.head()
train.shape
train.info()

test = pd.read_table('horseColicTest.txt', header=None)
test.head()
test.shape
test.info()
```

2. logistic回归分类函数

得到训练集和测试集之后, 我们可以利用前面的BGD_LR或者SGD_LR得到训练集的weights。

这里需要定义一个分类函数, 根据sigmoid函数返回的值来确定y是0还是1

```
"""
函数功能: 给定测试数据和权重, 返回标签类别
参数说明:
    inx: 测试数据
    weights: 特征权重
"""
def classify(inx, weights):
    p = sigmoid(sum(inx * weights))
    if p < 0.5:
        return 0
    else:
        return 1
```

构建logistic模型:

```
"""
函数功能: logistic分类模型
参数说明:
    train: 测试集
    test: 训练集
    alpha: 步长
    maxCycles: 最大迭代次数
返回:
    retest: 预测好标签的测试集
"""
def get_acc(train, test, alpha=0.001, maxCycles=5000):
    weights = SGD_LR(train, alpha=alpha, maxCycles=maxCycles)
```

```
xMat = np.mat(test.iloc[:, :-1].values)
xMat = regularize(xMat)
result = []
for inX in xMat:
    label = classify(inX, weights)
    result.append(label)
retest=test.copy()
retest['predict']=result
acc = (retest.iloc[:, -1]==retest.iloc[:, -2]).mean()
print(f'模型准确率为: {acc}')
return retest
```

```
get_acc(train, test, alpha=0.001, maxCycles=5000)
```

其他

- 菊安酱的直播间: <https://live.bilibili.com/14988341>
- 下周一 (2018/12/3) 将讲解**支持向量机**, 欢迎各位进入菊安酱的直播间观看直播
- 如有问题, 可以给我留言哦~