2.3 对数几率回归

名字是回归，实际是分类

要讲解对数几率回归模型，得从线性模型讲起。线性模型是最简单、最基本的模型之一，是构成一大类复杂模型的基础，从本节开始一直到课程结束，读者会经常看到他的身影。

2.3.1 线性分类模型

简单来讲，线性模型是指通过样本特征的线性组合来进行预测的模型。举个例子，直线方程$y = kx + b$，就是一个线性模型。为什么称为线性模型呢？因为这个模型描述了一条 2D（x和y）空间里的直线，直线当然是线性的。类似的，可以推广到 3D 及更高维空间，这时候$\boldsymbol{k}$和$\boldsymbol{x}$就是向量了，而$y = \boldsymbol{k}^T\boldsymbol{x} + b$就表示 3D 空间里的平面或更高维空间里的“超平面”。关于“维数”和“降维”，后面第 4 章还会更深入讨论。

正式地，定义一个d维的特征向量$\boldsymbol{x} = (x_1, \ldots, x_d)$，一个$d$维的权重向量$\boldsymbol{w} = (w_1, \ldots, w_d)$，一个偏置$b$（即$\boldsymbol{x}$为$\boldsymbol{0}$时的取值），称$f(\boldsymbol{x}; \boldsymbol{w})$为一个线性模型，如式（2.3.1）所示：

$$f(\boldsymbol{x}; \boldsymbol{w}) = \boldsymbol{w}^T\boldsymbol{x} + b \tag{2.3.1}$$

也可以用图 2.3.1 来表示式（2.3.1）所描述的线性模型。注意，图 2.3.1 中，偏置b对应的是固定的输入分量“1”！有时候，也会把这个分量“1”也放到$\boldsymbol{x}$里，从而得到$d + 1$维的“增扩”特征向量，相应的把b放到$\boldsymbol{w}$里，从而形成$d + 1$维的“增扩”权重向量。另外，按照 1.2.2 节关于含参模型和非参模型的定义，很显然，线性模型属于含参模型。而前面已经介绍的 k-近邻和决策树则都属于非参模型。

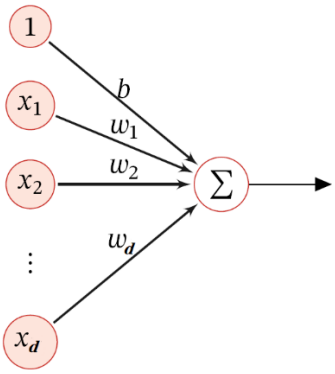


图 2.3.1 线性模型

式（2.3.1）所示的线性模型可以方便地应用到回归问题上，也就是直接用$\hat{y} = f(\boldsymbol{x}; \boldsymbol{w})$来预测连续目标变量$y$的值。这是后面第 3 章的主题。那么，读者可以思考下，对于本节的分类问题怎么办呢？实际上，在分类问题中，由于目标变量y是一些离散的标签，而$f(\boldsymbol{x}; \boldsymbol{w})$的值域为实数，因此无法直接用$f(\boldsymbol{x}; \boldsymbol{w})$来进行预测，需要引入一个非线性的决策函数$g(\cdot)$来预测目标变量$y$的值：

$$y = g(f(\boldsymbol{x}; \boldsymbol{w})) \tag{2.3.2}$$

式（2.3.2）就是需要的线性分类模型，可以用图 2.3.2 来表示。图 2.3.2 中的上图，表示增加的决策函数$g(\cdot)$可以将$f(\boldsymbol{x}; \boldsymbol{w})$按照阈值划分为正类（+）和负类（-）。图 2.3.2 中的下图，则表示以$\boldsymbol{w}$和b为参数的平面$\boldsymbol{w}^T\boldsymbol{x} + b = 0$把两类数据（实心圆点和空心圆点）成功的分开。

注意，图 2.3.2 下图中，特征向量$\boldsymbol{x} = (x_1, x_2)$。

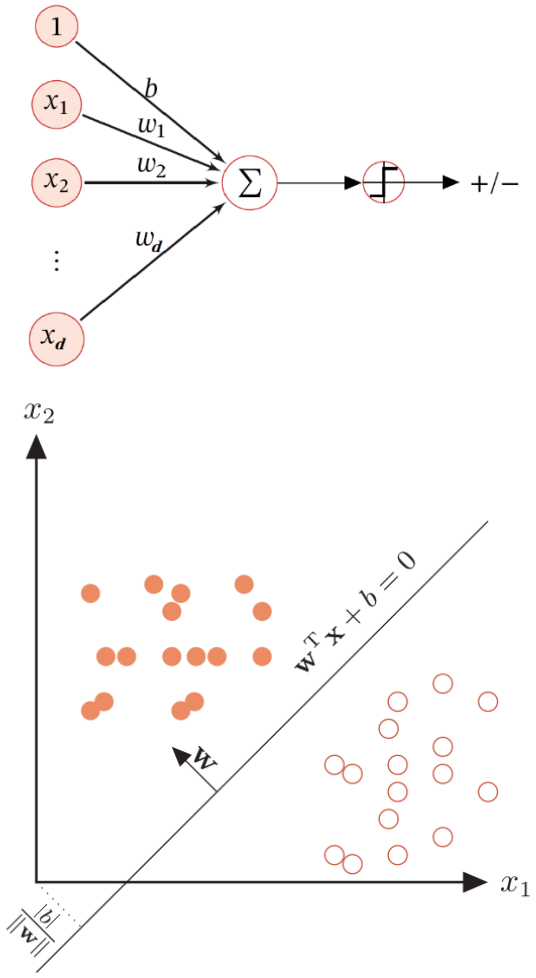


图 2.3.2 线性分类模型

对于二分类问题，$g(\cdot)$可以是符号函数（Sign Function），定义为：

$$\begin{aligned} g\big(f(\boldsymbol{x};\boldsymbol{w})\big) &= sgn\big(f(\boldsymbol{x};\boldsymbol{w})\big) \\ &= \begin{cases} +1, & if\ f(\boldsymbol{x};\boldsymbol{w}) \geq 0 \\ -1, & if\ f(\boldsymbol{x};\boldsymbol{w}) < 0 \end{cases} \end{aligned} \tag{2.3.3}$$

2.3.2 对数几率函数

对于二分类问题，$g(\cdot)$也可以是对数几率函数，如式（2.3.4）和图 2.3.3 所示：

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.3.4}$$

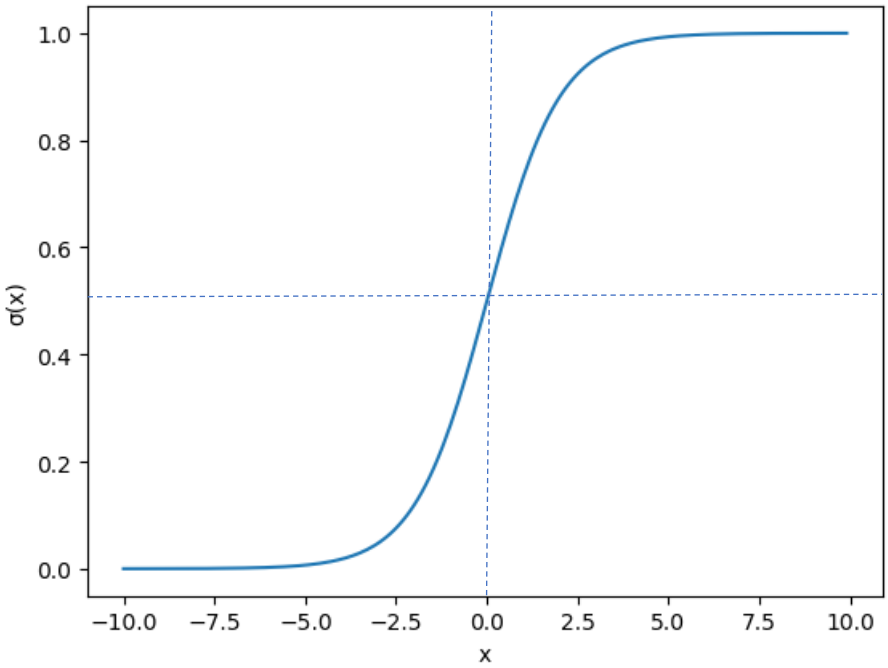


图 2.3.3 对数几率函数

图 2.3.4 把符号函数（也称为单位阶跃函数）和对数几率函数画在了一张图里。由图 2.3.4 可见，符号函数并不连续（$z=0$为不连续点），而对数几率函数则既连续又可微。一般来讲，要求决策函数$g(\cdot)$单调可微，这样便于利用数值优化的方法（比如马上就要用到的“梯度下降”优化算法）来学习模型的参数（称为可微分学习）。对数几率函数能够在一定程度上近似符号函数，而且单调可微，所以常用他来替代符号函数！

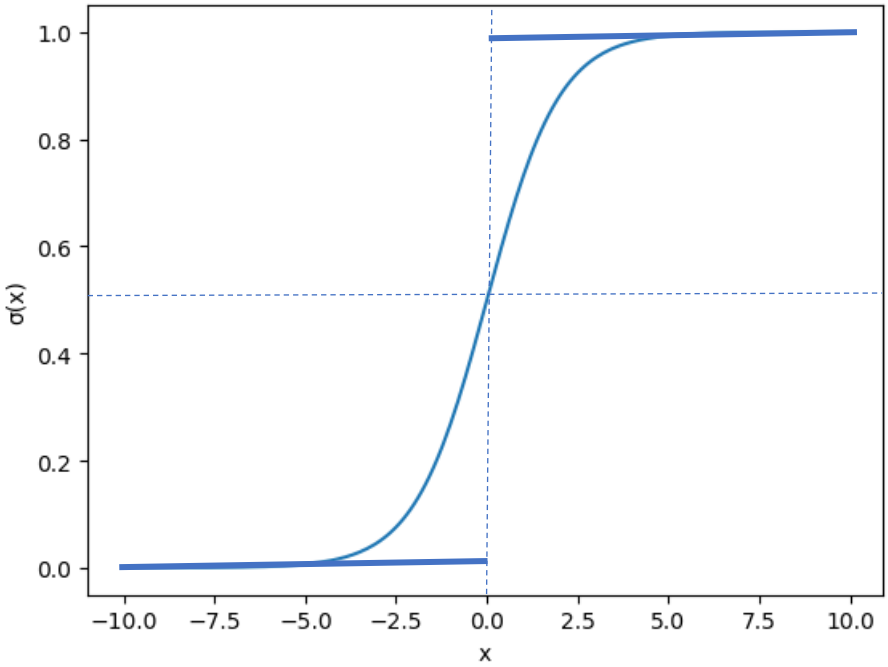


图 2.3.4 符号函数和对数几率函数

实际上，对数几率函数是“S型曲线函数”的一种，他把实数域的输入“挤压”到(0,1)

的输出范围内。当输入值在 0 附近时，该函数近似为线性函数；而当输入值靠近两端时，则对输入变化不敏感（输入越小，越接近于 0；输入越大，越接近于 1）。重要的是，这样的特点也和生物神经元类似，对一些输入会产生兴奋（输出为 1），对另一些输入则产生抑制（输出为 0）。所以在神经网络中（参见 2.5 节），也习惯将其称为“激活函数”。

类似对数几率函数这一类的S型曲线函数还有很多，比如式（2.3.5）和图 2.3.5 给出的$tanh$函数（双曲正切）。可见，$tanh$函数实际上就是在水平方向压缩、在垂直方向拉伸并平移的对数几率函数。

$$f(x) = tanh(x) = 2\sigma(2x) - 1 \tag{2.3.5}$$

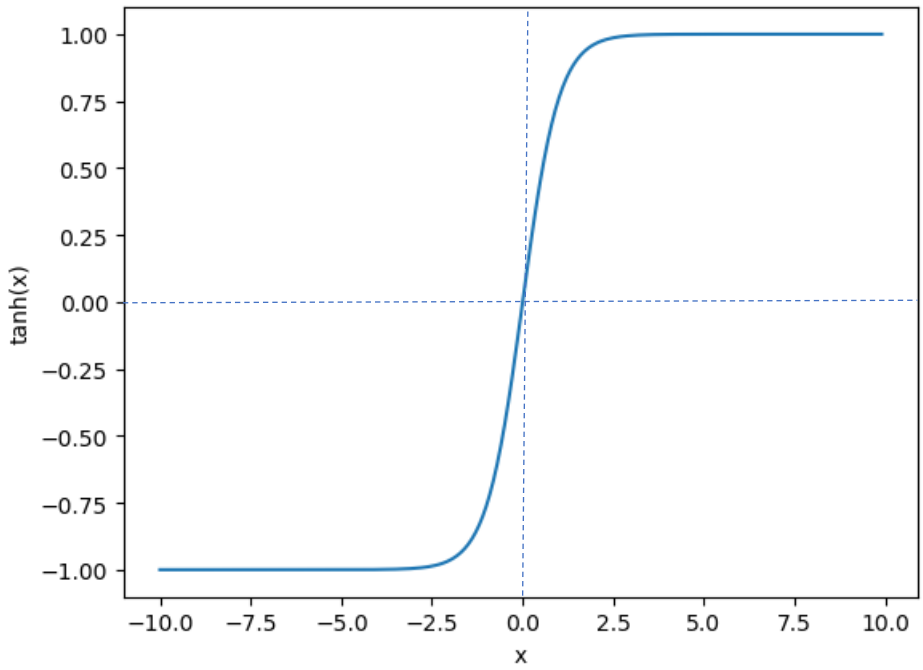


图 2.3.5 $tanh$函数

对比图 2.3.3 所示的对数几率函数，可以看到两个函数主要的不同之处：值域不同；关于原点的对称性不同。$tanh$函数是关于原点中心对称的，称之为 0-中心化；对数几率函数则是非 0-中心化。这个性质对梯度下降算法的收敛速度是有影响的，需要引起注意。

2.3.3 对数几率回归

对于二分类问题，如果引入对数几率函数作为决策函数，就得到对数几率回归模型。如式（2.3.6）和（2.3.7）所示：

$$\begin{aligned} p(y=1|\boldsymbol{x}) &= \sigma(\boldsymbol{w}^T\boldsymbol{x}) \\ &= \frac{1}{1+exp(-\boldsymbol{w}^T\boldsymbol{x})} \end{aligned} \tag{2.3.6}$$

$$\begin{aligned} p(y=0|\boldsymbol{x}) &= 1 - p(y=1|\boldsymbol{x}) \\ &= \frac{exp(-\boldsymbol{w}^T\boldsymbol{x})}{1+exp(-\boldsymbol{w}^T\boldsymbol{x})} \end{aligned} \tag{2.3.7}$$

其中，$\boldsymbol{w}^T\boldsymbol{x}$中的$\boldsymbol{x}$是增扩特征向量，而$\boldsymbol{w}$是对应的增扩权重向量；而且为了书写的方便，用$exp(x)$表示指数函数$e^x$。

对数几率函数的值域为$(0,1)$，自然的对应概率。这样，对于二分类问题，对数几率回归

模型实际上就是在拟合二值的伯努利概率分布。有趣的是，将式（2.3.6）和（2.3.7）联立起来可以推出（习题 2.3.3）：

$$\boldsymbol{w}^T\boldsymbol{x} = \ln\frac{p(y=1|\boldsymbol{x})}{1-p(y=1|\boldsymbol{x})} = \ln\frac{p(y=1|\boldsymbol{x})}{p(y=0|\boldsymbol{x})} \tag{2.3.8}$$

其中，$\frac{p(y=1|\boldsymbol{x})}{p(y=0|\boldsymbol{x})}$表示样本$\boldsymbol{x}$为正负类后验概率的比值，称为几率。几率的对数$\ln\frac{p(y=1|\boldsymbol{x})}{p(y=0|\boldsymbol{x})}$称为对数几率。式（2.3.8）说明，对数几率回归可以看作是预测值为“标签的对数几率”的线性回归模型。这大概就是“对数几率回归”名字的来由——关于“对数几率”的线性回归模型（式（2.3.1））！回归的是“对数几率”！但最终解决的是式（2.3.6）和（2.3.7）定义的二分类问题。所以，对数几率回归，虽然名字叫回归，实际却是分类。

那么，如何基于训练数据$\boldsymbol{x}$学习参数$\boldsymbol{w}$呢？这个就涉及“极大似然估计”和“梯度下降”这两个知识点，接下来依次作个简要的介绍。

极大似然估计（Maximum Likelihood Estimate，MLE）的思想其实很简单，就是假定观察到的数据分布$p(\boldsymbol{x})$（称为似然）即为真实的数据分布$P(\boldsymbol{x})$（称为总体分布），并据此拟合出参数$\boldsymbol{w}$，从而得到所需的类别概率分布$p(y|\boldsymbol{x};\boldsymbol{w})$。以抛硬币为例，根据实验结果可以直接得到为正面或反面的概率——正面或反面次数除以总的次数。当然，这里有个关键问题，如果抛的次数太少，就会因为一些偶然因素而导致得到的正反面概率值不符合真实的情况（比如对于一枚均匀的硬币，正反面应该各占 50%），出现过拟合的现象。要克服这个问题，从理论层面，概率与统计理论有一个基本假设——样本数趋于无穷（大数定律、中心极限定理）；从实际操作层面，希望样本数越多越好。比如，对于分类问题，可以要求每一类的样本数不少于 1000。

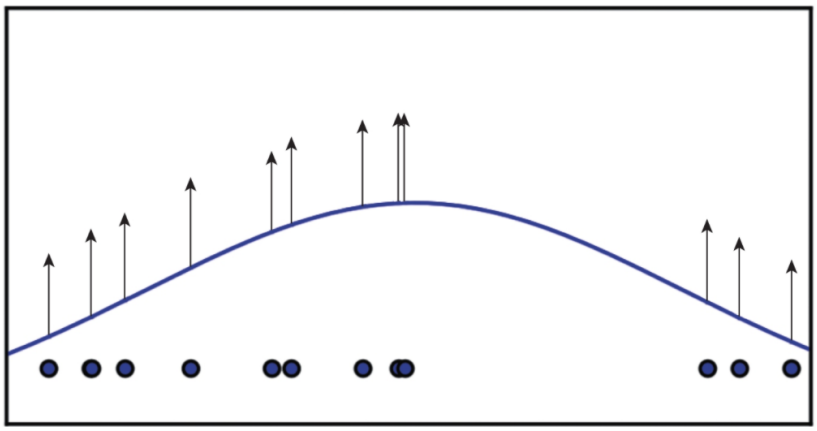


图 2.3.6 用极大似然估计法拟合一维高斯分布

再举个例子，如图 2.3.6 所示，采样到一些样本点（图中圆点所示），基于这些样本点用极大似然估计法拟合出一个一维高斯分布（图中曲线所示）。显然，极大似然估计法拟合出的一维高斯分布能够反映出样本的出现概率，并且能够满足尽可能多的训练样本点（包括最右边的 3 个“离群点”），这正体现了极大似然估计法的本质——观察到的就是真实的。

接着介绍梯度下降（Gradient Descent，GD）。如图 2.3.7 所示，用一个简单二次函数$f(x) = \frac{1}{2}x^2$来说明。图 2.3.7 中的点划线是$f(x)$的图像，其导函数$f'(x) = x$对应图中的实线。很显然，$f(x)$具有全局最小值点$x = 0$，该点的导数也为 0。那么，如果从任意x值出发，如何找

到这个全局最小值点$x=0$呢？假如从$x>0$的点出发，此时由于导数值大于 0（表示$f(x)$的值随着x的增加而增加），因此必须向着x减小的方向移动（即向左移动）才可能到达$f(x)$值最小的点。如果从$x<0$的点出发，情况则刚好相反。此时由于导数值小于 0（表示$f(x)$的值随着x的增加而减小），因此必须向着x增加的方向移动（即向右移动）才可能到达$f(x)$值最小的点。不管哪种情况，目的都是“下山”，到达“山脚”，即$f(x)$的最小值点$x=0$。由于“导数”在最优化术语中一般被称为“梯度”（对于多元函数是一个向量），所以这种最优化方法一般被称为“梯度下降”。也称为“最陡下降”（习题 2.3.4）。如果将$f(x)$取反，得到$f(x)=-\frac{1}{2}x^2$，则问题就反过来，需要找$f(x)$的最大值，此时则需要“上山”，到达“山顶”，所以相应的称为“梯度上升”或“最陡上升”。为了方便，后面统一用“梯度下降”这个提法。

还有个问题，不管是“下山”还是“上山”，都有个步子迈多大的问题。比如刚开始步子可以迈大点，当接近“山脚”或“山顶”时，步子就要迈小些，要不然就会错过“山脚”或“山顶”。实际上，可以用一个超参数$\varepsilon>0$来表示步长，梯度下降就可以表示为$f(x-\varepsilon*sign(f'(x)))$：如果$f'(x)<0$，就通过$x+\varepsilon$向右迈一步；否则，就通过$x-\varepsilon$向左迈一步。

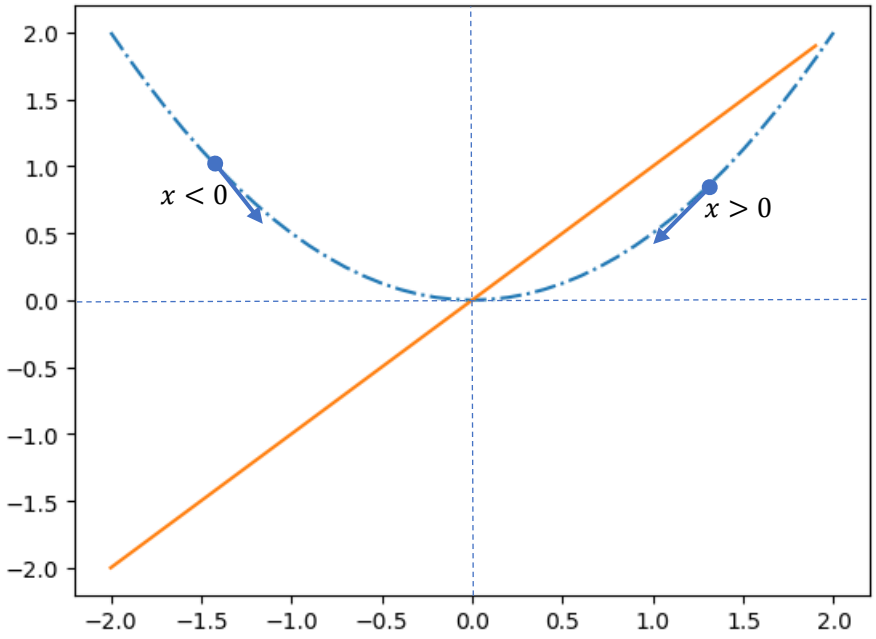


图 2.3.7 梯度下降最优化方法

理解了极大似然估计和梯度下降这两个知识点，就可以开始讨论对数几率回归的学习过程了。

首先，需要定义参数$\boldsymbol{w}$的对数似然函数$L(\boldsymbol{w})$。既然叫似然函数，那一定是在训练集D上（观察到的数据，称为似然）定义的：

$$L(\boldsymbol{w})=\sum_{i=1}^{N}\ln\left[p(y_i|\boldsymbol{x_i};\boldsymbol{w})\right] \tag{2.3.9}$$

其中，$\boldsymbol{x_i}$表示每一个训练样本，总共有N个训练样本，构成训练集D。$p(y_i|\boldsymbol{x_i};\boldsymbol{w})$称为似然项，表示给定训练样本$\boldsymbol{x_i}$和参数$\boldsymbol{w}$时，预测的类别值$y_i$的概率。显然，根据概率的积规则，这些似然项需要相乘。实际中，由于概率值为(0,1)，相乘容易导致下溢，所以，为了数值计算的稳定性，对其取对数（从而把乘积化为求和），就得到了对数似然项$\ln\left[p(y_i|\boldsymbol{x_i};\boldsymbol{w})\right]$。对于二分类问题，$y_i$的取值要么为 0、要么为 1，所以进一步得到：

$$L(\boldsymbol{w})=\sum_{i=1}^{N}ln\left[y_i p(y_i=1|\boldsymbol{x_i};\boldsymbol{w})+(1-y_i)\,p(y_i=0|\boldsymbol{x_i};\boldsymbol{w})\right] \tag{2.3.10}$$

式（2.3.10）的ln里有两项，第一项表示类别 1 的预测概率值$p(y_i=1|\boldsymbol{x_i};\boldsymbol{w})$与类别真值

y_i是否符合，如果相符（预测类别 1，真值也为类别 1），则对应的预测概率值$p(y_i = 1|\boldsymbol{x_i};\boldsymbol{w})$会被累加起来；反过来，如果不相符（预测类别 1，真值为类别 0），则乘积为 0。类似的，第二项表示类别 0 的预测概率值$p(y_i = 0|\boldsymbol{x_i};\boldsymbol{w})$与类别真值$y_i$是否符合，如果相符（预测类别 0，真值也为类别 0），则对应的预测概率值$p(y_i = 0|\boldsymbol{x_i};\boldsymbol{w})$会被累加起来；反过来，如果不相符（预测类别 0，真值为类别 1），则乘积为 0。所以，式（2.3.10）的意义就很清楚了：仅累加分类正确样本的概率值。显然，这个累加值$L(\boldsymbol{w})$越大越好（不仅要预测正确，而且要求预测正确的概率越高越好），优化的目标就是要找到合适的$\boldsymbol{w}$（记为$\boldsymbol{w}^*$），使得在训练集D上，$L(\boldsymbol{w}^*)$取得最大值：

$$\boldsymbol{w}^* = argmax_{\boldsymbol{w}}\boldsymbol{L}(\boldsymbol{w}) \tag{2.3.11}$$

以上过程就是用极大似然估计法估计参数$\boldsymbol{w}$，即找到使似然函数$L(\boldsymbol{w})$取得最大值的$\boldsymbol{w}^*$。

把式（2.3.6）和（2.3.7）带入到式（2.3.10）中，得到：

$$\begin{aligned} L(\boldsymbol{w}) &= \sum_{i=1}^{N} \ln\left[y_i \frac{1}{1+\exp(-\boldsymbol{w^T x_i})} + (1-y_i)\frac{\exp(-\boldsymbol{w^T x_i})}{1+\exp(-\boldsymbol{w^T x_i})}\right] \\ &= \sum_{i=1}^{N} \ln\left[\frac{y_i + \exp(-\boldsymbol{w^T x_i}) - y_i\exp(-\boldsymbol{w^T x_i})}{1+\exp(-\boldsymbol{w^T x_i})}\right] \\ &= \sum_{i=1}^{N}[(y_i-1)\boldsymbol{w^T x_i} - \ln(1+\exp(-\boldsymbol{w^T x_i}))] \\ &= \textstyle\sum_{i=1}^{N}[y_i\boldsymbol{w^T x_i} - \ln(1+\exp(\boldsymbol{w^T x_i}))] \end{aligned} \tag{2.3.12}$$

上面从第二行到第三行的推导用到了y_i为 0 或 1。以上过程大家一定要自己去推一遍，加深理解：光看跟自己动手去做有根本性差异!!（习题 2.3.5）

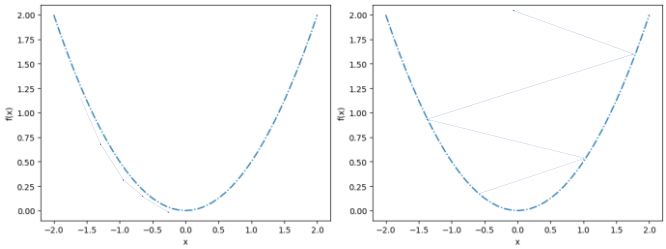
既然要找到$L(\boldsymbol{w})$的最大值，那么可以用前面讲到的梯度上升方法来求解。习惯上，可以等价的求$-L(\boldsymbol{w})$的最小值（式（2.3.13）），相应的就用梯度下降方法（式（2.3.14））：

$$\boldsymbol{w}^* = argmin_{\boldsymbol{w}}[-L(\boldsymbol{w})] \tag{2.3.13}$$

$$\boldsymbol{w} = \boldsymbol{w} - \alpha[-\nabla_{\boldsymbol{w}}L(\boldsymbol{w})] \tag{2.3.14}$$

式（2.3.14）中，$\nabla_{\boldsymbol{w}}L(\boldsymbol{w})$表示似然函数$L(\boldsymbol{w})$关于参数向量$\boldsymbol{w}$的梯度向量（符号∇表示微分算子，读作“纳布拉”）。α>0 称为学习率。式（2.3.14）与前面讲的梯度下降公式$x - \varepsilon * sign(f'(x))$稍有不同，还把梯度的幅度也考虑进来，将其与$\alpha$相乘，这样每一步的步长就为$\alpha|\nabla_{\boldsymbol{w}}L(\boldsymbol{w})|$。

关于学习率这个超参数，还要多说几句。还是以简单二次函数$f(x) = \frac{1}{2}x^2$为例来说明，如图 2.3.8 所示，该函数只有一个局部最小值，该值同时也是全局最小值。像这样的函数（任何局部最小都是全局最小）称为“凸函数”，对应的优化问题称为“凸优化”。凸函数的二阶导数大于 0，表示正曲率。对于凸函数，只要如图 2.3.8 左图所示，学习率α设得合适，则不管自变量的初值为多少，都能够通过梯度下降逐渐到达最小值点。但如果α设得过大，则会出现图 2.3.8（b）所示的发散情况：本来想“下山”，结果成了“上山”！一般来讲，学习率α是一个小的正数，在梯度下降过程中应该越来越小，尤其是在接近最小值的时候，这个过程如图 2.3.8（a）所示。



（a） （b）

图 2.3.8 凸优化问题

凸函数是比较理想的情况（本节的对数几率回归就属于这种情况）。一般的非凸情况如图 2.3.9 所示，其中，x_a为全局最小值点，也是期望到达的最小值点，但是通过梯度下降能否到达这个点，还取决于初值和学习率等诸多因素；x_b为局部最小值点，由于其值接近x_a的值，能够到达这个点也是可以接受的；x_c是另一个局部最小值点，由于其值远大于x_a的值，是一个不太好的局部最优解，应该尽量避免。实际中，由于$\boldsymbol{x}$一般具有很高的维数，不可能如图 2.3.9 一样把目标函数$f(\boldsymbol{x})$画出来，一目了然地看到最小值的具体情况，因此基于梯度下降的最优化算法就如同“瞎子摸象”或者“不识庐山真面目，只缘身在此山中”。

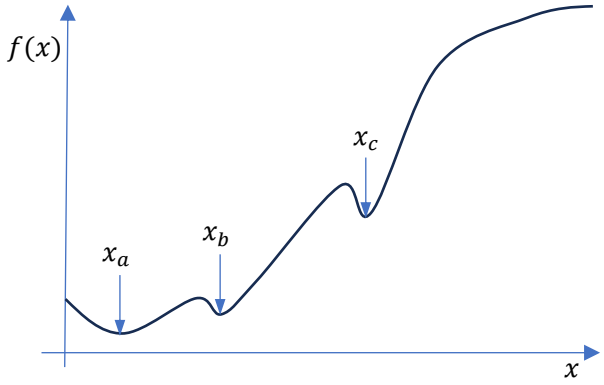


图 2.3.9 非凸优化问题

回到我们的问题上来，比较幸运，$-L(\boldsymbol{w})$是凸函数，因此式（2.3.12）的优化问题就是一个凸优化问题，用梯度下降（见式（2.3.14））就能求解。实际上，由$L(\boldsymbol{w}) = \sum_{i=1}^{N}[y_i\boldsymbol{w}^T\boldsymbol{x}_i - \ln(1+\exp(\boldsymbol{w}^T\boldsymbol{x}_i))]$，立即可以推出：

$$\nabla_{\boldsymbol{w}}L(\boldsymbol{w})=\sum_{i=1}^{N}\boldsymbol{x}_i[y_i-\hat{y}_i] \tag{2.3.15}$$

这个推导过程作为习题留给大家。

拓展一下，由$\nabla_{\boldsymbol{w}}L(\boldsymbol{w})=\sum_{i=1}^{N}\boldsymbol{x}_i[y_i-\hat{y}_i]=\boldsymbol{0}$，也可以直接推导出 w 的解析解，感兴趣的同学可以进一步尝试一下！

至此，得到了$\boldsymbol{w}^*$，所以最终学习到的对数几率回归模型就是：

$$p(y_i=1|\boldsymbol{x}_i;\boldsymbol{w}^*), p(y_i=0|\boldsymbol{x}_i;\boldsymbol{w}^*) \tag{2.3.16}$$

由此，给定一个测试样本$\boldsymbol{x}_i$，就可以直接得到其为类别 1 或 0 的概率，而概率高的类别就是最终的预测结果。

2.3.4 随机梯度下降

前面介绍了梯度下降的基本算法思想，并针对对数几率回归模型推导了其梯度向量计算

公式。由式（2.3.15）可见，需要把训练集D的所有N个样本（称为“batch”，意思是“一批”）都用来计算梯度向量，这种做法称之为“基本梯度下降 GD”。其优点是计算出的梯度向量更准确（因为每次计算都会把训练集的所有N个样本都用上）；其缺点也是显然的，如果N很大（比如上万），计算量将是非常大的！

针对 GD 计算量大的问题，“随机梯度下降 SGD”被提了出来。其基本思想是：既然梯度向量是一个训练集上的期望值，那么可以通过随机采样的一个小的训练集（称为“小 batch”）来逼近。也就是说，每次从具有N个样本的训练集D中随机抽取m个（$1 \le m \le N$）样本来计算梯度。只要重复的次数足够多（大数定律），小训练集就可以“逼近”完整训练集。用公式可以表示为：

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = \sum_{i=1}^{m} \boldsymbol{x}_i [y_i - \hat{y}_i] \tag{2.3.17}$$

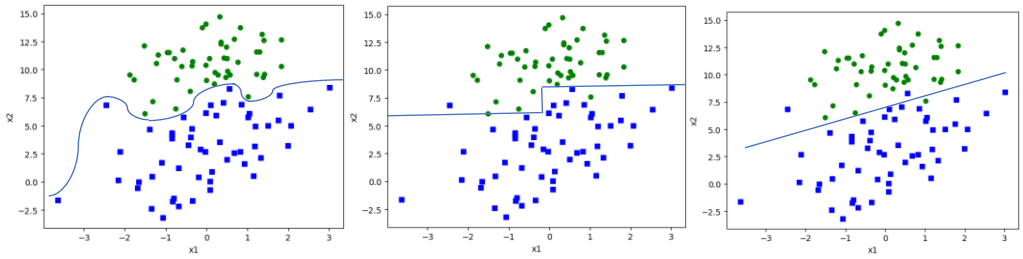
由于一般有$m << N$，所以当N很大时，SGD 的计算复杂度近似$O(1)$，非常高效。不仅如此，SGD 还能带来一些额外的好处：比如，“随机性”可能有助于非凸优化问题得到一个更好的局部最优解。再比如，可以实现“在线学习”——到达一个训练样本就学习一次。

关于“随机性”可能有助于非凸优化问题得到一个更好的局部最优解，读者可以结合图 2.3.8 的右图和图 2.3.9 思考一下为什么（习题 2.3.10）。

2.3.5 与 k-近邻和决策树的比较

2.1 节介绍了 k-近邻模型，这个模型有一个重要特点：未对数据自身作任何假设，具有最好的“弹性”，决策面几乎可以任意复杂（超参数$k = 1$时）。相比而言，对数几率回归模型是关于对数几率的线性模型，决策面就是一个超平面，这恰是最“刚性”的决策面。

可见，1-近邻和线性模型代表了模型的两个极端，这对分析各种模型的特点非常有指导意义，就像光谱一样，模型也有一个“模型谱”！比如，2.2 节介绍的决策树模型在这个“模型谱”上就处于 1-近邻和线性模型之间——不那么有弹性也不那么刚性。图 2.3.10 给出了一个可视化的比较：（a）是 1-近邻的决策边界，最为复杂，具有最好的“弹性”；（c）是对率回归的线性决策面，最为简单，也最为“刚性”；（b）是介于两者之间的决策树的决策面，是由一些平行坐标轴的线段连接而成的折线。



（a） （b） （c）

图 2.3.10 决策面可视化对比

换一个角度，对数几率回归模型可以用梯度下降求解，所以属于“可微分学习模型”，简称可微模型；而 k-近邻（基于距离计算）和决策树模型（基于信息熵计算）则属于“不可微分学习模型”，简称不可微模型。

2.3.6 对率回归的核心代码实现

本节将详细介绍 GD 算法和 SGD 算法的代码实现，并给出代码的简单运行实例。相信读者在此基础上能够在实验课中进一步完善和改进代码，完成更具挑战性和实用性的应用任务。

1．GD 算法的 Python 实现

首先来看数据装入函数 loadDataSet：

```
def loadDataSet(filename='dataSet.txt'):
    dataList = []; labelList = []
    fr = open(fileName)
    for line in fr.readlines():
        lineList = line.strip().split()
        dataList.append([1.0, float(lineList[0]), float(lineList[1])]) #1.0 是参数 b 对应的属性值
        labelList.append(int(lineList[2]))
    return dataList, labelList
```

该函数中，首先打开数据文件 fileName（是函数的唯一输入参数，默认值为'dataSet.txt'），然后用 for 循环以迭代方式读入文件的每一行。针对每一行，首先用字符串函数 strip 和 split 分别去掉尾部的换行符、以空格分割各个记录项，得到一个列表 lineList。然后，lineList 的前两个元素是两个特征，所以将其添加到数据列表 dataList 中。为什么在 lineList[0]和 lineList[1])的前面要添加一个固定的 1.0 呢？这个其实就是 2.3.1 说到的对应偏置b的固定输入“1”。类似地，lineList[2]是类别标签，相应的添加到标签列表 labelList 中。注意代码中的类型转换：由于 lineList 的元素都是字符串类型，所以分别要转换成对应的数值类型，dataList 要求浮点数，labelList 要求整型。函数最后返回 dataList 和 labelList。

显然，dataList 最终是一个嵌套列表，每一个列表元素仍然是一个列表，表示数据文件的一行（对应一个样本）。注意，代码中假定了一个样本是两个特征，这个留给读者在实验中来改进，以适应一般情况。类似地，labelList 最终是一个列表，每一个列表元素是对应样本的类别标签。

接下来定义对数几率函数 sigmoid（式（2.3.4））：

```
def sigmoid(fX):
    return 1.0/(1+np.exp(-fX))
```

该函数的参数只有一个，特征向量 fX。注意，这里使用的是 Numpy 的指数函数 np.exp。

然后就是 GD 算法的实现——函数 gradDescent（式（2.3.13）~（2.3.15））：

```
def gradDescent(dataList, labelList):
    dataMat = np.mat(dataList)                  #将列表转换为 Numpy 矩阵
    labelMat = np.mat(labelList).transpose()
    m = np.shape(dataMat)[1]                    #m 是特征的个数
    alpha = 0.001                               #学习率
    maxCycles = 500                             #训练轮数
    weights = np.ones((m, 1))                   #m*1 权重参数数组
    for k in range(maxCycles):                  #大量矩阵运算
```

```
            h = sigmoid(dataMat * weights)        #“*”是矩阵乘法
            error = (labelMat - h)                #“-”是向量减
            weights = weights + alpha * dataMat.transpose() * error
        return weights
```

该函数首先将 dataList 和 labelList 转换为 Numpy 矩阵，以便于下面利用 Numpy 的高效矩阵运算。注意，labelMat 是作了转置操作的，即由之前的 1 列 1 个样本转置为 1 行 1 个样本。接下来，用 np.shape 得到 dataMat 的列数 m（对应 m 个特征）。

学习率 alpha 定义为一个小的整数 0.001（式（2.3.14）中的α），迭代的轮数（一轮称为一个“epoch”）定义为 500。这两个是 GD 算法的超参，需要根据经验进行选择，或者采用交叉验证方式来选择。

接下来，定义一个 m 行 1 列的权重数组 weights。然后开始进入 GD 的迭代过程：for 循环里，第一行是计算$\sigma(\boldsymbol{w}^T\boldsymbol{x})$，得到当前的模型预测值 h；第二行是将类别标签 labelMat 与 h 相减，得到 error；第三行里，“dataMat.transpose() * error”对应式（2.3.15）——计算梯度，然后梯度乘以学习率 alpha，得到更新的步长，进而完成更新（式（2.3.14））。

迭代完成后，函数最后返回权重数组 weights。

下面以一个简单数据集 dataSet.txt（1 行 1 个样本，共 100 行；每行有 3 列，前 2 列对应 2 个特征（浮点类型），第 3 列对应类别标签（取值为 0 或 1））为例来验证一下代码的正确性：

```
>>> dataList, labelList = loadDataSet()
>>> gradDescent(dataList, labelList)
matrix([[ 4.12414349],
        [ 0.48007329],
        [-0.6168482 ]])
```

代码运行正常，也能得到正确的结果。为了更直观的查看 GD 算法的优化结果，可以用 Matplotlib 库将其可视化出来，代码如下：

```
def plotBestFit(dataList, labelList, weights):
    import matplotlib.pyplot as plt
    dataArr = np.array(dataList)
    n = np.shape(dataArr)[0] #得到样本数 n
    xcord1 = []; ycord1 = [] #类别 1 的坐标
    xcord2 = []; ycord2 = [] #类别 0 的坐标
    for i in range(n):
        if labelList[i] == 1:
            xcord1.append(dataArr[i,1]); ycord1.append(dataArr[i,2])
        else:
            xcord2.append(dataArr[i,1]); ycord2.append(dataArr[i,2])
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.scatter(xcord1, ycord1, s=30, c='blue', marker='s')
    ax.scatter(xcord2, ycord2, s=30, c='green')
    x1 = np.arange(-3.0, 3.0, 0.1)
    x2 = (-weights[0, 0]-weights[1, 0]*x1)/weights[2, 0] #0 = w0x0 + w1x1 + w2x2, x0=1
    ax.plot(x1, x2)
    plt.xlabel('x1'); plt.ylabel('x2')
```

plt.show()

简单说一下这个可视化函数 plotBestFit。其三个参数的前两个就是函数 loadDataSet 的返回结果，第三个参数 weights 则来自于函数 gradDescent 的返回结果。plotBestFit 函数首先将绘图库 matplotlib.pyplot 导入为 plt，然后由 np.shape 得到样本数 n。接下来，根据样本点的类别，将其分别存放到两组坐标列表中（xcord1, ycord1）与（xcord2, ycord2）。然后，用函数 ax.scatter 将其以散点图方式在画布上画出。剩下的就是画线性决策面了，用函数 ax.plot 即可完成。关键在于：根据 weights 由 x1 算出 x2。这可以通过决策面的表达式$0 = w_0x_0 + w_1x_1 + w_2x_2$来计算：因为$w_0$对应$b$，故$x_0 = 1$，则有$x_2 = (-w_0 - w_1x_1)/w_2$。

至此，在命令行执行如下语句即可将样本点和线性决策面画出（如图 2.3.11 所示）。

```
>>> dataList,labelList = loadDataSet()
>>> weights = gradDescent(dataList, labelList)
>>> plotBestFit(dataList, labelList, weights)
```

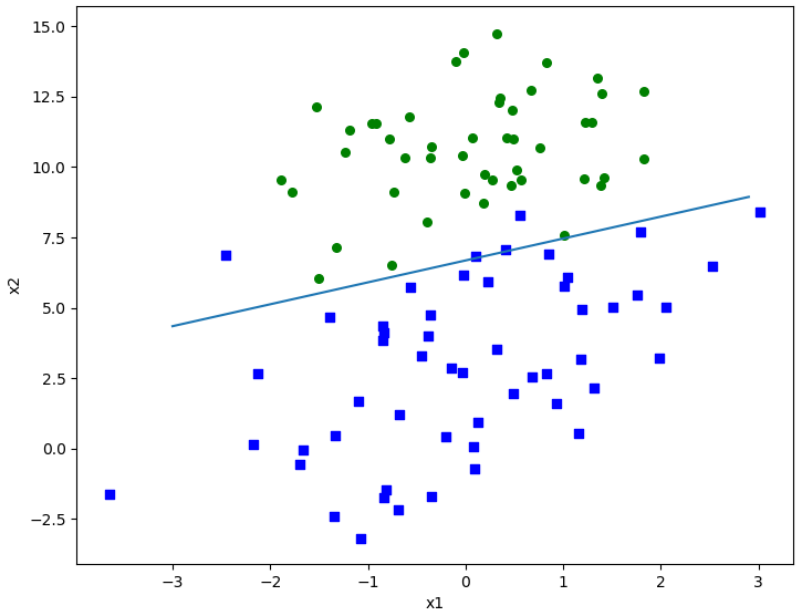


图 2.3.11 简单数据集的样本点和线性决策面

由图 2.3.11 可见，对于训练集中的 100 个样本点，GD 算法找到的线性决策面能够较好的将两类样本点分开：类别 1（图中的正方形点）有 4 个样本点被分错了，而类别 0（图中的圆形点）则没有样本点被分错。表 2.3.1 给出了训练集上的混淆矩阵。由该表可见，类 1 的样本总数为 53，有 4 个样本被分错；类 0 的样本总数为 47，有 0 个样本被分错。两个类的样本总数基本是平衡的。进一步，可以计算出P、R、$F1$，并画出 P-R 曲线，这些留给读者来完成。

表 2.3.1 训练集上的混淆矩阵

真实值/预测值	类 1	类 0
类 1	**49**	4
类 0	0	**47**

2．SGD 算法的 Python 实现

下面实现了一个“小 batch”为 1 的 SGD 算法，代码如下：

```
def stocGradDescent(dataList, labelList, numIter=500):
```

```
    dataArr = np.array(dataList)
    n, m = np.shape(dataArr)
    weights = np.ones(m) #1D 数组，全部初始化为 1
    for j in range(numIter):
        for i in range(n):
            alpha = 4/(1.0 + j + i) + 0.0001 #apha 随着迭代次数的增加而逐渐减小，但是由于加了一个小常量 0.0001 不会为 0
            randIndex = int(np.random.uniform(0, n))
            print("iter %d sample %d randIndex %d" % (j, i, randIndex))
            h = sigmoid(sum(dataArr[randIndex]*weights)) #“*”是向量乘
            error = labelList[randIndex] - h #scalar sub
            weights = weights + alpha * error * dataArr[randIndex] #“+”是向量加
    return weights
```

重点比较下与 gradDescent 的不同之处。

（1）为了方便，stocGradDescent 增加了一个指定迭代次数的默认参数 numIter。这个参数就是前面谈到的迭代轮数（一轮称为一个“epoch”）。

（2）weights 是一个 m 个元素的 1D 数组，而 gradDescent 里他是一个 m 行 1 列的 2D 数组。为何有此不同？这是因为，在 gradDescent 里，用了矩阵运算来进行计算的加速，特别是“h = sigmoid(dataMat * weights)”这一句代码，dataMat 是 m 列的矩阵，每一行就是一个样本，假设有 n 行；而 weights 则是 m 行 1 列的数组，两者作矩阵运算就得到 n 行 1 列的矩阵，该矩阵每一行就对应一个样本的预测值（即式（2.3.15）中的$\hat{y}_i$）。而 stocGradDescent 里，对应的代码修改为“h = sigmoid(sum(dataArr[randIndex]*weights))”。dataArr[randIndex] 是 m 个元素的向量，与 m 个元素的 weights 向量作相乘运算（对应元素直接相乘），得到的结果还是 m 个元素的向量，要得到预测值$\hat{y}_i$，还需进行求和运算。

（3）由于“小 batch”为 1，因此内层循环“for i in range(n):”里，每次只需“随机选择”一个样本 randIndex（因此总共选择 n 次，从而完成一轮训练），然后根据式（2.3.17）对其进行相应的向量和标量运算即可。

（4）gradDescent 里，采用了固定学习率“alpha = 0.001”，而在 stocGradDescent 中，采用了变化的学习率“alpha = 4/(1.0 + j + i) + 0.0001”，这个学习率总体上会随着迭代次数的增加而逐渐减小，而且不会小于 0.0001。正如前面介绍的，随着迭代的持续进行，会越来越接近目标函数的最小值（“山脚”），为了避免步子迈得过大而错过这个最小值，降低学习率从而减小步长是明智的做法。

需要注意的是，由于引入了随机性，stocGradDescent 每次执行的结果会有所不同。

习题 2.3

2.3.1 试解释 2.3.1 中的公式$y = \boldsymbol{k}^T\boldsymbol{x} + b$：$\boldsymbol{k}^T\boldsymbol{x}$做的什么运算，请写出其分量形式；$y$和$b$分别代表什么。

2.3.1’ 推导图 2.3.2 下图中原点到直线的距离$|b|/\|\boldsymbol{w}\|$（$\|\boldsymbol{w}\|$表示权重向量$\boldsymbol{w}$的二范数，即各分量的平方和再开平方根）。

2.3.2 比较公式（2.3.1）和公式（2.3.6）中的特征向量$\boldsymbol{x}$和权重向量$\boldsymbol{w}$的区别。

2.3.3 试推导公式（2.3.8）。

2.3.4 请解释为什么“梯度下降”也被称为“最陡下降”。

2.3.5 推导公式（2.3.12）。

2.3.6 根据你对相关概念的理解，详细解读图 2.3.11。

2.3.7 推导公式（2.3.15）。

2.3.7’ 在 2.3.7 的基础上，进一步推导$-L(\boldsymbol{w})$的二阶导数，并据此分析$-L(\boldsymbol{w})$的凸性。

2.3.8 推导对数几率函数$\sigma(x)$（公式（2.3.4））的导函数$\sigma'(x)$，并将其用$\sigma(x)$表示出来。

2.3.9 如果用$p(y_i = 1|\boldsymbol{x_i}; \boldsymbol{w})^{y_i} p(y_i = 0|\boldsymbol{x_i}; \boldsymbol{w})^{(1-y_i)}$替换公式(2.3.10)里$ln$函数的式子，你能够推导出公式(2.3.12)吗？如果能，你认为哪个更简单呢？为什么？

2.3.10 请用学习率解释图 2.3.8 的右图，是什么原因导致无法找到最小值？

2.3.11 请结合图 2.3.8 的右图和图 2.3.9 分析：为什么具有“随机性”的 SGD 可能有助于非凸优化问题得到一个更好的局部最优解。

2.3.12 在 2.3.3 节中，以抛硬币为例介绍了极大似然估计，如果抛的次数太少，就会因为一些偶然因素而导致得到的正反面概率值不符合真实的情况（比如对于一枚均匀的硬币，正反面应该各占 50%），出现过拟合的现象。你如何理解此处谈到的过拟合？存在欠拟合的情况吗？

2.3.13 设伯努利分布的参数$\mu = p(x_n = 1)$，其中$x_n \in \{0,1\}$。请给出μ的极大似然估计，并对得到的结果进行分析和解释。