第 3 章 线性模型

在本章中，我们将学习线性模型中的线性回归模型和 logistic 模型。通常，线性回归模型用于处理回归问题，logistic 模型用于处理分类问题。我们将学习线性模型的数学表示和模型解释并使用梯度下降法求解模型参数。线性模型本身具有广泛用途，更是深度学习模型的基石，相信通过学习理解相对简单的线性模型有助于学习复杂的深度学习模型。如果你对线性模型不是很熟悉，可以认真学习本章，打好学习深度学习的基础；如果你熟悉线性模型，也请浏览本章内容，因为本章将介绍深度学习中也会用到的术语、符号和优化算法（梯度下降法）。

3.1 线性回归模型

3.1.1 线性回归模型简介

在线性回归模型中，我们希望可以在模型中输入一个已知且相对容易测量的数据向量 $\boldsymbol{x} = (x_1, x_2, \cdots, x_p)$，并得到一个希望预测的数据类型是实数的因变量$y$。线性回归模型可以写成如下形式，

$$y = b + w_1 x_1 + w_2 x_2 + \cdots + w_p x_p + \epsilon$$

其中，b是一个常数（称为截距项或者偏差），$w_1, \cdots, w_p$分别是对应自变量的权重（$w_1, \cdots, w_p$也称为自变量的系数），ϵ是误差项。b和$w_1, \cdots, w_p$为未知常数，都是模型参数。误差项ϵ包含了没有体现在自变量$\boldsymbol{x}$，但是又对因变量y有影响的信息。

通常，我们把数据记为$(\boldsymbol{x}_1, y_1), \cdots, (\boldsymbol{x}_n, y_n)$。每个自变量向量和因变量的组合$(\boldsymbol{x}_i, y_i)$，$i = 1, \cdots, n$称为一个观测点，其中，$\boldsymbol{x}_i = (x_{i1}\ x_{i2}\ \cdots\ x_{ip})$为一个行向量，$y_i$为一个实数。数据的观测点数量称为数据的样本量，记为n；数据的自变量个数称为数据的维度，记为p。记列向量$\boldsymbol{w} = (w_1\ \cdots\ w_p)^{\mathsf{T}}$为权重向量。线性回归模型也可以写成如下形式，

$$y_i = b + \sum_{j=1}^{p} x_{ij} w_j + \epsilon_i = b + \boldsymbol{x}_i \boldsymbol{w} + \epsilon_i, \quad i = 1,2, \cdots, n$$

进一步用$\boldsymbol{X}$、$\boldsymbol{y}$、$\boldsymbol{w}$、$\boldsymbol{\epsilon}$表示自变量矩阵、因变量向量、权重向量、误差项向量，

$$\boldsymbol{X}=\begin{pmatrix} x_{11} & \cdots & x_{1p} \\ x_{21} & \cdots & x_{2p} \\ \vdots & & \vdots \\ x_{n1} & \cdots & x_{np} \end{pmatrix},\quad \boldsymbol{y}=\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix},\quad \boldsymbol{w}=\begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_p \end{pmatrix},\quad \boldsymbol{\epsilon}=\begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{pmatrix}$$

线性回归模型可以写成更加简洁的矩阵形式，

$$\boldsymbol{y}=b+\boldsymbol{X}\boldsymbol{w}+\boldsymbol{\epsilon}$$

在本节中，我们将使用一个简单数据来帮助介绍线性回归模型。该数据包含了某个商品在 200 个市场的广告投入和销量。广告投入分为 3 种不同的形式，分别是 TV、radio 和 newspaper，广告投入的单位是千美元（thousands of dollars）。例如，在第一个市场中，广告在 TV 的投入为 230 100 美元。在这个例子中，我们关心各种广告投入对商品销量（sales）的影响。商品销量的单位是 thousands of units。例如，第一个市场的销量为 22 100 个商品。表 3-1 列出了数据的前 5 行。

```
"""
载入一些本章需要用到的包: pandas, matplotlib, numpy
"""
# 该设置可以使得图片分辨率更高
%config InlineBackend.figure_format = 'retina'

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

"""
读入保存在 data 文件夹中的数据"Advertising.csv"， 并显示前 5 行

同学们运行下面代码时，需要注意数据保存路径，
1. 如果数据保存在当前工作路径中，
   那么代码 pd.read_csv("Advertising.csv")即可读入数据
2. 如果数据没有保存在当前工作路径中，
   需要在函数 pd.read_csv()中写上数据的完整路径
   例如，pd.read_csv("C:/Users/Documents/Advertising.csv")

在我们的电脑中，数据"Advertising.csv"保存在当前工作路径的文件夹 data 中。
因此，完整路径可以写成'./data/Advertising.csv'
"""
advertising = pd.read_csv('./data/Advertising.csv')
advertising.head()
```

表 3-1 广告数据的前 5 行

	TV	radio	newspaper	sales
0	230.1	37.8	69.2	22.1
1	44.5	39.3	45.1	10.4
2	17.2	45.9	69.3	9.3
3	151.5	41.3	58.5	18.5

4	180.8	10.8	58.4	12.9

我们分别以 TV、radio、newspaper 为横轴，以 sales 为纵轴画散点图，如图 3-1 所示。圆圈表示 TV 与 sales 的散点图；星号表示 radio 与 sales 的散点图；正方形表示 newspaper 与 sales 的散点图。从图 3-1 可以看到，在这三种广告方式中，TV 和 radio 对 sales 的影响较大，newspaper 对 sales 影响较小。

```
"""
分别以 TV, radio, newspaper 为横轴，画散点图
"""
plt.scatter("TV", "sales", data=advertising, label="TV")
plt.scatter("radio", "sales", data=advertising, marker="*",\
            label="radio")
plt.scatter("newspaper", "sales", data=advertising,    \
            marker="s", label="newspaper")
# 为图片增加图例，参数 loc 可以设置图例的位置。
plt.legend(loc="lower right")
plt.xlabel("Advertising Budgets", fontsize=16)     # 添加 x 轴名称
plt.ylabel("Sales", fontsize=16)                   # 添加 y 轴名称
plt.show()
```

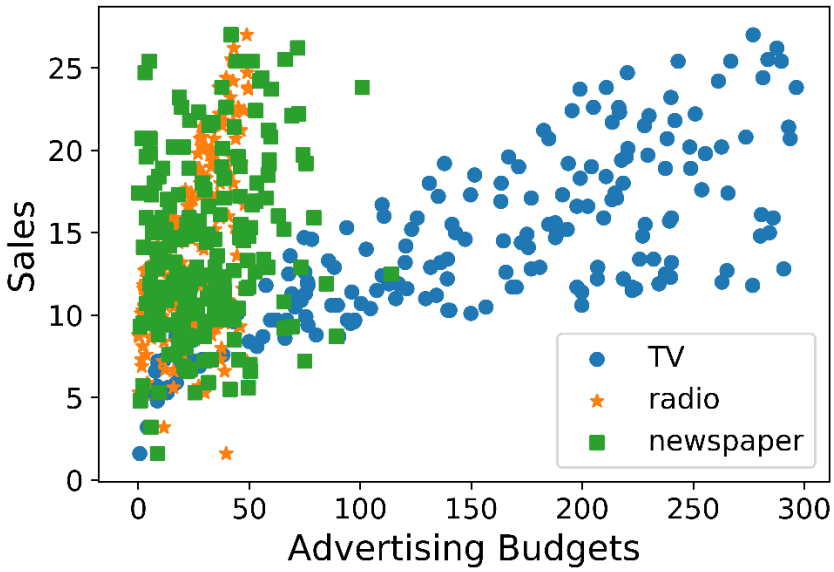

图 3-1 广告数据的散点图

在这个例子中，3 种方式的广告投入，TV、radio、newspaper，是模型的自变量。分别把 TV、radio、newspaper 记为x_1, x_2, x_3，然后把 3 个自变量放在一个行向量中，记为$\boldsymbol{x} = (x_1 \quad x_2 \quad x_3)$。销量是该数据的因变量，记为$y$。数据的每一行表示一个观测点。第一个观测点的输入向量记为$\boldsymbol{x}_1 = (x_{11} \quad x_{12} \quad x_{13})$，第一个观测点的因变量记为$y_1$；第二个观测点输入向量为$\boldsymbol{x}_2 = (x_{21} \quad x_{22} \quad x_{23})$，

第二个观测点的因变量记为y_2，等等。在广告数据中，观测点个数为 200，自变量个数为 3。因此，$n = 200$，$p = 3$。

广告数据的自变量矩阵$\boldsymbol{X}$为，

$$\boldsymbol{X} = (\boldsymbol{x}_1^{\mathrm{T}} \quad \boldsymbol{x}_2^{\mathrm{T}} \quad \cdots \quad \boldsymbol{x}_{200}^{\mathrm{T}})^{\mathrm{T}} = \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ \vdots & \vdots & \vdots \\ x_{n1} & x_{n3} & x_{n3} \end{pmatrix} = \begin{pmatrix} 230.1 & 37.8 & 69.2 \\ 44.5 & 39.3 & 45.1 \\ \vdots & \vdots & \vdots \\ 232.1 & 8.6 & 8.7 \end{pmatrix}$$

广告数据的因变量向量$\boldsymbol{y}$为，

$$\boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{200} \end{pmatrix} = \begin{pmatrix} 22.1 \\ 10.4 \\ \vdots \\ 13.4 \end{pmatrix}$$

当b和$w_1, w_2, \cdots, w_p$已知，它们可以分别记为$\hat{b}$和$\hat{w}_1, \hat{w}_2, \cdots, \hat{w}_p$。这时，只要输入$x_1, x_2, \cdots, x_p$，即可得到因变量预测值，$\hat{y}$，

$$\hat{y} = \hat{b} + \hat{w}_1 x_1 + \hat{w}_2 x_2 + \cdots + \hat{w}_p x_p$$

从上式可以看到，线性回归模型把一个输入的各个自变量的值分别乘以对应的权重，然后把所有乘积的结果相加，再加上截距项，得到回归模型对该输入的预测值。如果我们把截距项的自变量记为$x_0 = 1$，截距项系数记为$w_0 = b$。那么，因变量预测值可以写成

$$\hat{y} = \sum_{j=0}^{p} \hat{w}_j x_j$$

所以，可以简单地认为线性回归模型的预测值是所有自变量的加权和，权重$\hat{w}_0, \hat{w}_1, \cdots, \hat{w}_p$反应了各个自变量对因变量的影响大小和方向。为了方便，在以后的章节中，我们有时候把输入数据的加权和加上截距项简称为加权和。

- 权重绝对值表示对应自变量对因变量的影响大小。例如，如果$\hat{w}_1 = 10$，那么自变量x_1对预测值$\hat{y}$的影响为 10 倍的x_1；如果$\hat{w}_1 = 0.1$，那么自变量x_1对预测值$\hat{y}$的影响为x_1的 1/10。
- 权重的符号表示对应自变量对因变量的影响方向。如果权重是正值，那么增加对应自变量的值会使得因变量预测变大；如果权重是负值，那么增加对应自变量的值会使得因变量的预测变小。

在 Python 中，可以如下定义线性回归模型的预测函数 `linear_model()`。

```
# 线性回归模型的预测函数 linear_model()
```

```
def linear_model(input, weight, b):
    # 计算线性模型的预测值，函数 np.sum()计算数组的元素和
    prediction = np.sum(input * weight) + b
    return prediction
```

例如，当$b=10$，$\boldsymbol{w}=(0.1 \quad 0.1 \quad 0.1)^{\intercal}$时，我们可以如下计算广告数据的第一个观测点的预测值。

```
b = 10                                  # 给定截距项
w = np.array([0.1, 0.1, 0.1])           # 给定自变量权重

# 从数据 advertising 中得到第一个观测点的自变量向量
input_0 = advertising.values[0][0:3]
print("第一个观测点的自变量向量为：" + str(input_0))

# 使用线性回归模型得到 input_0 的预测值
pred = linear_model(input_0, w, b)
print("第一个观测点的预测值为：" + str(pred))

第一个观测点的自变量向量为：(230.1 37.8 69.2)
第一个观测点的预测值为：43.71
```

通过函数 `linear_model()`，得到自变量的值为$(230.1 \quad 37.8 \quad 39.2)$的观测点的预测值$43.71$。现在，一个非常自然的问题是，该预测结果准确吗？为了回答这个问题，我们可以计算预测值与真实值的差，$\hat{y}-y$。如果$\hat{y}-y$是一个正数，说明预测值高估了；如果$\hat{y}-y$是一个负数，说明预测值低估了。总的来说，$\hat{y}-y$的绝对值很大说明预测效果不好，$\hat{y}-y$的绝对值很小说明预测效果好。实际应用中，通常使用残差平方$(\hat{y}-y)^2$来衡量观测点的预测误差。$(\hat{y}-y)^2$越小说明预测效果越好。

```
"""
从数据 advertising 中得到第 1 行，第 4 列的因变量，target
并计算残差平方
"""
target = advertising.values[0][3]
loss = (pred - target) ** 2
print(loss)

466.9921
```

对于数据$(\boldsymbol{x}_1,y_1),\cdots,(\boldsymbol{x}_n,y_n)$，我们可以计算每一个观测点的预测值$\hat{y}_1=b+\boldsymbol{x}_1\boldsymbol{w},\cdots,\hat{y}_n=b+\boldsymbol{x}_n\boldsymbol{w}$与真实因变量$y_1,\cdots,y_n$的残差平方，并且定义残差平方和，$\mathrm{RSS}(b,\boldsymbol{w})$，从而衡量模型总的预测误差。

$$\mathrm{RSS}(b,\boldsymbol{w})=\frac{1}{2n}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2=\frac{1}{2n}\sum_{i=1}^{n}((b+\boldsymbol{x}_i\boldsymbol{w})-y_i)^2$$

这里，残差平方和乘以常数$\frac{1}{2}$只是为了稍后估计b和$\boldsymbol{w}$方便一些。当$b=10$，$\boldsymbol{w}=(0.1 \quad 0.1 \quad 0.1)^{\intercal}$，可以使用如下代码计算整个广告数据的残差平方和。

```
"""
计算广告数据的残差平方和
"""
rss = 0
for i in range(len(advertising)):
    # input 为自变量向量
    input = advertising.values[i][0:3]
    # 调用函数 linear_model()得到预测值
    pred = linear_model(input, w, b)
    # target 为第 i 个观测值的真实销量
    target = advertising.values[i][3]
    # 计算残差平方和
    rss += (pred - target) ** 2/2/len(advertising)
print(rss)

143.69051025000013
```

可以看到，当$b=10$，$\boldsymbol{w}=(0.1 \quad 0.1 \quad 0.1)^{\mathrm{T}}$，整个数据的残差平方和为143.69。这里$b=10$，$\boldsymbol{w}=(0.1 \quad 0.1 \quad 0.1)^{\mathrm{T}}$是随意给定的，算出来的残差平方和比较大是预料之内的。在建模过程中，我们希望可以让模型学习数据中蕴含的规律，从而找到更好的b和$\boldsymbol{w}$，最终得到更加准确的预测。而$\mathrm{RSS}(b,\boldsymbol{w})$可以用来衡量模型预测误差大小，因此一个很自然的方法是最小化$\mathrm{RSS}(b,\boldsymbol{w})$得到$b$和$\boldsymbol{w}$的估计，即

$$\underset{b,\boldsymbol{w}}{\text{minimize}} \quad \mathrm{RSS}(b,\boldsymbol{w})=\frac{1}{2n}\sum_{i=1}^{n}((b+\boldsymbol{x}_i\boldsymbol{w})-y_i)^2$$

$\mathrm{RSS}(b,\boldsymbol{w})$也称为目标函数或者损失函数。当给定$\hat{b}$和$\hat{\boldsymbol{w}}$时，$\mathrm{RSS}(\hat{b},\hat{\boldsymbol{w}})$也称为预测误差或者模型误差。在数学方法中，求$\mathrm{RSS}(b,\boldsymbol{w})$最小值的方法有很多。最常见的方法是求$\mathrm{RSS}(b,\boldsymbol{w})$关于$b$和$\boldsymbol{w}$的偏导数，即$\frac{\partial \mathrm{RSS}(b,\boldsymbol{w})}{\partial b}$和$\frac{\partial \mathrm{RSS}(b,\boldsymbol{w})}{\partial \boldsymbol{w}}$，然后再令这些偏导数等于 0，解方程得到$b$和$\boldsymbol{w}$的估计值。在这里，我们不详细介绍该方法，因为该方法只适合少数结构比较简单的模型（例如，这里的线性回归模型），不能用于求解像深度学习这类复杂模型的参数。我们着重介绍的算法是深度学习中常用的优化算法：梯度下降法。我们将逐步介绍梯度下降法的 3 个不同变体：随机梯度下降法、全数据梯度下降法和批量随机梯度下降法。

3.1.2 随机梯度下降法

我们先考虑一个简单情况，没有截距项而只有一个自变量的线性回归模型，而且假设数据只有一个观测点（$x=0.5, y=0.8$）。线性回归模型为$y=xw+\epsilon$，残差平方和为

$$\mathrm{RSS}(w)=\frac{1}{2}(xw-y)^2$$

RSS(w)是一个关于w的一元二次函数。当观测点为($x=0.5$，$y=0.8$)，$w=3$时，在 Python 中，可以通过如下方式计算残差平方和。

```
x, y = 0.5, 0.8
w = 3                          # 权重设为 3
rss = (y - x * w)**2 / 2       # 计算残差平方和
print(rss)

0.24499999999999997
```

在梯度下降法中，很重要的一个概念是梯度（gradient），记为∇。例如，函数RSS(w)关于参数w的梯度，记为∇_wRSS(w)；在不引起混淆的情况下，可以更加简洁的记为∇_w。函数RSS(w)的梯度为RSS(w)关于w的偏导数，

$$\nabla_w \mathrm{RSS}(w) = \frac{\partial \mathrm{RSS}(w)}{\partial w}$$

在本节简单的例子中，

$$\nabla_w \mathrm{RSS}(w) = \frac{\partial \frac{1}{2}(y - xw)^2}{\partial w} = (xw - y)x$$

当$w=3$时，$\nabla_w \mathrm{RSS}(w) = (0.5 \times 3 - 0.8) \times 0.5 = 0.35$。图 3-2 用橙色虚线表示RSS($w$)在$w=3$处的切线，切线的斜率即为梯度。可以看到，沿着梯度方向（当$w=3$时，梯度为正值），函数RSS(w)增长最快；沿着梯度相反方向，函数RSS(w)下降最快。

```
w = 3
pred = x * w
rss = ((pred - y)**2) / 2
grad = (pred - y) * x    # 计算梯度
print("当 w=3，预测值为：" + str(pred))
print("当 w=3，残差平方和为：" + str(round(rss, ndigits=3)))
print("当 w=3，RSS(w)的梯度：" + str(grad))

当 w=3，预测值为：1.5
当 w=3，残差平方和为：0.245
当 w=3，RSS(w)的梯度：0.35

"""
计算不同权重下，残差平方和 RSS(w)的值
"""

# 函数 np.linspace()可以给出给定区间内等间隔的序列
w_vec = np.linspace(-1, 4, 100)
rss_vec = []
for w_tmp in w_vec:
    rss_tmp = (y - x * w_tmp)**2/2
    rss_vec.append(rss_tmp)
"""
```

```
画出残差平方和随着权重变化的曲线
画出 w=3 时，RSS(w)的斜率
"""
plt.plot(w_vec, rss_vec)

# 画出 w=3，及对应 rss 的散点图
plt.scatter(w, rss, s=100, c="y", marker="o")
# 通过 w=3 的切线
plt.plot(np.linspace(2.5,3.5,50), \
         np.linspace(2.5,3.5,50)*0.35-0.805,'--',linewidth=2.0)
plt.xlabel("w", fontsize=16)
plt.ylabel("RSS", fontsize=16)
plt.show()
```

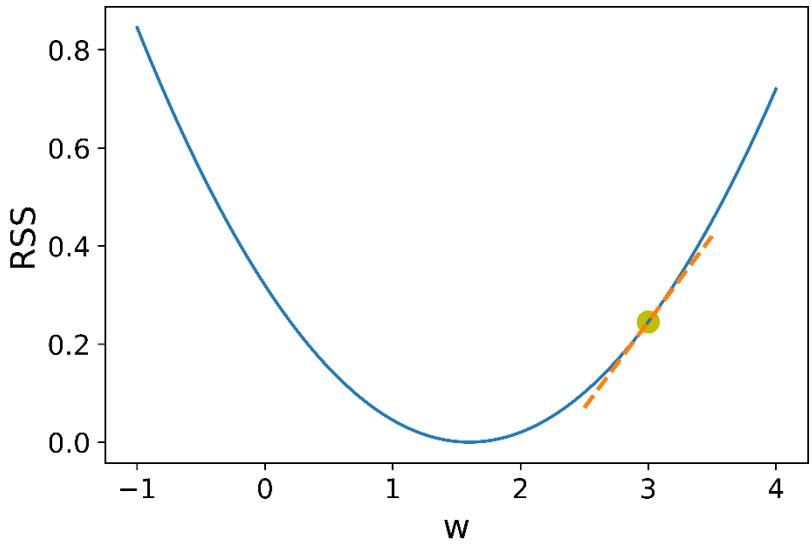

图 3-2 RSS(w)在$w = 3$的切线

进一步分析预测误差、预测值以及权重之间的关系。当$w = 3$时，预测值 pred 为1.5，预测值 1.5 大于真实因变量的值 0.8，预测误差为 0.245。为了减小预测误差，需要减小预测值；因为 pred 等于x乘以w，一个直接的想法是让w减去一个正的数，使得w的值变小，这样可以使得预测值变小，从而最终减小预测误差。而梯度$\text{grad} = (\text{pred} - y) \times x = 0.35$恰巧是一个正数（这里的 grad，其实不是“恰巧”是正数。而是因为 grad 表示梯度，而梯度表示函数增长最快的方向；沿着梯度相反的方向，函数下降的最快）。$w - \text{grad}$会使得w变小，这正是我们所期望的。

当$w = 0$时，$\nabla_w \text{RSS}(w) = (0.5 \times 0 - 0.8) \times 0.5 = -0.4$。图 3-3 用橙色虚线表示RSS($w$)在$w = 0$处的切线，切线的斜率即为梯度。可以看到，沿着梯度方向（当$w = 0$时，梯度为负值），函数RSS(w)增长最快；沿着梯度相反方向，函数RSS(w)下降最快。

```
w = 0
pred = x * w
rss = ((pred - y)**2) / 2
grad = (pred - y) * x
print("当 w=0，预测值为：" + str(pred))
print("当 w=0，残差平方和为：" + str(round(rss, ndigits=3)))
print("当 w=0，RSS(w)的梯度：" + str(grad))

当 w=0，预测值为：0.0
当 w=0，残差平方和为：0.32
当 w=0，RSS(w)的梯度：-0.4

"""
画出残差平方和随着权重的变化的曲线
画出 w=0 时，RSS(w)的斜率
"""
plt.plot(w_vec, rss_vec)
plt.scatter(0, y**2/2, s=100,c="y", marker="o")
plt.plot(np.linspace(-0.5, 0.5, 50), \
         np.linspace(-0.5, 0.5, 50) * (-0.4) + 0.32, \
         '--', linewidth=2.0)
plt.xlabel("w", fontsize=16)
plt.ylabel("RSS", fontsize=16)
plt.show()
```

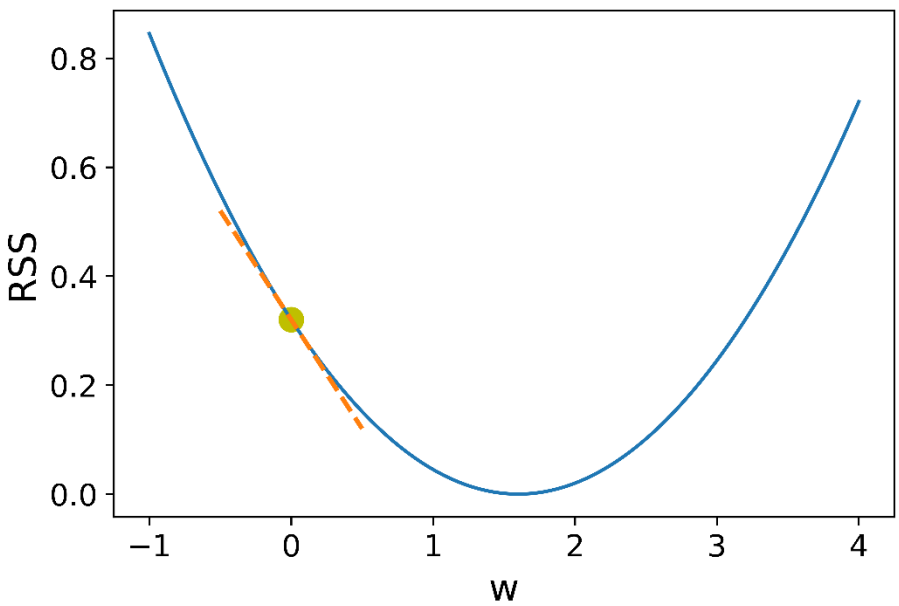

图 3-3 RSS(w)在$w = 0$的切线

当$w = 0$时，预测值 pred 为0，预测值0小于真实因变量的值0.8，预测误差为0.32。为了减小预测误差，需要增大预测值；因为预测值等于x乘以w，一个直接的想法是让w减去一个负数，使得w的值变大，这样就可以使得预测值变大，

最终减小预测误差。而梯度$\text{grad} = (0 - 0.8) \times 0.5 = -0.4$恰巧是一个负数。 w - grad 会使得w变大，这也正是我们所期望的。

可以看到，无论w的初始值在最小值的左边还是右边，$w = w - \nabla_w \text{RSS}(w)$都可以让$w$朝着使$\text{RSS}(w)$变小的方向移动。基于这样的观察，我们可以设计下面算法找到$\text{RSS}(w)$的最小值点。

算法 3.1 梯度下降法（单个自变量，单个样本点）

给定初始的参数值w，学习步长α

迭代直至收敛：

计算观测点的梯度$\nabla_w \text{RSS}(w)$

更新w：$w = w - \alpha \nabla_w \text{RSS}(w)$

该算法就是梯度下降法。在更新w的过程中，通常不直接让参数w减去$\text{RSS}(w)$；而是让参数w减去$\alpha \text{RSS}(w)$。这里，α是一个较小的正数，称为学习步长。这样可以让w的每次更新速度变慢，使得w更容易收敛。

对于本节的简单例子，我们尝试迭代多次，观察每次更新w后，$\text{RSS}(w)$是否变小。令w的初始值为0.0，那么初始预测值为0.0，初始误差为0.32。预测值 0 小于真实因变量的值 0.8。这时w需要减去一个负数，使得w的值变大，这样可以使预测值变大，更接近真实因变量，最终减小预测误差。初始梯度为$\nabla_w \text{RSS}(w) = (0 - 0.8) * 0.5 = -0.4$。

```
"""
初始权重，初始预测值，预测误差和初始梯度
"""
w = 0; lr = 0.5  # lr 为学习步长
pred = x * w
loss = ((pred - y)**2) / 2
grad = (pred - y) * x
print("自变量的值：" + str(x))
print("真实因变量：" + str(y))
print("初始权重：" + str(w))
print("初始预测：" + str(pred))
print("初始误差：" + str(round(loss, ndigits=3)))
print("初始梯度：" + str(grad))

自变量的值：0.5
真实因变量：0.8
初始权重：0
初始预测：0.0
初始误差：0.32
初始梯度：-0.4
```

第一次迭代，$w = 0 - 0.5 \times (-0.4) = 0.2$。第一次迭代后，预测值为 0.1，预测误差为 0.245。预测误差变小了。预测值还是小于真实因变量的值 0.8，依然

可以让w减去一个负数，使得w的值变大，这样可以使预测值变大，从而更接近真实因变量，进一步减小预测误差。第一次迭代后，梯度$\nabla_w \text{RSS}(w) = (0.1 - 0.8) \times 0.5 = -0.35$。

```
"""
第一次迭代，以及计算迭代后的预测值，预测误差和梯度
"""
w = w - lr * grad   # 更新 w
pred = x * w
loss = ((pred - y)**2) / 2
grad = (pred - y) * x
print("第一次迭代后的权重：" + str(w))
print("第一次迭代后的预测：" + str(pred))
print("第一次迭代后的误差：" + str(round(loss, ndigits=3)))
print("第一次迭代后的梯度：" + str(round(grad, ndigits=3)))

第一次迭代后的权重：0.2
第一次迭代后的预测：0.1
第一次迭代后的误差：0.245
第一次迭代后的梯度：-0.35
```

第二次迭代，$w = 0.2 - 0.5 \times (-0.35) = 0.375$。第二次迭代后，预测值为 0.1875，预测误差为 0.188。预测误差进一步变小了。预测值还是小于真实因变量的值 0.8，依然可以让w减去一个负数，使得w的值变大，这样可以使预测值变大，更接近真实因变量，从而进一步减小预测误差。第二次迭代后，梯度$\nabla_w \text{RSS}(w) = (0.1875 - 0.8) \times 0.5 = -0.306$。

```
"""
第二次迭代，以及计算迭代后的预测值，预测误差和梯度
"""
w = w - lr * grad           # 更新 w
pred = x * w
loss = ((pred - y)**2) / 2
grad = (pred - y) * x
print("第二次迭代后的权重：" + str(w))
print("第二次迭代后的预测：" + str(pred))
print("第二次迭代后的误差：" + str(round(loss, ndigits=3)))
print("第二次迭代后的梯度：" + str(round(grad, ndigits=3)))

第二次迭代后的权重：0.375
第二次迭代后的预测：0.1875
第二次迭代后的误差：0.188
第二次迭代后的梯度：-0.306
```

从上面结果可以看到，随着迭代次数增加，w不断增大，预测值也不断增大，预测值与因变量真实值的差距越来越小，预测误差越来越小。我们可以继续重复上面的步骤，找到使得$\text{RSS}(w)$最小的w。在 Python 中，可以使用 for 循环如下实现梯度下降法。

```
"""
使用 for 循环，对 w 迭代 20 次
```

```
"""
w, lr = 0, 0.5          # 给定权重初始值为 0，学习步长设为 0.5
w_record = []           # w_record 用于记录迭代过程的权重
loss_record = []        # loss_record 用于记录迭代过程的残差平方和

# for 循环，函数 range()可以产生数列[0,1,2,...,19]
for iter in range(20):
    pred = x * w
    loss = ((pred - y)**2) / 2

    w_record.append(w)
    loss_record.append(loss)

    delta = pred - y
    w = w - lr * (delta * x)    # delta * x 表示梯度
    if (iter%5==0 or iter==19):
        print("iter: %2d; Loss: %0.3f" % (iter, loss))

w_record.append(w)
loss_record.append((x * w - y)**2/2)

iter: 0;    Loss: 0.320
iter: 5;    Loss: 0.084
iter: 10;   Loss: 0.022
iter: 15;   Loss: 0.006
iter: 19;   Loss: 0.002
```

从代码运行结果以及图 3-4 可以看到，预测误差RSS(w)随着w的更新逐步变小。最终，w收敛到1.49，RSS(w)的最小值为0.002。

```
"""
画出权重 w 与残差平方和 RSS(w)随着迭代的变化曲线
"""
plt.plot(w_vec, rss_vec)
plt.scatter(0, 0.32, s=100, c="y", marker="o")  # 画初始点
for i in range(len(w_record)-1):
    # 画箭头
    plt.arrow(w_record[i], loss_record[i], \
              w_record[i+1]-w_record[i],   \
              loss_record[i+1]-loss_record[i], width=0.01, \
              color="y", head_width=0.05)
plt.xlabel("w", fontsize=16)
plt.ylabel("RSS", fontsize=16)
plt.show()
```

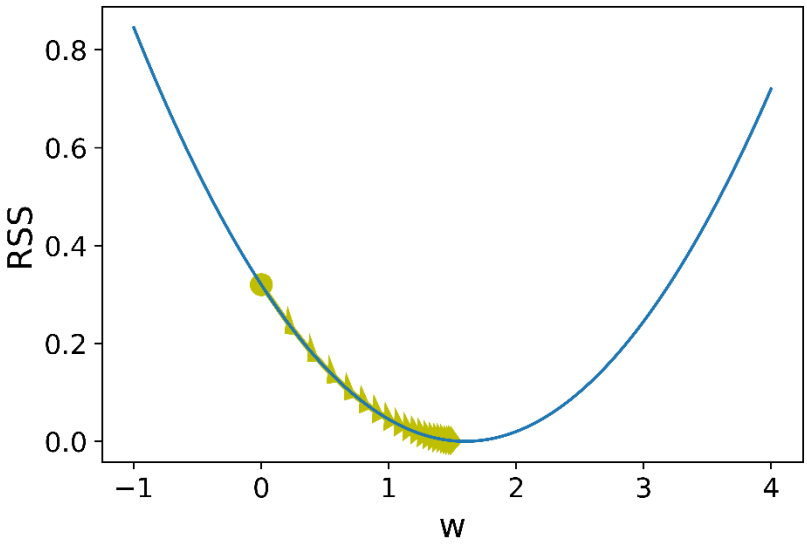

图 3-4 RSS(w)的收敛路径

通常情况下，数据包含多个自变量，损失函数将包含多个参数。记$L(\boldsymbol{w}) = L(w_1, \cdots, w_p)$为一个包含$p$个参数的损失函数，其梯度可以表示为：

$$\nabla_{\boldsymbol{w}} L(\boldsymbol{w}) = \begin{pmatrix} \dfrac{\partial L(\boldsymbol{w})}{\partial w_1} \\ \vdots \\ \dfrac{\partial L(\boldsymbol{w})}{\partial w_p} \end{pmatrix}$$

函数$L(\boldsymbol{w})$的梯度是由函数$L(\boldsymbol{w})$对各个参数的偏导数构成的向量。

而且，现实数据会有很多观测点。这时，我们可以计算每个观测点的预测值，计算梯度，然后更新$\boldsymbol{w}$：$\boldsymbol{w} = \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} L(\boldsymbol{w})$。当数据包含多个自变量和多个观测点时，我们可以设计下面算法来找到 $L(\boldsymbol{w})$的最小值点。

算法 3.2 随机梯度下降法

给定初始参数值$\boldsymbol{w} = (w_1, w_2, \ldots, w_p)$，学习步长$\alpha$
迭代直至收敛：
　　对数据集中的每个观测点
　　　　计算该数据点的梯度, $\nabla_{\boldsymbol{w}} L(\boldsymbol{w})$
　　　　更新$\boldsymbol{w}$：$\boldsymbol{w} = \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} L(\boldsymbol{w})$

该算法就是随机梯度下降法（Stochastic Gradient Descent, SGD）。随机指每次只使用一个观测点计算梯度；而且在实现随机梯度下降法过程中，可以随机抽取观测点来计算梯度并更新参数。

在广告数据中，我们以 TV 和 radio 为自变量，sales 为因变量建立没有截距项的线性回归模型，并且使用所有观测点组成的数据集作为训练数据。首先，分别对自变量进行标准化，对因变量进行中心化。标准化可以使得变换之后所有自变量的均值为 0，方差为 1。中心化可以使得因变量变换之后均值为 0。自变量标准化和因变量中心化可以让梯度下降法的数值表现更加稳定，也更容易找到合适的初始值和学习步长。标准化方法之一是让数据的每一列减去该列的均值，然后除以该列的样本标准差，即

$$\text{scaled\_x} = \frac{x - \overline{x}}{\text{sd(x)}}$$

中心化只需要让数据减去样本均值，即$\text{centered_y} = y - \overline{y}$。

```
"""
自变量矩阵 x，因变量向量 y
对数据进行标准化和中心化得到 scaled_x 和 centered_y
"""
x = advertising.iloc[:,0:2].values
y = advertising.iloc[:,3].values
scaled_x = (x - np.mean(x, axis=0, keepdims=True))/  \
             np.std(x, axis=0, keepdims=True)
centered_y = y - np.mean(y)

"""
使用随机梯度下降法迭代更新 w
"""
lr = 0.1
w = np.zeros(2)
w_record = [w.copy()]
for iter in range(5):                          # 所有的数据点重复使用 5 次
    total_loss = 0                             # 总的预测误差
    for i in range(len(scaled_x)):
        # 计算每个观测点的预测值
        pred = np.sum(scaled_x[i] * w)
        # 计算每个观测点的预测误差，并把该误差加入到总的预测误差中
        total_loss += ((pred - centered_y[i])**2) / 2
        delta = (pred - centered_y[i])
        w -=  lr * (delta * scaled_x[i])             # 更新 w
        w_record.append(w.copy())

    print("Loss: %0.5f" % (total_loss/(i+1)))

Loss: 1.98663
Loss: 1.62702
Loss: 1.62702
Loss: 1.62702
```

```
Loss: 1.62702
w
array([3.46941055, 3.19188802])
```

从上面结果可以看到，通过所有观测点的迭代，并且重复使用所有观测点 5 次之后，参数$\boldsymbol{w}$对应的预测误差先是逐步的变小，之后大致稳定在 1.627 附近。参数$\boldsymbol{w}$的估计值为 3.47，3.19。所以，随着 TV 和 radio 的增加，sales 也会增加，即 TV 和 radio 对销量都有正面影响。图 3-5 的黑色实线表示等高线，同一个椭圆形上的(w_1, w_2)组合的RSS($\boldsymbol{w}$)相等。椭圆形越大，RSS($\boldsymbol{w}$)越大；椭圆形越小，RSS($\boldsymbol{w}$)越小。红色星号表示真正的RSS($\boldsymbol{w}$)最小值（通过解方程的方式求得），蓝色圆点表示通过随机梯度下降法得到的(w_1, w_2)的估计值。带有箭头的黄色线表示(w_1, w_2)的迭代路径。随机梯度下降法中，每次迭代只使用一个观测点，计算的梯度随机性比较大，有时候(w_1, w_2)的值不会朝着使得RSS($\boldsymbol{w}$)最小的方向移动。

```
"""
画出 RSS(w) 的等高线和 (w1,w2) 的迭代路径
这部分代码用于画出图 3-5，同学们只需要看懂图 3-5，可以不用看代码的细节
"""
def f(w1, w2, x, y):
    pred = x[:,0] * w1 + x[:, 1] * w2
    return np.mean((pred - y)**2)

a = np.linspace(0, 6, 300)
b = np.linspace(0, 4, 300)

A, B = np.meshgrid(a, b)

C = np.zeros_like(A)
for i in range(A.shape[0]):
    for j in range(A.shape[1]):
        C[i][j] = f(A[i][j], B[i][j], scaled_x, centered_y)

w_equation = np.dot(np.dot(np.linalg.inv(np.dot(scaled_x.T, \
                    scaled_x)), scaled_x.T), centered_y)

plt.contour(A, B, C, colors='black')
for i in range(len(w_record)-1):
    plt.arrow(w_record[i][0], w_record[i][1], \
              w_record[i+1][0]-w_record[i][0], \
              w_record[i+1][1]-w_record[i][1],  \
              width=0.01, color="y", head_width=0.05, zorder = 1)
plt.scatter(w_record[-1][0], w_record[-1][1], s=50, c="b", \
            marker="o", zorder=2)
plt.scatter(w_equation[0], w_equation[1], s=100, c="r", \
            marker="*", zorder=2)
plt.xlabel("w1", fontsize=16)
plt.ylabel("w2", fontsize=16)
plt.show()
```

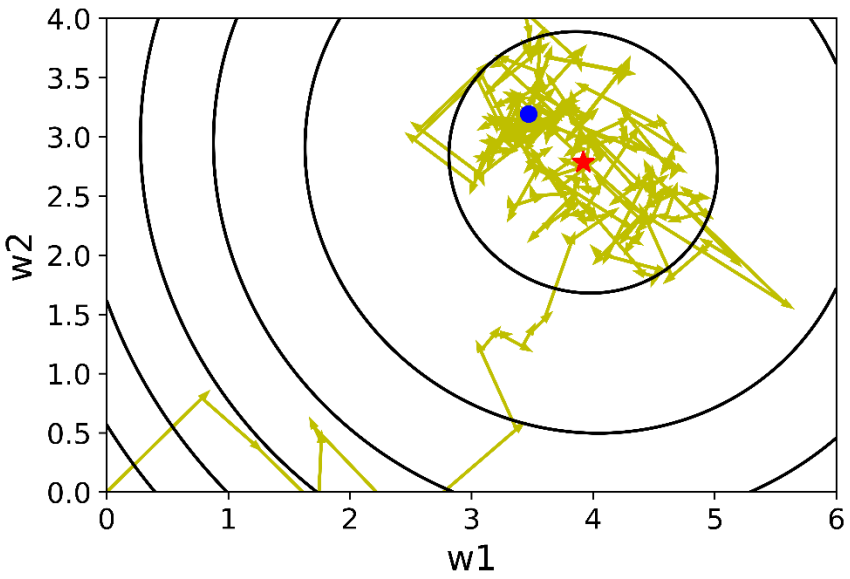

图 3-5 RSS($\boldsymbol{w}$)的等高线及参数$\boldsymbol{w}$收敛路径（随机梯度下降法）

下一小节将介绍全数据梯度下降法，该方法在计算梯度时用到所有观测点，因此计算出来的梯度会比较稳定，参数$\boldsymbol{w}$可以更快收敛到使得RSS($\boldsymbol{w}$)最小的点。

3.1.3 全数据梯度下降法

在这里，函数$L(\boldsymbol{w})$表示使用数据所有观测点得到的损失函数，$\nabla_{\boldsymbol{w}}L(\boldsymbol{w})$为$L(\boldsymbol{w})$的梯度。全数据梯度下降法（Full Gradient Descent）的主要步骤为：

算法 3.3 全数据梯度下降法

给定初始参数值$\boldsymbol{w} = (w_1, w_2, \cdots, w_p)$，学习步长$\alpha$
迭代直至收敛：
　　计算全数据集的梯度$\nabla_{\boldsymbol{w}}L(\boldsymbol{w})$
　　更新$\boldsymbol{w}$：$\boldsymbol{w} = \boldsymbol{w} - \alpha\nabla_{\boldsymbol{w}}L(\boldsymbol{w})$

为了方便说明全数据梯度下降法的特点，我们先考虑只有一个自变量的情况。这时，残差平方和为

$$\mathrm{RSS}(w) = \frac{1}{2n}\sum_{i=1}^{n}(x_i w - y_i)^2 = \frac{1}{2n}(\boldsymbol{X}w - \boldsymbol{y})^{\mathrm{T}}(\boldsymbol{X}w - \boldsymbol{y})$$

RSS(w)是w的一元二次函数。 在广告数据中，现在只考虑 TV 对 sales 的影响。首先，对自变量进行标准化，对因变量进行中心化。

```
"""
```

```
以 TV 为自变量，sales 为因变量
对 TV 标准化，对 sales 中心化
"""
TV = advertising["TV"].values
sales = advertising["sales"].values
scaled_TV = (TV - np.mean(TV))/np.std(TV)
centered_sales = sales - np.mean(sales)
```

图 3-6 画出了广告数据中RSS(w)随着w变化的曲线。函数RSS(w)关于w的偏导数为

$$\frac{\partial \mathrm{RSS}(w)}{\partial w}=\frac{1}{n}\sum_{i=1}^{n}(x_i w-y_i)x_i=\frac{1}{n}\boldsymbol{X}^{\mathrm{T}}(\boldsymbol{X}w-\boldsymbol{y})$$

图 3-6 的圆点为随意给定的w的初始值；经过圆点的虚线是RSS(w)在$w=0$处的切线。星号是RSS(w)的最小值点。令$\frac{\partial \mathrm{RSS}(w)}{\partial w}=0$，求解方程得到RSS($w$)的$w_{min}=\frac{\sum_{i=1}^{n}x_i y_i}{\sum_{i=1}^{n}x_i^2}$。在后面将要学习的深度学习模型中，我们无法通过数学解析的方式求得参数估计值；这里只是为了先标示出梯度下降法的目标，所以用数学解析的方法求出了RSS(w)的最小值。

```
"""
画出 RSS 随着 w 变化的曲线
画出 RSS 在 w=0 处的切线；画出 RSS 的最小值点
"""
n = len(scaled_TV)
w_vec = np.linspace(-2, 10, 100)                    # 权重向量
rss_vec = []                                        # 记录不同权重下，RSS 值
for w in w_vec:
    rss = np.sum((centered_sales - scaled_TV * w)**2)/2/n
    rss_vec.append(rss)

w_0 = 0                                             # 初始权重设为 0
# 初始权重的残差平方和
rss_0 = np.sum((centered_sales - scaled_TV * w_0)**2)/2/n
# RSS 最小值点，对应的权重
w_min = np.sum(scaled_TV * centered_sales)/np.sum(scaled_TV **2)
# RSS 最小值
rss_min = np.sum((centered_sales - scaled_TV * w_min)**2)/2/n

plt.plot(w_vec, rss_vec)
plt.scatter(w_0, rss_0, s=100,c="y", marker="o")
plt.scatter(w_min, rss_min, s=100, c="r", marker="*")
plt.plot(np.linspace(-1,1,50), np.linspace(-1,1,50)*  \
         np.mean(-scaled_TV*centered_sales)+ \
         np.sum((centered_sales)**2)/2/n, '--', linewidth=2.0)
plt.xlabel("w", fontsize=16)
plt.ylabel("RSS", fontsize=16)
plt.show()
```

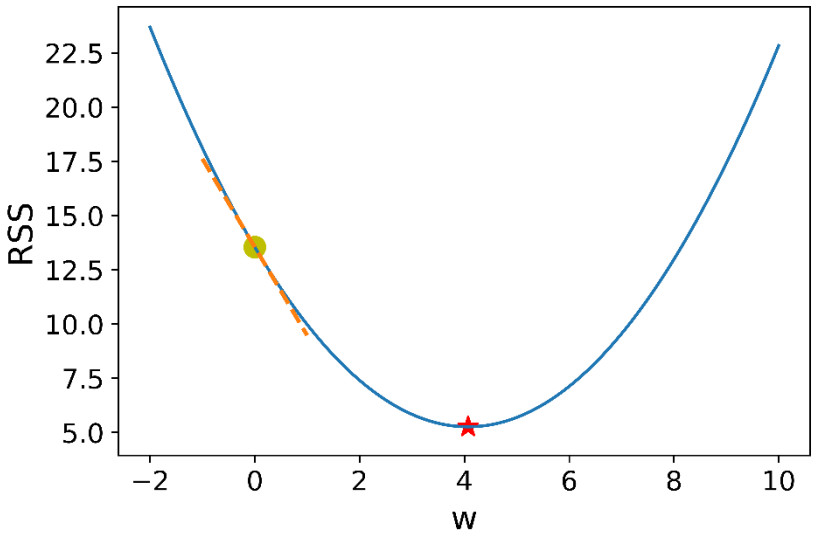

图 3-6 RSS(w)函数曲线和RSS(w)在$w = 0$处的导数

在 Python 中，可以使用下面的代码来实现全数据梯度下降法。在代码中，`delta` 是一个长度为200的向量，表示预测值与真实值的差；`np.sum(input * delta)`计算$\boldsymbol{X}^{\mathrm{T}}(\boldsymbol{X}w - \boldsymbol{y})$。

```
"""
使用全数据梯度下降法迭代更新 w
"""
w, lr = 0, 0.1
input, target = scaled_TV, centered_sales
w_record = []
loss_record = []
for iter in range(20):
    pred = input * w                          # 所有观测点的预测值
    loss = np.sum((pred - target)**2)/2/n  # 全数据预测误差

    w_record.append(w)
    loss_record.append(loss)

    delta = pred - target
    w = w - lr * np.sum(input * delta) / n      # 更新权重 w

    if (iter%5==0 or iter==19):
        print("iter: %3d; Loss: %0.3f"%(iter, loss))

w_record.append(w)
loss_record.append(np.sum((pred - target)**2)/2/n)

iter: 0;     Loss: 13.543
iter: 5;     Loss: 8.146
iter: 10;    Loss: 6.264
iter: 15;    Loss: 5.608
iter: 19;    Loss: 5.408
```

从上面结果可以看到，预测误差随着迭代逐步变小。从图 3-7 也可以看到，参数w不断地朝着RSS(w)的最小值点（红色星号）移动；算法开始时w更新幅度较大，接近最小值点时，w更新幅度越来越小，这是因为随着w靠近最小值点，RSS(w)关于w的梯度的绝对值越来越小。

```
"""
画出权重 w 与残差平方和 RSS(w) 随着迭代的变化曲线
"""
plt.plot(w_vec, rss_vec)
plt.scatter(0, rss_0, s=100, c="y", marker="o")
for i in range(len(w_record)-1):
    plt.arrow(w_record[i], loss_record[i], \
              w_record[i+1]- w_record[i], \
              loss_record[i+1]-loss_record[i], \
              width=0.1, color="y", head_width=0.4)

plt.scatter(w_min, rss_min, s=100, c="r", marker="*")
plt.xlabel("w", fontsize=16)
plt.ylabel("RSS", fontsize=16)
plt.show()
```

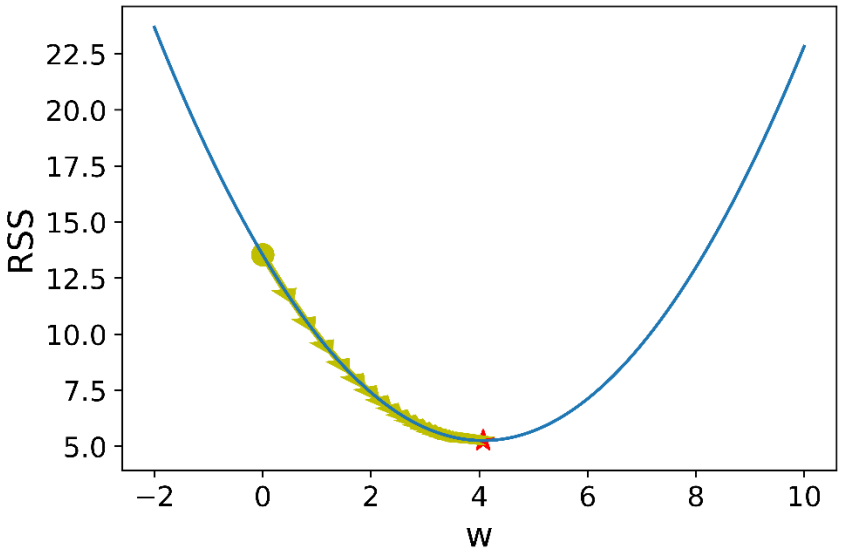

图 3-7 RSS(w)的收敛路径（全数据梯度下降法）

图 3-8 以 TV 为横轴，以 sales 为纵轴画散点图；实线表示回归直线。这里要注意，在建立模型时，我们对数据的自变量做了标准化，并且对因变量做了中心化；使用模型进行预测时，要把预测值变换回原来的数值范围。首先取 0 到 300 之间的等间隔的 100 个数字（把这 100 个数字当成是需要预测的 TV 的值），然后对这 100 个数字标准化（减去 TV 的均值再除以 TV 的标准差）。标准化后的 100 个数字乘以权重是标准化之后的预测值，这些预测值加上 sales 的均值才是最终的预测值（请想想为什么这样做可以得到最终预测值）。

```
"""
计算预测值，需要先对自变量标准化，代入模型，
然后再把预测值变换回原来的数值范围
"""
xx = np.linspace(0, 300, 100)                    # 拟预测的 TV 值
scaled_xx = (xx - np.mean(TV))/np.std(TV)        # 拟预测的 TV 值标准化
# 计算预测值，并把预测值变换回原来的数据范围
yy = (scaled_xx * w) + np.mean(sales)

"""
画出散点图和拟合的直线
"""
plt.figure()
plt.scatter("TV", "sales", data = advertising, label="TV")
plt.plot(xx, yy, "r")
plt.xlabel("TV", fontsize=16)
plt.ylabel("Sales", fontsize=16)
plt.show()
```

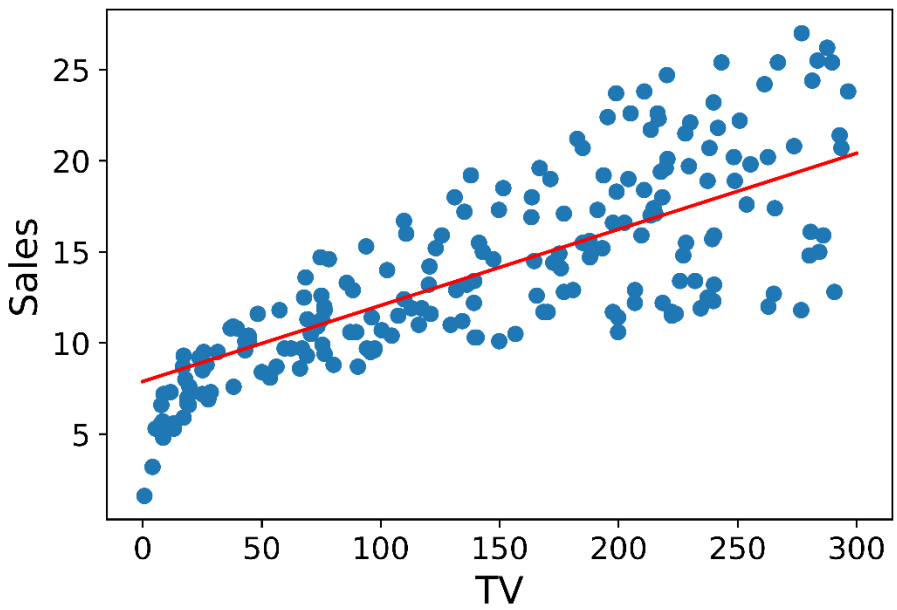

图 3-8 散点图（TV 为横轴，sales 为纵轴）和拟合直线

现在以 TV 和 radio 为自变量，sales 为因变量建立没有截距项的线性回归模型。该线性回归模型的损失函数为

$$\mathrm{RSS}(\boldsymbol{w}) = \frac{1}{2n}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^{\mathrm{T}}(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})$$

这里，$\boldsymbol{X}$包含两列，分别是 TV 和 radio；$\boldsymbol{w}$有两个元素，分别是 TV 和 radio 的权重。从下面代码运行结果以及图 3-9 可以看到，全数据梯度下降法可以更容易达到$\mathrm{RSS}(\boldsymbol{w})$的最小值点，且收敛过程更加稳定。

```
"""
以 TV 和 radio 为自变量建立线性模型，使用全数据梯度下降法迭代权重 w
"""
x = advertising.iloc[:,0:2].values
y = advertising.iloc[:,3].values
# 自变量标准化
scaled_x = (x - np.mean(x, axis=0, keepdims=True))/  \
             np.std(x, axis=0, keepdims=True)
 #因变量中心化
centered_y = y - np.mean(y)
w = np.zeros(2)
lr = 0.1
n = len(scaled_x)
w_record = [w.copy()]
loss_record = []
for iter in range(40):
    pred = np.dot(scaled_x, w)                    # 计算预测值
    loss = np.sum((pred - centered_y)**2)/2/n  # 计算损失函数

    w_record.append(w.copy())
    loss_record.append(loss)

    delta = pred - centered_y                     # 计算 delta
    w = w - lr * np.dot(scaled_x.T, delta)/n   # 更新权重 w

    if (iter % 5==0 or iter==19):
        print("iter: %2d; loss: %0.5f" % (iter, loss))

w_record.append(w.copy())
loss_record.append(np.sum((np.dot(scaled_x, w) - \
                                        centered_y)**2)/2/n)

iter: 0;  loss: 13.54287
iter: 5;  loss: 5.39080
iter: 10; loss: 2.70865
iter: 15; loss: 1.82585
iter: 19; loss: 1.57067
iter: 20; loss: 1.53516
iter: 25; loss: 1.43939
iter: 30; loss: 1.40783
iter: 35; loss: 1.39742

"""
画出 RSS(w) 的等高线和 (w1,w2) 的迭代路线
这部分代码为了是画出图 3-9，同学们只需要看懂图 3-9，可以不看代码细节
"""
plt.contour(A, B, C, colors='black')
for i in range(len(w_record)-1):
    plt.arrow(w_record[i][0], w_record[i][1],   \
              w_record[i+1][0]-w_record[i][0],
              w_record[i+1][1]-w_record[i][1], width=0.01, \
              color="y",head_width=0.05)
plt.scatter(w_record[-1][0], w_record[-1][1], s=50, c="b", \
            marker="o", zorder = 2)
plt.scatter(w_equation[0], w_equation[1], s=50, c="r", \
```

```
            marker="*", zorder = 2)
plt.xlabel("w1", fontsize=16)
plt.ylabel("w2", fontsize=16)
plt.show()
```

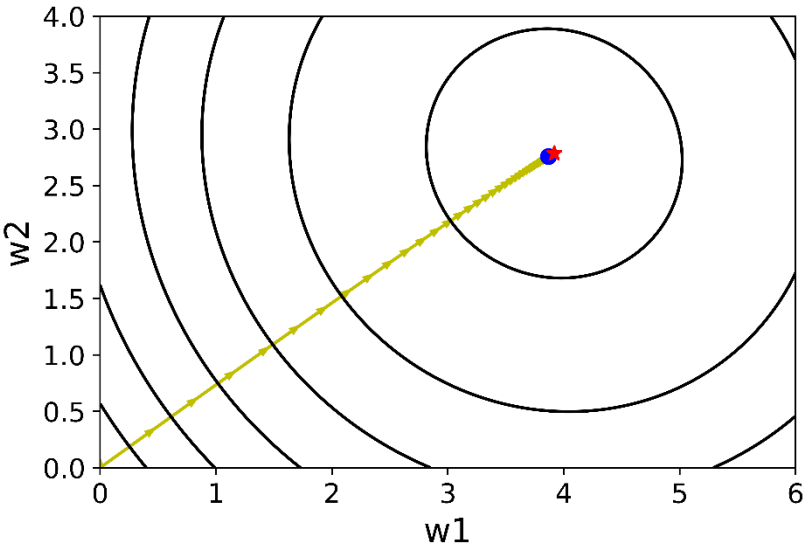

图 3-9 RSS($\boldsymbol{w}$)的等高线及参数$\boldsymbol{w}$收敛路径（全数据梯度下降法）

3.1.4 批量随机梯度下降法

我们已经介绍了两种梯度下降法：随机梯度下降法和全数据梯度下降法。它们的优缺点如下。

- 随机梯度下降法每次只根据一个观测点计算梯度，并更新参数，每次迭代都很快且占用内存少，但是梯度稳定性差，需要迭代很多次才能收敛。
- 全数据梯度下降法每次计算梯度用到所有观测点，因此需要把整个数据集读入内存中，要求计算机有较大内存，且每次计算梯度都比较耗时。但是，全数据梯度下降法得到的梯度稳定性好，相对随机梯度下降法，只需迭代更少次数。

随机梯度下降法和全数据梯度下降法的优缺点总结如表 3-2 所示。

表 3-2 随机梯度下降法和全数据梯度下降法的优缺点

方法	每次迭代速度	需迭代次数	内存占用量	梯度稳定性
随机梯度下降法	快	多	少	不稳定
全数据梯度下降法	慢	少	多	稳定

相对来说，随机梯度下降法更适合数据集大，模型参数多的情形；全数据梯度下降法更适合数据集较小，模型参数少的情形。

本小节将介绍批量随机梯度下降法（Batch Stochastic Gradient Descent）。批量随机梯度下降法是随机梯度下降法和全数据梯度下降法的折中，每次计算梯度时既不是使用单个观测点也不是使用所有观测点。批量随机梯度下降法每次用一小部分观测点计算梯度。在实际中，常用 16、32、64、128、256、512 或者 1024 个观测点，根据计算机内存和模型规模等选取合适的观测点个数。用一小部分观测点计算梯度既可以保证快速地进行每次迭代，又可以通过平均少量观测点的梯度获得比较稳定的梯度，同时内存占用量不会显著增加。与随机梯度下降法相比，迭代次数可以显著减少。

在批量随机梯度下降法中，损失函数$L(\boldsymbol{w})$由一小部分观测点得到。记用于计算损失函数$L(\boldsymbol{w})$的观测点集合为$\mathcal{D}$。批量随机梯度下降法的步骤如下：

算法 3.4 批量随机梯度下降法

给定初始的参数值$\boldsymbol{w} = (w_1, w_2, \cdots, w_p)$，学习步长$\alpha$
迭代直至收敛：
　对每个数据集$\mathcal{D}$
　　计算该数据集的梯度，$\nabla_{\boldsymbol{w}} L(\boldsymbol{w})$
　　更新$\boldsymbol{w}$：$\boldsymbol{w} = \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} L(\boldsymbol{w})$

在广告数据中，以 TV 和 radio 为自变量，以 sales 为因变量建立没有截距项的线性回归模型。在批量随机梯度下降法中，记每次用于预测，计算损失函数$\mathrm{RSS}(\boldsymbol{w})$，以及梯度$\nabla_{\boldsymbol{w}}\mathrm{RSS}(\boldsymbol{w})$的观测点集合为$\mathcal{D}$。梯度$\nabla_{\boldsymbol{w}}\mathrm{RSS}(\boldsymbol{w})$可以表示为

$$\frac{\partial \mathrm{RSS}(\boldsymbol{w})}{\partial \boldsymbol{w}} = \frac{1}{|\mathcal{D}|}\sum_{i\in\mathcal{D}} (\boldsymbol{x}_i\boldsymbol{w} - y_i)\boldsymbol{x}_i^{\mathrm{T}}$$

在 Python 中，可以采用如下方式实现批量随机梯度下降法。在这里，每个批量数据的观测点数设为 20，而且按顺序获取每个批量数据。

```
"""
以 TV 和 radio 为自变量建立没有截距项的线性回归模型
使用批量随机梯度下降法迭代权重 w
"""
x = advertising.iloc[:,0:2].values
y = advertising.iloc[:,3].values
scaled_x = (x - np.mean(x, axis=0, keepdims=True))/  \
                np.std(x, axis=0, keepdims=True)
centered_y = y - np.mean(y)

n = len(scaled_x)
w = np.zeros(2)
lr = 0.1
batch_size = 20                              # 每批计算梯度的观测点个数
batch_num = int(np.ceil((n/batch_size))) # 批次数量

w_record = [w.copy()]
```

```
for iter in range(20):
    total_loss = 0
    for i in range(batch_num):
        batch_start = i * batch_size     # 每批次数据的开始位置
        batch_end = (i+1) * batch_size   # 每批次数据的结束位置
        pred = np.dot(scaled_x[batch_start:batch_end], w)
        loss = np.sum((pred - \
           centered_y[batch_start:batch_end])**2)/2/batch_size

        delta = pred - centered_y[batch_start:batch_end]

        w = w - lr * np.dot(scaled_x[batch_start:batch_end].T,\
                            delta)/batch_size
        w_record.append(w.copy())
        total_loss += loss
    if iter % 5 == 0:
        print("Error: "+str(total_loss/(i+1)))

Error: 6.879805204373558
Error: 1.4299567798383759
Error: 1.4297701878155769
Error: 1.4297693073089275
```

在这个例子中，观测点数量是 200。观测点数量较少，我们选择 batch size 为 20，这样刚好有 10 批数据。在图 3-10 中，星号表示真正的RSS($\boldsymbol{w}$)最小值（通过解方程的方式求得），圆点表示通过批量随机梯度下降法最终收敛得到的(w_1, w_2)的值。带有箭头的黄色实线表示(w_1, w_2)的迭代路线。可以看到，批量随机梯度下降法的梯度稳定性比随机梯度下降法提高很多。采用相同的学习步长，批量随机梯度下降法可以让(w_1, w_2)更为稳定地朝着RSS($\boldsymbol{w}$)的最小值点移动。当(w_1, w_2)接近星号时，也只是小幅波动，不会像随机梯度下降法一样波动很大。

```
"""
画出 RSS(w) 的等高线和 (w1,w2) 的迭代路线
这部分代码只是画出图 3-10，同学们只需要看懂图，不需要看代码的细节
"""
plt.contour(A, B, C, colors='black')
for i in range(len(w_record)-1):
    plt.arrow(w_record[i][0], w_record[i][1],  \
              w_record[i+1][0]-w_record[i][0], \
              w_record[i+1][1]-w_record[i][1], \
              width=0.01, color="y", head_width=0.05, \
              alpha = 0.5, zorder=1)
plt.scatter(w_record[-1][0], w_record[-1][1], s=50, c="b",  \
            marker="o", zorder=2)
plt.scatter(w_equation[0], w_equation[1], s=50, c="r",  \
            marker="*", zorder=2)
plt.xlabel("w1", fontsize=16)
plt.ylabel("w2", fontsize=16)
plt.show()
```

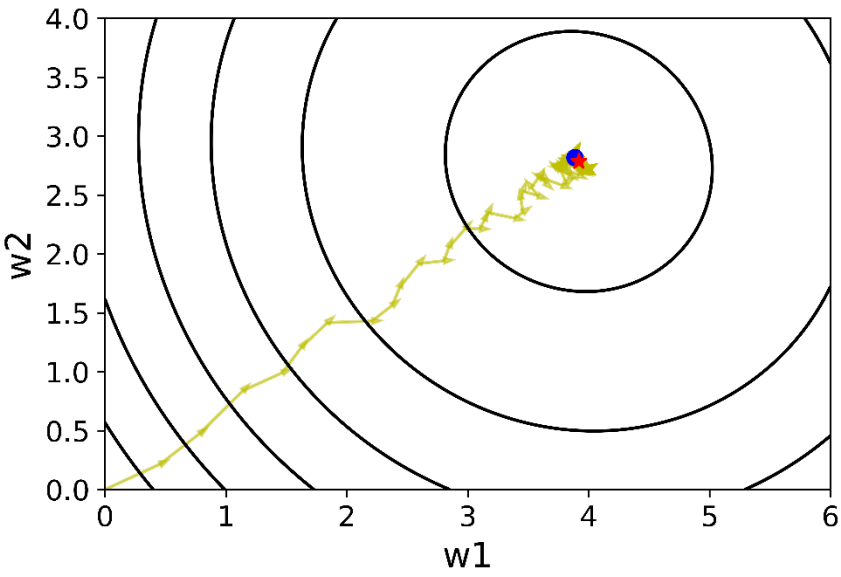

图 3-10 RSS($\boldsymbol{w}$)等高线及参数$\boldsymbol{w}$收敛路径（批量随机梯度下降法）

3.1.5 学习步长

在梯度下降法中，学习步长α是一个很重要的参数。学习步长α太小，算法会收敛得很慢；学习步长α太大，容易造成算法不收敛甚至发散。因此，在梯度下降法中，选择一个合适步长是非常重要的。在下面两个例子中，以 TV 为自变量，以 sales 为因变量建立线性回归模型，使用全数据梯度下降法迭代得到参数$\boldsymbol{w}$，分别设定不同的学习步长。第一种情况，设定学习步长为 0.01；第二种情况，设定学习步长为 2.1。可以看到，当学习步长为 0.01 时，迭代次数超过 150 次，预测误差仅降到 5.6 左右；作为对比，当学习步长为 0.1 时，迭代次数只要达到 20 次，预测误差就降到了 5.4 左右。

```
"""
全数据梯度下降法
步长等于 0.01，收敛较慢
"""
w, lr = 0, 0.01
input, target = scaled_TV, centered_sales
w_record = []
loss_record = []
for iter in range(200):
    pred = input * w
    loss = np.sum((pred - target)**2) / 2 / n

    w_record.append(w)
    loss_record.append(loss)

    delta = pred - target
    w -= lr * np.sum(input * delta) / n
```

```
    if (iter % 30==0 or iter == 199):
        print("iter: %3d; Loss: %0.3f"%(iter, loss))
```

```
iter: 0;      Loss: 13.543
iter: 30;     Loss: 9.790
iter: 60;     Loss: 7.737
iter: 90;     Loss: 6.614
iter: 120;    Loss: 5.999
iter: 150;    Loss: 5.663
iter: 180;    Loss: 5.479
iter: 199;    Loss: 5.408
```

当学习步长为 2.1 时，可以看到w不仅没有收敛到RSS(w)的最小值点，而且发散。从图 3-11 可以看到，w发散速度越来越快。

```
"""
全数据梯度下降法
步长等于 2.1，预测误差发散
"""
w, lr = 2, 2.1
input, target = scaled_TV, centered_sales
w_record = []
loss_record = []
for iter in range(200):
    pred = input * w
    loss = np.sum((pred - target)**2) / 2 / n

    w_record.append(w)
    loss_record.append(loss)

    delta = pred - target
    w -= lr * np.sum(input * delta) / n

    if (iter % 30==0 or iter == 199):
        print("iter: %3d; Loss: %0.3f"%(iter, loss))
```

```
iter: 0;    Loss: 7.401
iter: 30;   Loss: 658.227
iter: 60;   Loss: 198822.930
iter: 90;   Loss: 60536336.540
iter: 120;  Loss: 18432201406.354
iter: 150;  Loss: 5612266902969.234
iter: 180;  Loss: 1708832228158519.000
iter: 199;  Loss: 63917747551538624.000
```

```
"""
权重 w 与残差平方和 RSS(w) 随着迭代变化曲线
"""
plt.plot(w_vec, rss_vec)
w = 2      # 初始权重设为 2
# 初始权重的残差平方和
rss_0 = np.sum((centered_sales - scaled_TV * w)**2)/2/n
plt.scatter(2, rss_0, s=100,c="y", marker="o")
```

```
for i in range(len(w_record)-1):
    plt.arrow(w_record[i], loss_record[i], \
              w_record[i+1]-w_record[i], \
              loss_record[i+1]-loss_record[i],
              width=0.1, color="y", head_width=0.4, \
              length_includes_head=True)
plt.scatter(w_min, rss_min, s=100, c="r", marker="*")
plt.xlabel("w", fontsize=16)
plt.ylabel("RSS", fontsize=16)
plt.show()
```

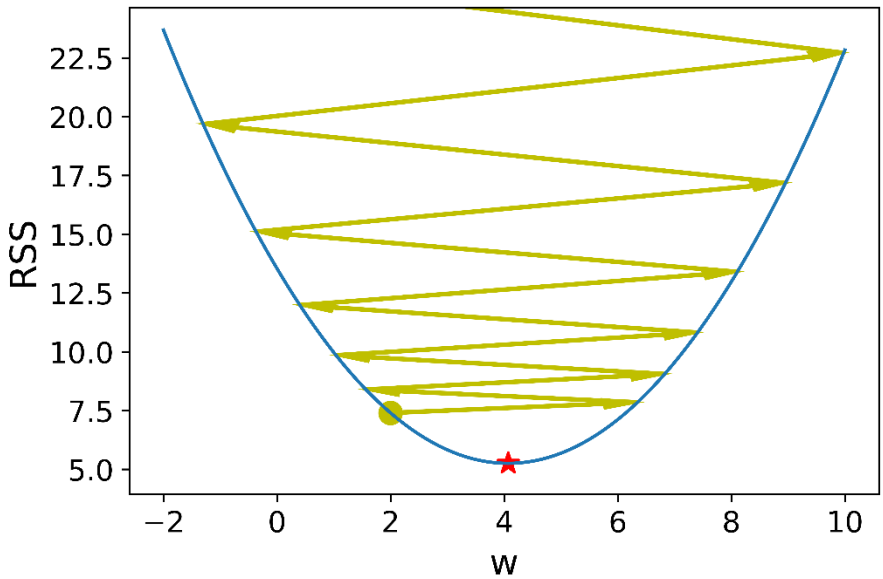

图 3-11 RSS(w)的收敛路径（学习步长为 2.1）

3.1.6 标准化和中心化

自变量的标准化和因变量的中心化是建立深度学习模型过程中常用的数据预处理方法。那么，为什么要这么做呢？从图 3-12 可以看到，自变量标准化和因变量中心化对梯度下降算法的影响。当自变量没有标准化时，因为不同自变量的方差不一样，RSS($\boldsymbol{w}$)等高线的椭圆形有可能更扁平。在这种情况下，$\boldsymbol{w}$有可能需要更长的路径才能到达最小值点。而且需要选择更加合适的步长，略大的步长便有可能造成算法不收敛。该缺点在参数$\boldsymbol{w}$维度大时，会更加明显。另一方面，自变量标准化之后，RSS($\boldsymbol{w}$)的等高线呈圆形。这种情况下，$\boldsymbol{w}$对初始值更不敏感，且更容易收敛到最小值点。自变量的标准化和因变量的中心化可以让梯度下降法的数值表现更加稳定，也更容易找到合适的初始值和步长。

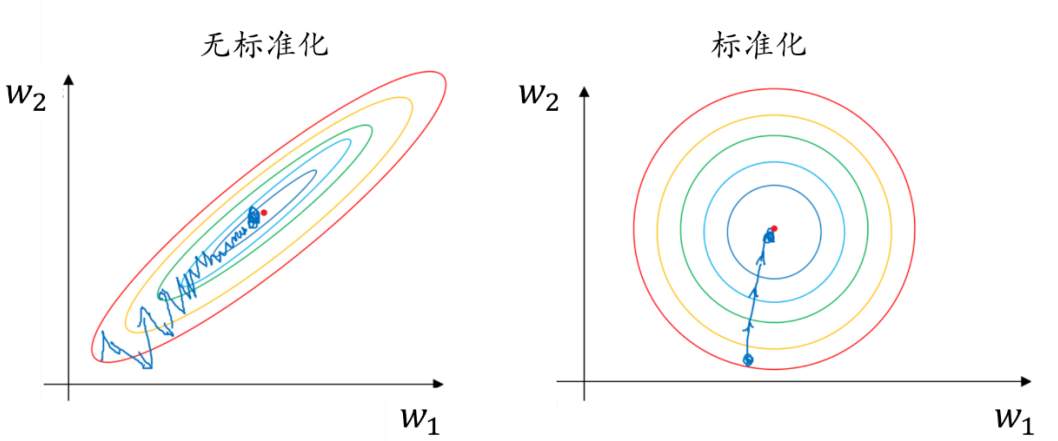

图 3-12 数据标准化

3.1.7 小结

对于三种优化方法，随机梯度下降法、全数据梯度下降法和批量随机梯度下降法，我们都需要对数据的自变量标准化，对因变量中心化，选取合适的初始值和学习步长。

- 数据标准化可以让梯度下降法的数值表现更加稳定，也更容易找到合适的初始值和步长。
- 在本节例子中，因为目标函数RSS($\boldsymbol{w}$)都是$\boldsymbol{w}$的的二次函数，函数有唯一最小值点，因此初始值的选取对本节的例子来说影响不太大。然而，在深度学习模型中，模型较为复杂，参数个数可能达到百万级别，选取好的初始值将变得非常重要。
- 学习步长对梯度下降法的数值结果有很大影响。学习步长α太小，算法会收敛得很慢；学习步长α太大，容易造成算法不收敛甚至发散。

对于批量随机梯度下降法，还有一个额外的参数需要给定：每批量数据的观测点个数（batch size）。

- batch size 太小，则每次迭代计算速度快，占用内存少，但是梯度稳定性差，需要更多的迭代次数；
- batch size 太大，则梯度稳定性好，需要更少的迭代次数，但是每次迭代计算速度慢，占用内存多。

总的来说，选择合适的 batch size，批量随机梯度下降法可以以较快的速度计算梯度且梯度稳定性较好，同时批量随机梯度下降法需要的内存空间和迭代次数都会较少。在梯度下降法中，学习步长，batch size 需要事先给定，而不像参数$\boldsymbol{w}$一样通过最小化损失函数得到，这类参数在机器学习中通常称为超参数。

表 3-3 三种梯度下降法的优缺点

方法	每次迭代速度	需迭代次数	内存占用量	梯度稳定性	超参数
随机梯度下降法	快	多	少	不稳定	初始值，步长
全数据梯度下降法	慢	少	多	稳定	初始值，步长
批量随机梯度下降法	较快	较少	较少	较稳定	初始值, 步长 batch size

3.2 logistic 模型

3.2.1 logistic 模型简介

本节将介绍一个线性分类模型，logistic 模型。我们先学习数据的两种类型：定量数据和定性数据，如表 3-4 所示。定量数据可以在某个区间内取任意值。例如，销售额，理论上可以是$(0,\infty)$中的任意数字。定性数据只能取若干个不同的值。例如，性别，男性和女性；眼睛的颜色，黑色、棕色和绿色；天气，下雨和天晴。

表 3-4 定量数据和定性数据

数据类型	特点	例子
定量数据	可以在某个区间内取任意值	身高、销售额、财富等
定性数据	只能取若干个不同的值	性别、颜色、天气、学历等

回归模型和分类模型要求输入的自变量向量是一样的，都是一个已知或者相对容易测量的数据向量（数据类型可以是定量数据，也可以是定性数据；当然，如果有的自变量是定性数据，需要转化成傀儡变量）。回归模型和分类模型的主要区别是因变量y的数据类型不一样。

- 在回归模型中，要求因变量y是一个定量变量，也就是说因变量可以在某个区间内取任意值。
- 在分类模型中，要求因变量y是一个定性变量，也就是说因变量只能取若干个不同的值。

处理实际问题时，我们首先判断数据的因变量是定量数据还是定性数据，然后再选择建立回归模型还是分类模型，如表 3-5。

表 3-5 回归模型和分类模型

模型	自变量的数据类型	因变量的数据类型
回归模型	定性数据或定量数据	定量数据

分类模型	定性数据或定量数据	定性数据

本节使用 Default 数据学习建立 logistic 模型的具体过程。该数据有两个自变量，分别为 Balance 和 Income，因变量为 Default。在这个例子中，我们感兴趣的是基于信用卡用户的每月信用卡余额（Balance）和每月收入（Income）判断该用户是否会信用卡违约（Default）。下面代码定义函数 `loadDataSet()`，该函数可以读入数据 Default.txt，并对数据进行简单处理，然后输出自变量矩阵 `x` 和因变量 `y`。具体来说，我们使用函数 `open()`打开放在 data 文件夹的文件 Default.txt，用函数 `readlines()`读取文件的所有行，然后用 `for` 循环，逐行对读取的文件进行处理。主要处理步骤是：先用函数 `strip()`去除每一行结尾的换行符，然后用函数 `split()`把每一行根据分隔符`\t` 分割数据。最后，把前两个数字（Balance 和 Income）转换成浮点型保存在列表 `x` 中；第三个数字（Default）保存在列表 `y` 中。

```
"""
定义函数 loadDataSet()
该函数可以载入数据 Default.txt
输出自变量矩阵 x，因变量 y
"""
def loadDataSet():
    x = []; y = []
    # 打开 data 文件中的文件 Default.txt
    f = open("./data/Default.txt")
    # 函数 readlines()读入文件 f 的所有行
    for line in f.readlines():
        lineArr = line.strip().split()
        x.append([float(lineArr[0]), float(lineArr[1])])
        y.append(lineArr[2])
    return np.array(x), y
x, y = loadDataSet()
print(y)
```

```
['Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes',
'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes',
'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes',
'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes',
'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'Yes',
'Yes', 'Yes', 'Yes', 'Yes', 'Yes', 'No', 'No', 'No', 'No', 'No',
'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No',
'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No',
'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No',
'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No', 'No',
'No', 'No', 'No', 'No', 'No']
```

可以看到，`y` 只能取两个值：Yes 和 No，因此，因变量 y 是定性变量，需要建立 logistic 模型。为了接下来方便建立模型，下面代码把 Yes 记为 1，No 记为 0。

```
y01 = np.zeros(len(y))
```

```
for i in range(len(y)):
    if y[i] == "Yes":
        y01[i] = 1
y = y01
y
```

```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0.])
```

图 3-13 根据 y 的不同，在散点图中绘制了颜色不同且大小不等的点。小圆点表示不违约（Default=“No”）；大圆点表示违约（Default=“Yes”）。从图 3-13 可以看到，Income 对是否信用卡违约影响不是很大，Balance 对是否信用卡违约影响很大。

```
plt.scatter(x[y==0,0], x[y==0,1], label="No")
plt.scatter(x[y==1,0], x[y==1,1], s = 80, label="Yes")
plt.xlabel("Balance", fontsize=16)
plt.ylabel("Income", fontsize=16)
plt.legend()
plt.show()
```

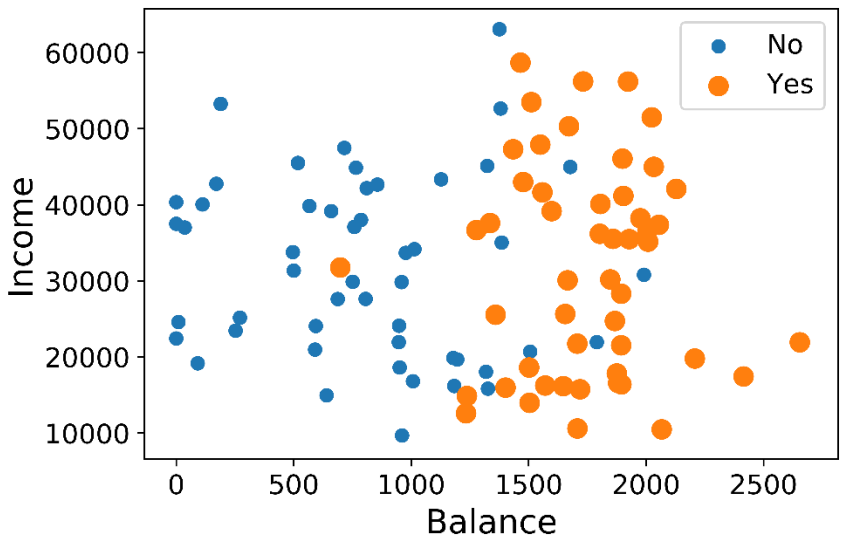

图 3-13 Default 数据散点图

在信用卡违约的例子中，因变量只有两个不同的值，Yes 和 No。我们通常不会通过建立模型来直接预测某用户的信用卡是否会违约，而是通过建立模型来预测某用户的信用卡违约的概率。当某用户的 Balance 和 Income 已知时，该用户信用卡违约的概率可以写成P(Default|Balance, Income)，接着可以通过条件概率

$\mathrm{P}(\mathrm{Default}|\mathrm{Balance},\mathrm{Income})$来判断信用卡用户是否违约。例如，如果$\mathrm{P}(\mathrm{Default}|\mathrm{Balance},\mathrm{Income}) \geq 0.5$，表明信用卡用户违约的风险大于50%，可以认为该用户有较大的违约可能性，因此可以预测该用户的 default 为 Yes；如果$\mathrm{P}(\mathrm{Default}|\mathrm{Balance},\mathrm{Income}) < 0.5$，表明信用卡用户违约的风险小于50%，可以认为该用户有较小的违约可能性，因此可以预测该用户的 Default 为 No。

对于一般的问题，记因变量Y的取值为 0 或者 1（在本章中，我们只考虑因变量分成两类的情况），那么$Y = 1$的概率可以写成，

$$\mathrm{P}(Y = 1|x_1, x_2, \cdots, x_p)$$

这里，$x_1, x_2, \cdots, x_p$为p个自变量。因变量$Y = 0$的条件概率为$\mathrm{P}(Y = 0|x_1, x_2, \cdots, x_p) = 1 - \mathrm{P}(Y = 1|x_1, x_2, \cdots, x_p)$。因此，我们只需要得到$\mathrm{P}(Y = 1|x_1, x_2, \cdots, x_p)$便可以很容易计算出$\mathrm{P}(Y = 0|x_1, x_2, \cdots, x_p)$。通过条件概率$\mathrm{P}(Y = 1|x_1, x_2, \cdots, x_p)$，可以得到$Y$的预测值为 0 还是 1。通常，

- 如果$\mathrm{P}(Y = 1|x_1, x_2, \cdots, x_p) \geq 0.5$，则预测$Y$为 1；
- 如果$\mathrm{P}(Y = 1|x_1, x_2, \cdots, x_p) < 0.5$，则预测$Y$为 0。

在 logistic 模型中，我们希望可以用类似线性回归模型的方式，通过计算自变量的加权和得到$\mathrm{P}(Y = 1|x_1, x_2, \cdots, x_p)$。最直接的方式是让

$$\mathrm{P}(Y = 1|x_1, x_2, \cdots, x_p) = b + w_1 x_1 + w_2 x_2 + \cdots + w_p x_p$$

但是，根据概率的定义，$\mathrm{P}(Y = 1|x_1, x_2, \cdots, x_p)$必须大于等于 0，小于等于 1 的；而上式等号右边的加权和并不能保证计算结果在 0 和 1 之间。基于该原因，在 logistic 模型中，不直接让$\mathrm{P}(Y = 1|x_1, x_2, \cdots, x_p)$等于加权和，而是对加权和进行变换，

$$\mathrm{P}(Y = 1|x_1, x_2, \cdots, x_p) = \frac{1}{1 + \mathrm{e}^{-(b + w_1 x_1 + w_2 x_2 + \cdots + w_p x_p)}}$$

函数$f(x) = \frac{1}{1+\mathrm{e}^{-x}}$称为 logistic 函数或者 sigmoid 函数（通常，在统计学文献中，该函数称为 logistic 函数；而在机器学习文献中，该函数称为 sigmoid 函数）。该函数的定义域为$(-\infty, \infty)$，值域为$(0,1)$。函数$f(x) = \frac{1}{1+\mathrm{e}^{-x}}$是连续的单调递增的函数，如图 3-14 所示。因此，容易看出，$1/(1 + \mathrm{e}^{-(b + w_1 x_1 + w_2 x_2 + \cdots + w_p x_p)})$的值在 0 和 1 之间。

```
a = np.linspace(-10,10,100)
plt.plot(a, 1/(1+np.exp(-a)), linewidth=3)
plt.axvline(x=0, linestyle="--", color="m", linewidth=2)
plt.axhline(y=0.5, linestyle="--", color="m", linewidth=2)
plt.show()
```

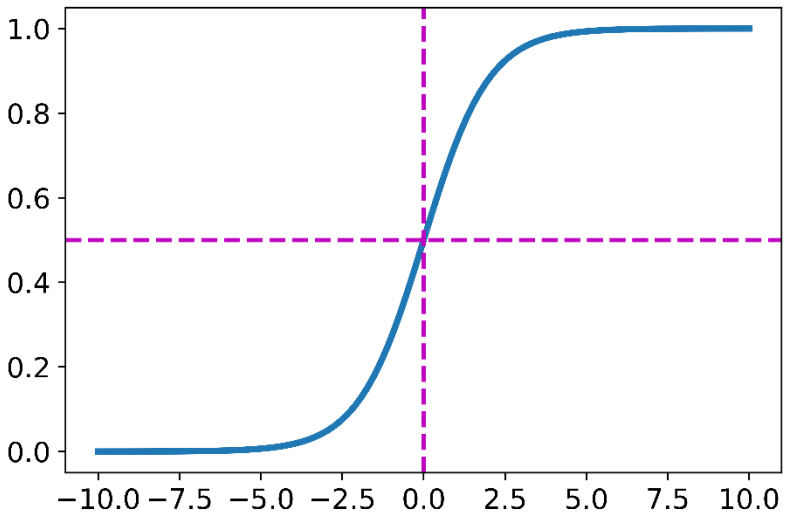

图 3-14 logistic 函数

从图 3-14 还可以看到，当$x = 0$时，$f(x) = 0.5$。因此，在 logistic 模型中，

- 预测$Y = 1$，即$\mathrm{P}(Y = 1|x_1, x_2, \cdots, x_p) \geq 0.5$等同于$b + w_1x_1 + w_2x_2 + \cdots + w_px_p \geq 0$;
- 预测$Y = 0$，即$\mathrm{P}(Y = 1|x_1, x_2, \cdots, x_p) < 0.5$等同于$b + w_1x_1 + w_2x_2 + \cdots + w_px_p < 0$。

3.2.2 估计b和$w_1, w_2, \cdots, w_p$

在 logisitc 模型中，损失函数可以定义为，

$$L(b, \boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n}\{-y_i\log(p_i) - (1 - y_i)\log(1 - p_i)\}$$

其中，$p_i = 1/(1 + \mathrm{e}^{-(b+w_1x_{i1}+w_2x_{i2}+\cdots+w_px_{ip})})$表示第$i$个观测点的预测概率值，$y_i$表示第$i$个观测点因变量的值（0 或者 1），$\boldsymbol{w} = (w_1, w_2, \cdots, w_p)^{\mathrm{T}}$。认真观察$L(b, \boldsymbol{w})$，可以看到，

- 当$y_i = 1$时，$(1 - y_i)\log(1 - p_i) = 0$，而且$-y_i\log(p_i) = -\log(p_i)$
- 当$y_i = 0$时，$-y_i\log(p_i) = 0$，而且$-(1 - y_i)\log(1 - p_i) = -\log(1 - p_i)$

因此，可以对做如下变换，

$$L(b, \mathbf{w}) = \frac{1}{n}\left\{\sum_{\{i:y_i=1\}}\{-\log(p_i)\} + \sum_{\{i:y_i=0\}}\{-\log(1 - p_i)\}\right\}$$

为什么 logistic 模型的损失函数可以这样定义呢？直观上，理想的b和$w_1, w_2, \cdots, w_p$估计值有如下特点，在训练数据中，

- $Y_i = 1$时，计算得到的预测概率p_i尽可能地接近于 1。
- $Y_i = 0$时，计算得到的预测概率p_i尽可能地接近于 0。

因为$0 < p_i < 1$，$-\log(p_i)$为正且随着p_i的增大而减小；$-\log(1-p_i)$为正且随着p_i的减小而减小。因此，最小化$L(b, \boldsymbol{w})$等同于，当$y_i = 1$时，让p_i最大化；当$y_i = 0$时，让p_i最小化。

现在我们已经知道通过最小化损失函数（或称为目标函数）$L(b, \boldsymbol{w})$可以得到b和$\boldsymbol{w}$的估计值，接下来的任务是计算$L(b, \mathbf{w})$的最小值。在 logistic 模型中，因为$p_i = 1/(1 + \mathrm{e}^{-(b+w_1 x_{i1}+w_2 x_{i2}+\cdots+w_p x_{ip})})$是$b$和$\boldsymbol{w}$的非线性函数，$L(b, \boldsymbol{w})$最小值不能通过求解析解的方式得到。我们将使用梯度下降法求解$L(b, \boldsymbol{w})$的最小值。

在这里，我们以批量随机梯度下降法为例。无论是线性回归模型，还是 logistic 模型，批量梯度下降法的主要步骤都是一样的。只是损失函数的形式不同以及梯度不同。目标函数为$L(b, \boldsymbol{w})$，该目标函数的梯度为$\nabla_b L(b, \boldsymbol{w})$和$\nabla_{\boldsymbol{w}} L(b, \boldsymbol{w})$。记每次用于计算$\nabla_b L(b, \boldsymbol{w})$和$\nabla_{\boldsymbol{w}} L(b, \boldsymbol{w})$的观测点集合为$\mathcal{D}$。批量随机梯度下降法的步骤如算法 3.5 所示。

算法 3.5 批量随机梯度下降法（包含截距项）

给定初始的参数值$b, \boldsymbol{w} = (w_1, w_2, \ldots, w_p)$，学习步长$\alpha$
迭代直至收敛：
　对每个数据集$\mathcal{D}$
　　计算梯度, $\nabla_b L(b, \boldsymbol{w})$ 和$\nabla_{\boldsymbol{w}} L(b, \boldsymbol{w})$
　　更新$\boldsymbol{w}$：$\boldsymbol{w} = \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} L(b, \boldsymbol{w})$
　　更新b：$b = b - \alpha \nabla_b L(b, \boldsymbol{w})$

梯度$\nabla_b L(b, \boldsymbol{w})$和$\nabla_{\boldsymbol{w}} L(b, \boldsymbol{w})$分别是损失函数$L(b, \boldsymbol{w})$关于$b$和每个权重$w_1, w_2, \cdots, w_p$的偏导数。我们先计算$\frac{\partial L(b, \boldsymbol{w})}{\partial w_1}$。求偏导数的主要技巧是链式法则。

$$\begin{aligned}
\frac{\partial L(b,\boldsymbol{w})}{\partial w_1} &= \frac{1}{n}\frac{\partial \sum_{i=1}^{n}\{-y_i\log(p_i)-(1-y_i)\log(1-p_i)\}}{\partial w_1} \\
&= \frac{1}{n}\sum_{i=1}^{n}\{-y_i\frac{1}{p_i}\frac{\partial p_i}{\partial w_1}-(1-y_i)\frac{-1}{1-p_i}\frac{\partial p_i}{\partial w_1}\} \quad (\text{对}p_i\text{求导数}) \\
&= \frac{1}{n}\sum_{i=1}^{n}\{(-y_i\frac{1}{p_i}-(1-y_i)\frac{-1}{1-p_i})\times \\
&\quad \frac{\mathrm{e}^{-(b+w_1x_{i1}+\cdots+w_px_{ip})}}{(1+\mathrm{e}^{-(b+w_1x_{i1}+\cdots+w_px_{ip})})^2}x_{i1}\} \qquad (\text{代入}\frac{\partial p_i}{\partial w_1}) \\
&= \frac{1}{n}\sum_{i=1}^{n}\{(p_i-y_i)x_{i1}\} \qquad (\text{化简})
\end{aligned}$$

用同样的方法，可以计算得到

- $\frac{\partial L(b,\boldsymbol{w})}{\partial b}=\frac{1}{n}\sum_{i=1}^{n}\{p_i-y_i\}$
- $\frac{\partial L(b,\boldsymbol{w})}{\partial w_k}=\frac{1}{n}\sum_{i=1}^{n}\{(p_i-y_i)x_{ik}\}, \quad k=1,\cdots,p$

因此，总的来说，$\nabla_{\boldsymbol{w}}L(b,\boldsymbol{w})=\frac{1}{n}\boldsymbol{X}^{\mathsf{T}}(\boldsymbol{p}-\boldsymbol{y})$，其中$\boldsymbol{p}=(p_1,\cdots,p_n)^{\mathsf{T}}$。

在 Python 中，可以使用如下代码实现 logistic 模型的参数估计。在这里，我们对自变量进行了标准化处理；但是，不需要对因变量进行标准化，因为因变量只能取两个值，0 和 1。

```
"""
建立 logistic 模型，使用批量随机梯度下降法迭代权重 w 和偏差 b
"""
def sigmoid(input):                                  # 定义函数 sigmoid
    return 1.0/(1+np.exp(-input))

scaled_x = (x - np.mean(x, axis=0, keepdims=True))/    \
            np.std(x, axis=0, keepdims=True)  #  自变量标准化

b = 0
w = np.zeros(2)
lr, batch_size = 0.05, 20                  # 给定学习步长，batch size
batch_num = int(np.ceil((len(scaled_x)/batch_size)))
b_record = [b]
w_record = w.copy()
for iter in range(100):
    total_loss = 0
    for i in range(batch_num):
        batch_start = i * batch_size
        batch_end = (i+1) * batch_size
```

```
        # 计算预测值
        pred = sigmoid(np.dot(scaled_x[batch_start:batch_end],\
                              w) + b)
        # 计算误差
        loss = - np.sum(y[batch_start:batch_end]*np.log(pred)+
                 (1-y[batch_start:batch_end]) * np.log(1-pred))

        delta = pred - y[batch_start:batch_end]
        # 计算 b 的梯度
        b_derivate = np.sum(delta)/batch_size
        # 计算 w 的梯度
        w_derivate = np.dot(scaled_x[batch_start:batch_end].T,\
                            delta)/batch_size

        b -= lr * b_derivate                    # 更新 b
        w -= lr * w_derivate                    # 更新 w

        b_record.append(b)
        w_record = np.vstack((w_record, w.copy()))
        total_loss += loss
    if (iter % 30==0 or iter==99):
        print("iter: %3d; Loss: %0.5f" % (iter, total_loss/
                                          len(scaled_x)))
```

```
iter: 0;  Loss: 0.68478
iter: 30; Loss: 0.37937
iter: 60; Loss: 0.33653
iter: 90; Loss: 0.32139
iter: 99; Loss: 0.31873
```

从代码运行结果看，随着b和$\boldsymbol{w}$的迭代，$L(b,\boldsymbol{w})$不断变小，最后稳定于 0.32 左右。截距项b和权重$\boldsymbol{w}$也分别随着迭代而慢慢收敛到-0.168和$2.407,-0.012$。图 3-15 画出了b和$\boldsymbol{w}$随着每个批量数据的迭代的变化曲线。

```
plt.plot(b_record, linewidth=2, label="b")
plt.plot(w_record[:,0], linewidth=2, label="w1")
plt.plot(w_record[:,1], linewidth=2, label="w2")
plt.legend()
plt.xlabel("# of batches", fontsize=16)
plt.show()
```

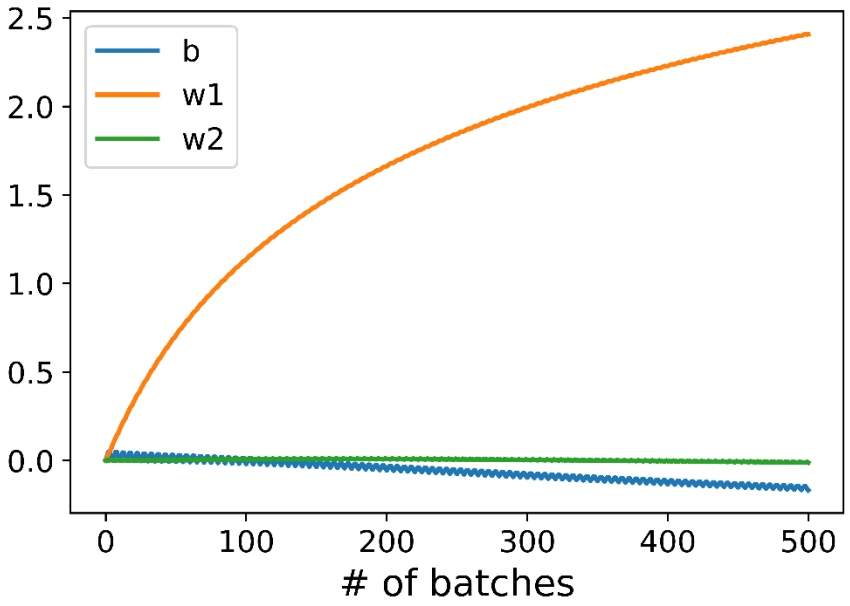

图 3-15 参数$b, \boldsymbol{w}$随着迭代的变化曲线

图 3-16 画出 logistic 模型的决策边界（decision boundary）。决策边界是$Y = 1$的概率和$Y = 0$的概率相等（即$\mathrm{P}(Y = 1|\boldsymbol{x}) = \mathrm{P}(Y = 0|\boldsymbol{x})$）的点连接的线。在下面代码中，我们先得到 1000 到 1500 的 100 个数，把这些数当成观测点中的 Balance，然后对这些数进行标准化，得到 `scaled_x1`。令$b + w_1 x_1 + w_2 x_2 = 0$，计算得到 `scaled_x2`，最后把 `scaled_x2` 变换回原数据 Income 的数值范围，画出决策边界。从图 3-16 可以看到，logistic 模型可以较好预测信用卡是否违约。

```
lp = np.linspace(1000, 1500, 1000)
means, stds = np.mean(x, axis=0), np.std(x, axis=0)
scaled_x1 = (lp - means[0]) / stds[0]
scaled_x2 = (-b - scaled_x1 * w[0])/w[1]

plt.scatter(x[y==0,0], x[y==0,1], label="No")
plt.scatter(x[y==1,0], x[y==1,1], s = 80, label="Yes")
plt.plot(lp, scaled_x2*stds[1]+means[1])
plt.ylim(8000, 60000)
plt.xlabel("Balance", fontsize=16)
plt.ylabel("Income", fontsize=16)
plt.legend()
plt.show()
```

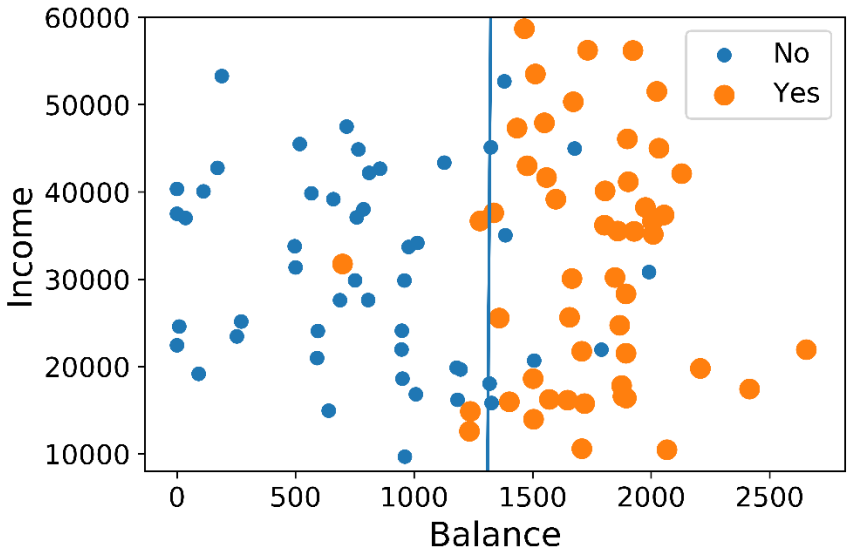

图 3-16 logistic 模型的决策边界

3.3 本章小结

线性模型的目的是通过自变量的加权和得到因变量的预测。在线性回归模型中，自变量的加权和即为因变量的预测；在 logistic 模型中，通过 sigmoid 函数变换的加权和为因变量预测概率。无论是线性回归模型还是 logistic 模型，建立模型的过程都可以分成以下 3 步：

- 给出模型的结构
- 根据建模的目的，构造损失函数（或称为目标函数）
- 最小化损失函数，得到参数估计值

在之后章节中，我们可以看到，对于更加复杂的深度学习模型，总的建模过程也是一样的。

本章还介绍了梯度下降法的 3 种变体：随机梯度下降法、全数据梯度下降法和批量随机梯度下降法。在实际应用中，3 种方法各有优缺点。通常情况下，选择合适的 batch size，批量随机梯度下降法可以以较快的速度计算梯度且梯度稳定性较好，同时批量随机梯度下降法需要的内存空间和迭代次数都较少。

习题

1. 思考：为什么线性回归模型中残差平方和乘$\frac{1}{2}$?
2. 在广告数据中，以 TV、radio、newspaper 为自变量，sales 为因变量建立线性回归模型，

$$\text{sales} = b + w_1 \times \text{TV} + w_2 \times \text{radio} + w_3 \times \text{newspaper}$$

- 自变量不要标准化，因变量不要中心化，使用批量随机梯度下降法计算模型的参数。
- 自变量标准化，因变量中心化，使用批量随机梯度下降法计算模型的参数。

观察两种情况下，批量梯度下降法的收敛情况，以及截距项和权重大小。

3. 在 Default 数据中，以 Balance 和 Income 为自变量，以 Default 为因变量建立线性回归模型。比较线性回归模型和 logistic 模型的结果，哪个模型表现得更好一些？

4. 在 Default 数据中，以 Balance 和 Income 为自变量，以 Default 为因变量建立 logistic 模型。使用随机梯度下降法和全数据梯度下降法计算模型参数。

5. 在题 4 的随机梯度下降法和全数据梯度下降法中，尝试不同的学习步长，观测梯度下降法的收敛情况。

6. 对于一般的 logistic 模型，$Y \sim \text{Bernouli}(p(\boldsymbol{x}))$，其中

$$p(\boldsymbol{x}) = \frac{1}{1 + \mathrm{e}^{-(b + w_1 x_1 + w_2 x_2 + \cdots + w_p x_p)}}$$

计算损失函数$L(b, \boldsymbol{w})$关于b和$\boldsymbol{w}$（$\boldsymbol{w} = (w_1, \cdots, w_p)^{\mathrm{T}}$）的偏导数，写出详细推导过程。