第1章

线 性 回 归

线性回归是最简单的机器学习模型，其形式简单，易于实现，同时也是很多机器学习模型的基础。第一个机器学习算法我们便从线性回归讲起。

1.1 线性回归模型

对于一个给定的训练集数据，线性回归的目标是找到一个与这些数据最为吻合的线性函数。举一个例子，中学物理我们学过的胡克定律指出：弹簧在发生弹性形变时，弹簧所受拉力F和弹簧的形变量 x 成正比，即$F = kx$。假设我们拿到一个新弹簧，测得了一组包含弹簧所受拉力F和形变量x的实验数据，如图 1-1 所示，根据实验数据估计弹簧倔强系数k的过程就是线性回归。

一般情况下，线性回归模型假设函数为：

$$h_{w,b}(x) = \sum_{i=1}^{n} w_i x_i + b = w^{\mathrm{T}} x + b$$

其中，$w \in \mathbf{R}^n$与 $b \in \mathbf{R}$ 为模型参数，也称为回归系数。为了方便，通常将 b 纳入权向量 w，作为 w_0，同时为输入向量 x 添加一个常数 1 作为 x_0：

$$w = (b, w_1, w_2, \ldots w_n)^{\mathrm{T}}$$
$$x = (1, x_1, x_2, \ldots x_n)^{\mathrm{T}}$$

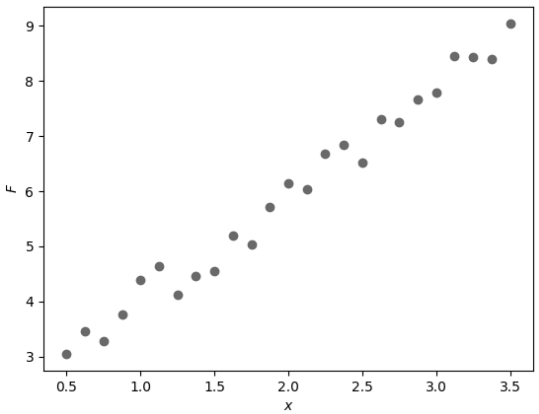

图 1-1

此时，假设函数为：

$$h_w(x) = \sum_{i=0}^{n} w_i x_i = w^{\mathrm{T}} x$$

其中，$w \in \mathbf{R}^{n+1}$，通过训练确定模型参数 w 后，便可使用模型对新的输入实例进行预测。

1.2 最小二乘法

线性回归模型通常使用均方误差（MSE）作为损失函数，假设训练集 $\boldsymbol{D}$ 有 m 个样本，均方误差损失函数定义为：

$$J(w) = \frac{1}{2m} \sum_{i=1}^{m} (h_w(x_i) - y_i)^2$$
$$= \frac{1}{2m} \sum_{i=1}^{m} (w^{\mathrm{T}} x - y_i)^2$$

均方误差的含义很容易理解，即所有实例预测值与实际值误差平方的均值，模型的训练目标是找到使得损失函数最小化的w。式中的常数 $\frac{1}{2}$ 并没有什么特殊的数学含义，仅是为了优化时求导方便。

损失函数 $J(w)$ 最小值点是其极值点，可先求 $J(w)$ 对 w 的梯度并令其为 0，再通过解方程求得。

计算 $J(w)$ 的梯度：

$$\begin{aligned}\nabla J(w) &= \frac{1}{2m}\sum_{i=1}^{m}\frac{\partial}{\partial w}(w^{\mathrm{T}}x_i - y_i)^2 \\ &= \frac{1}{2m}\sum_{i=1}^{m}2(w^{\mathrm{T}}x_i - y_i)\frac{\partial}{\partial w}(w^{\mathrm{T}}x_i - y_i) \\ &= \frac{1}{m}\sum_{i=1}^{m}(w^{\mathrm{T}}x_i - y_i)x_i\end{aligned}$$

以上公式使用矩阵运算描述形式更为简洁，设：

$$X = \begin{bmatrix} 1, & x_{11}, & x_{12} & \cdots & x_{1n} \\ 1, & x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1, & x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix} = \begin{bmatrix} x_1^{\mathrm{T}} \\ x_2^{\mathrm{T}} \\ \vdots \\ x_m^{\mathrm{T}} \end{bmatrix}$$

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

$$w = \begin{bmatrix} b \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

那么，梯度计算公式可写为：

$$\nabla J(w) = \frac{1}{m}X^{\mathrm{T}}(Xw - y)$$

令梯度为 0，解得：

$$\hat{w} = (X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}y$$

式中，$\hat{w}$ 即为使得损失函数（均方误差）最小的 w。需要注意的是，式中对 $X^{\mathrm{T}}X$ 求了逆矩阵，这要求 $X^{\mathrm{T}}X$ 是满秩的。然而实际应用中，$X^{\mathrm{T}}X$不总是满秩的（例如特

征数大于样本数），此时可解出多个 $\hat{w}$，选择哪一个由学习算法的归纳偏好决定，常见做法是引入正则化项。

以上求解最优 w 的方法被称为普通最小二乘法（Ordinary Least Squares，OLS）。

1.3 梯度下降

1.3.1 梯度下降算法

有很多机器学习模型的最优化参数不能像普通最小二乘法那样通过“闭式”方程直接计算，此时需要求助于迭代优化方法。通俗地讲，迭代优化方法就是每次根据当前情况做出一点点微调，反复迭代调整，直到达到或接近最优时停止，应用最为广泛的迭代优化方法是梯度下降（Gradient Descent）。图 1-2 所示为梯度下降算法逐步调整参数，从而使损失函数最小化的过程示意图。

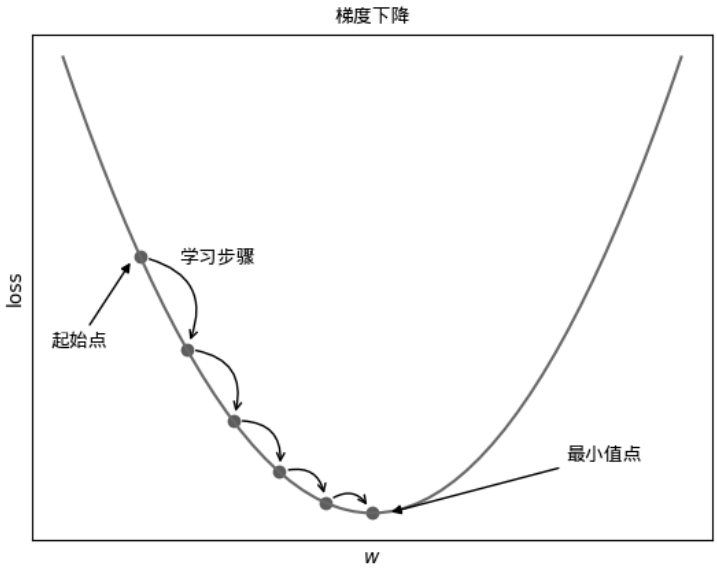

图 1-2

梯度下降算法常被形象地比喻为“下山”。如果你想尽快走下一座山，那么每迈一步的方向应选择当前山坡最陡峭的方向，迈一步调整一下方向，一直走到发现脚下已是平地。对于函数而言，梯度向量的反方向是其函数值下降最快的方向，即最陡峭的方向。梯度下降算法可描述为：

（1）根据当前参数 w计算损失函数梯度 $\nabla J(w)$。

（2）沿着梯度反方向 $-\nabla J(w)$ 调整 w，调整的大小称为步长，由学习率 η 控制。使用公式表述为：

$$w := w - \eta \nabla J(w)$$

（3）反复执行上述过程，直到梯度为 0 或损失函数降低小于阈值，此时称算法已收敛。

应用梯度下降算法时，超参数学习率 η 的选择十分重要。如果 η 过大，则有可能出现走到接近山谷的地方又一大步迈到了山另一边的山坡上，即越过了最小值点；如果 η 过小，下山的速度就会很慢，需要算法更多次的迭代才能收敛，这会导致训练时间过长。以上两种情形如图 1-3 所示。

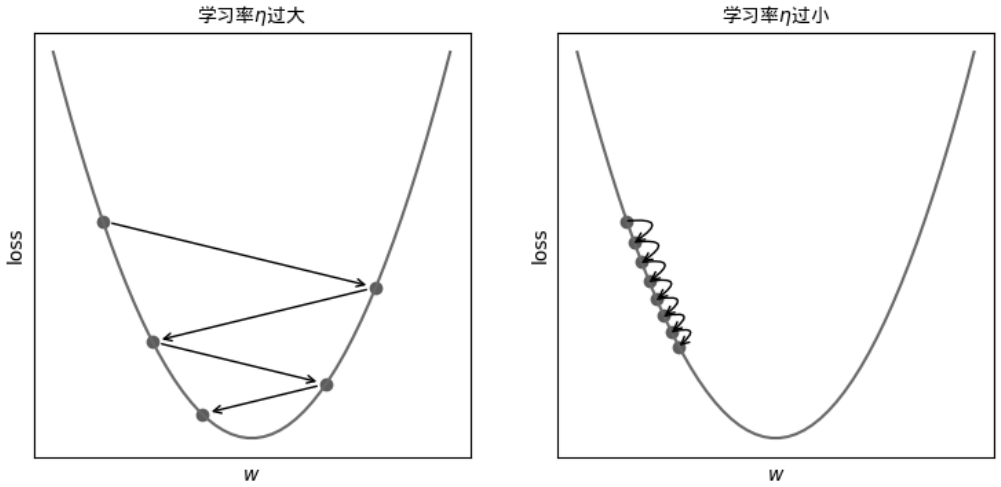

图 1-3

另外，还需知道的是，图 1-3 中的损失函数对于梯度下降算法是很理想的，它仅有一个全局最小值点，算法最终将收敛于该点。但也有很多机器学习模型的损失函数存在局部最小值，其曲线如绵延起伏的山脉，如图 1-4 所示。

对于图 1-4 中的损失函数，假设梯度下降算法的起始点位于局部最小值点左侧，算法则有可能收敛于局部最小值，而非全局最小值。此例子表明，梯度下降算法并不总收敛于全局最小值。

本节我们讨论的线性回归的损失函数是一个凸函数，不存在局部最小值，即只有一个全局最小值，因此梯度下降算法可收敛于全局最小值。

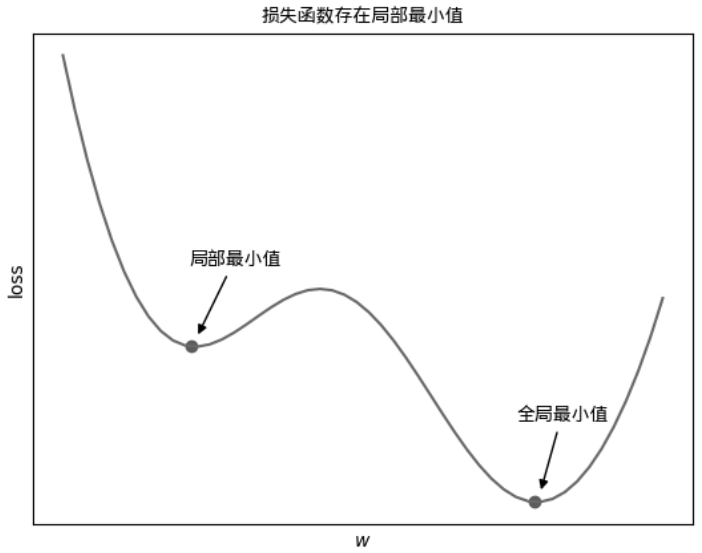

图 1-4

在 1.2 节中，我们计算出线性回归损失函数的梯度为：

$$\nabla J(w) = \frac{1}{m}\sum_{i=1}^{m}(w^{\mathrm{T}}x_i - y_i)\,x_i$$
$$= \frac{1}{m}X^{\mathrm{T}}(Xw - y)$$

设学习率为 η，梯度下降算法的参数更新公式为：

$$w := w - \eta\frac{1}{m}X^{\mathrm{T}}(Xw - y)$$

可以看出，执行梯度下降算法的每一步都是基于整个训练集 $\boldsymbol{X}$ 计算梯度的，因此梯度下降也被称为批量梯度下降：每次使用整批训练样本计算梯度，在训练集非常大时，批量梯度下降算法会运行得极慢。1.3.2 小节将介绍的随机梯度下降和小批量梯度下降可以解决该问题。

1.3.2 随机梯度下降和小批量梯度下降

随机梯度下降和小批量梯度下降可以看成是对批量梯度下降的近似，算法流程基本相同，只是每步使用少量的训练样本计算梯度。

随机梯度下降是与批量随机下降相反的极端情况，每一步只使用一个样本来计算梯度。

随机梯度下降算法的梯度计算公式为：

$$\nabla J(w) = (w^{\mathrm{T}} x_i - y_i)\, x_i$$

设学习率为 η，随机梯度下降算法的参数更新公式为：

$$w := w - \eta (w^{\mathrm{T}} x_i - y_i)\, x_i$$

因为每次只使用一个样本来计算梯度，所以随机梯度下降运行速度很快，并且内存开销很小，这使得随机梯度下降算法可以支持使用海量数据集进行训练。随机梯度下降过程中，损失函数的下降不像批量梯度下降那样缓缓降低，而是不断上下起伏，但总体上趋于降低，逐渐接近最小值。通常随机梯度下降收敛时，参数 w 是足够优的，但不是最优的。随机梯度下降算法的另一个优势是，当损失函数很不规则时（存在多个局部最小值），它更有可能跳过局部最小值，最终接近全局最小值。

随机梯度下降算法的一轮（Epoch）训练是指：迭代训练集中每一个样本，使用单个样本计算梯度并更新参数（一轮即 m 步），在每轮训练前通常要随机打乱训练集。

小批量梯度下降是介于批量梯度下降和随机梯度下降之间的折中方案，每一步既不使用整个训练集又不使用单个样本，而使用一小批样本计算梯度。

设一小批样本的数量为 N，小批量梯度下降算法的梯度计算公式为：

$$\nabla J(w) = \frac{1}{N} \sum_{i=k}^{k+N} (w^{\mathrm{T}} x_i - y_i)\, x_i$$

设学习率为 η，小批量梯度下降算法的参数更新公式为：

$$w := w - \eta \frac{1}{N} \sum_{i=k}^{k+N} (w^{\mathrm{T}} x_i - y_i)\, x_i$$

小批量梯度下降同时具备批量梯度下降和随机梯度下降二者的优缺点，应用时可视具体情况指定 N 值。

1.4　算法实现

1.4.1　最小二乘法

首先，我们基于最小二乘法实现线性回归，代码如下：

```
import numpy as np

class OLSLinearRegression:

    def _ols(self, X, y):
        '''最小二乘法估算 w'''
        tmp = np.linalg.inv(np.matmul(X.T, X))
        tmp = np.matmul(tmp, X.T)
        return np.matmul(tmp, y)

        # 若使用较新的 Python 和 Numpy 版本，可使用如下实现
        # return np.linalg.inv(X.T @ X) @ X.T @ y

    def _preprocess_data_X(self, X):
        '''数据预处理'''

        # 扩展 X，添加 x_0 列并设置为 1
        m, n = X.shape
        X_ = np.empty((m, n + 1))
        X_[:, 0] = 1
        X_[:, 1:] = X

        return X_

    def train(self, X_train, y_train):
        '''训练模型'''

        # 预处理 X_train(添加 x0=1)
        X_train = self._preprocess_data_X(X_train)

        # 使用最小二乘法估算 w
        self.w = self._ols(X_train, y_train)

    def predict(self, X):
        '''预测'''
        # 预处理 X_train(添加 x0=1)
        X = self._preprocess_data_X(X)
        return np.matmul(X, self.w)
```

此段代码十分简单，简要说明如下。

- _ols()方法：最小二乘法的实现，即 $\hat{w} = (X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}y$。
- _preprocess_data_X()方法：对 X 进行预处理，添加 x_0 列并设置为 1。
- train()方法：训练模型，调用_ols()方法估算模型参数 w，并保存。
- predict()方法：预测，实现函数 $h_w(x) = w^{\mathrm{T}}x$，对 X 中每个实例进行预测。

1.4.2　梯度下降

接下来，我们基于（批量）梯度下降实现线性回归，代码如下：

```
import numpy as np

class GDLinearRegression:

    def __init__(self, n_iter=200, eta=1e-3, tol=None):
        # 训练迭代次数
        self.n_iter = n_iter
        # 学习率
        self.eta = eta
        # 误差变化阈值
        self.tol = tol
        # 模型参数 w(训练时初始化)
        self.w = None

    def _loss(self, y, y_pred):
        '''计算损失'''
        return np.sum((y_pred - y) ** 2) / y.size

    def _gradient(self, X, y, y_pred):
        '''计算梯度'''
        return np.matmul(y_pred - y, X) / y.size

    def _gradient_descent(self, w, X, y):
        '''梯度下降算法'''

        # 若用户指定 tol，则启用早期停止法
        if self.tol is not None:
            loss_old = np.inf
```

```
29.
30.         # 使用梯度下降，至多迭代 n_iter 次，更新 w
31.         for step_i in range(self.n_iter):
32.             # 预测
33.             y_pred = self._predict(X, w)
34.             # 计算损失
35.             loss = self._loss(y, y_pred)
36.             print('%4i Loss: %s' % (step_i, loss))
37.
38.             # 早期停止法
39.             if self.tol is not None:
40.                 # 如果损失下降小于阈值，则终止迭代
41.                 if loss_old - loss < self.tol:
42.                     break
43.                 loss_old = loss
44.
45.             # 计算梯度
46.             grad = self._gradient(X, y, y_pred)
47.             # 更新参数 w
48.             w -= self.eta * grad
49.
50.     def _preprocess_data_X(self, X):
51.         '''数据预处理'''
52.
53.         # 扩展 X，添加 x0 列并设置为 1
54.         m, n = X.shape
55.         X_ = np.empty((m, n + 1))
56.         X_[:, 0] = 1
57.         X_[:, 1:] = X
58.
59.         return X_
60.
61.     def train(self, X_train, y_train):
62.         '''训练'''
63.
64.         # 预处理 X_train(添加 x0=1)
65.         X_train = self._preprocess_data_X(X_train)
66.
```

```
67.         # 初始化参数向量 w
68.         _, n = X_train.shape
69.         self.w = np.random.random(n) * 0.05
70.
71.         # 执行梯度下降训练 w
72.         self._gradient_descent(self.w, X_train, y_train)
73.
74.     def _predict(self, X, w):
75.         '''预测内部接口，实现函数 h(x).'''
76.         return np.matmul(X, w)
77.
78.     def predict(self, X):
79.         '''预测'''
80.         X = self._preprocess_data_X(X)
81.         return self._predict(X, self.w)
```

上述代码简要说明如下（详细内容参看代码注释）。

- __init__()方法：构造器（也称为构造函数），保存用户传入的超参数。
- _predict()方法：预测的内部接口，实现函数 $h_w(x) = w^{\mathrm{T}}x$。
- _loss()方法：实现损失函数 $J(w)$，计算当前 w 下的损失，该方法有以下两个用途。
  - 供早期停止法使用：如果用户通过超参数 tol 启用早期停止法，则调用该方法计算损失。
  - 方便调试：迭代过程中可以每次打印出当前损失，观察变化的情况。
- _gradient()方法：计算当前梯度 $\nabla J(w)$。
- _gradient_descent()方法：实现批量梯度下降算法。
- _preprocess_data_X()方法：对 X 进行预处理，添加 x_0 列并设置为 1。
- train()方法：训练模型。该方法由 3 部分构成：
  - 对训练集的 X_train 进行预处理，添加x_0 列并设置为 1。
  - 初始化模型参数 w，赋值较小的随机数。
  - 调用_gradient_descent()方法训练模型参数 w。
- predict()方法：预测。内部调用_predict()方法对 X 中每个实例进行预测。

1.5 项目实战

最后，我们来做一个线性回归的实战项目：分别使用 OLSLinearRegression 和 GDLinearRegression 预测红酒口感，如表 1-1 所示。

表 1-1 红酒口感数据集（https://archive.ics.uci.edu/ml/datasets/wine+quality）

列号	列名	含义	特征/目标	可取值
1	fixed acidity	非挥发性酸	特征	实数
2	volatile acidity	挥发性酸	特征	实数
3	citric acid	柠檬酸	特征	实数
4	residual sugar	残留糖分	特征	实数
5	Chlorides	氯化物	特征	实数
6	free sulfur dioxide	游离二氧化硫	特征	实数
7	total sulfur dioxide	总二氧化硫	特征	实数
8	Density	密度	特征	实数
9	pH	pH 值	特征	实数
10	Sulphates	硫酸盐	特征	实数
11	Alcohol	酒精含量	特征	实数
12	Quality	口感	目标	3~8 的整数

数据集中包含 1599 条数据，其中每一行包含红酒的 11 个化学特征以及专家评定的口感值。虽然口感值只是 3~8 的整数，但我们依然把该问题当作回归问题处理，而不是当作包含 6 种类别（3~8）的分类问题处理。如果当作分类问题，则预测出的类别间无法比较好坏，例如我们不清楚第 1 类口感是否比第 5 类口感好，但我们明确知道 5.3 比 4.8 口感好。

读者可使用任意方式将数据集文件 winequality-red.csv 下载到本地，此文件所在的 URL 为：https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-red.csv。

1.5.1 准备数据

调用 Numpy 的 genfromtxt 函数加载数据集：

```
1.  >>> import numpy as np
2.  >>> data = np.genfromtxt('winequality-red.csv', delimiter=';',
skip_header=True)
```

```
>>> X = data[:, :-1]
>>> X
array([[ 7.4  ,  0.7  ,  0.   , ...,  3.51 ,  0.56 ,  9.4  ],
       [ 7.8  ,  0.88 ,  0.   , ...,  3.2  ,  0.68 ,  9.8  ],
       [ 7.8  ,  0.76 ,  0.04 , ...,  3.26 ,  0.65 ,  9.8  ],
       ...,
       [ 6.3  ,  0.51 ,  0.13 , ...,  3.42 ,  0.75 , 11.     ],
       [ 5.9  ,  0.645,  0.12 , ...,  3.57 ,  0.71 , 10.   2  ],
       [ 6.   ,  0.31 ,  0.47 , ...,  3.39 ,  0.66 , 11.     ]])
>>> y = data[:, -1]
>>> y
array([5., 5., 5., ..., 6., 5., 6.])
```

1.5.2　模型训练与测试

我们要训练并测试两种不同方法实现的线性回归模型：OLSLinearRegression 和 GDLinearRegression。

1. OLSLinearRegression

先从更为简单的 OLSLinearRegression 开始。

首先创建模型：

```
>>> from linear_regression import OLSLinearRegression
>>> ols_lr = OLSLinearRegression()
```

创建 OLSLinearRegression 时无须传入任何参数。

然后，调用 sklearn 中的 train_test_split 函数将数据集切分为训练集和测试集（比例　为 7:3）：

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3)
```

接下来，训练模型：

```
>>> ols_lr.train(X_train, y_train)
```

因为训练集容量及实例特征数量都不大，所以很短时间内便可完成训练。

使用已训练好的模型对测试集中的实例进行预测：

```
>>> y_pred = ols_lr.predict(X_test)
>>> y_pred
array([5.97884966, 5.97298391, 5.24300126, 5.06622202, 5.34749778,
       5.75578547, 5.12227758, 5.42068169, 5.45180575, 6.13254774,
       5.34770062, 5.83609291, 6.05261885, 6.12793756, 6.02340132,
       ...
       5.57593107, 6.52179897, 5.96058307, 5.34186329, 5.72550139,
       5.22740437, 5.07311142, 5.78794577, 5.9205651 , 5.87093099,
       5.87183078, 5.45259226, 5.44723566, 6.0368874 , 5.36931666])
```

仍以均方误差（MSE）衡量回归模型的性能，调用 sklearn 中的 mean_squared_error 函数计算 MSE：

```
>>> mse = mean_squared_error(y_test, y_pred)
>>> mse
0.4211724526626152
```

模型在测试集上的 MSE 为 0.421，其平方根约为 0.649。还可以测试模型在训练集上的 MSE：

```
>>> y_train_pred = ols_lr.predict(X_train)
>>> mse_train = mean_squared_error(y_train, y_train_pred)
>>> mse_train
0.41723084614277367
```

模型在训练集与测试集的性能相差不大，表明未发生过度拟合现象。

过度拟合也称为过拟合，不过在中文上下文中使用“过拟合”容易产生歧义，故本书统一使用“过度拟合”。

另一个常用的衡量回归模型的指标是平均绝对误差（MAE），其定义如下：

$$\frac{1}{m}\sum_{i=1}^{m}|y_{i\_pred}-y_i|$$

MAE 的含义更加直观一些：所有实例预测值与实际值之误差绝对值的平均值。调用 sklearn 中的 mean_absolute_error 函数计算模型在测试集上的 MAE：

```
>>> from sklearn.metrics import mean_absolute_error
>>> mae = mean_absolute_error(y_test, y_pred)
>>> mae
0.4924678778849731
```

MAE 为 0.492，即预测口感值比实际口感值平均差了 0.492。

2. GDLinearRegression

再来训练并测试 GDLinearRegression，该过程比之前的 OLSLinearRegression 麻烦一些，因为它有 3 个超参数需要我们设置，而最优的超参数组合通常需要通过大量实验得到。

GDLinearRegression 的超参数有：

（1）梯度下降最大迭代次数 n_iter

（2）学习率 eta

（3）损失降低阈值 tol（tol 不为 None 时，开启早期停止法）

先以超参数（n_iter=3000，eta=0.001，tol=0.00001）创建模型：

```
>>> from linear_regression import GDLinearRegression
>>> gd_lr = GDLinearRegression(n_iter=3000, eta=0.001, tol=0.00001)
```

为了与之前的 OLSLinearRegression 进行对比，我们使用与之前相同的训练集和测试集（不重新切分 X,y）训练模型：

```
>>> gd_lr.train(X_train, y_train)
   0 Loss: 12.200517575720156
   1 Loss: 37.31474276210929
```

以上输出表明，经过一步梯度下降以后，损失 Loss 不降反升，然后算法便停止了，这说明步长太大，已经迈到对面山坡上了，需调小学习率。将学习率调整为 eta=0.0001 再次尝试：

```
>>> gd_lr = GDLinearRegression(n_iter=3000, eta=0.0001, tol=0.00001)
>>> gd_lr.train(X_train, y_train)
   0 Loss: 16.023304785489813
   1 Loss: 11.497026121171373
   2 Loss: 9.592720523188712
   3 Loss: 8.75383172330532
   4 Loss: 8.348468678827183
   5 Loss: 8.120102508707241
    ...
2994 Loss: 0.5387288272167047
2995 Loss: 0.5387143011814725
2996 Loss: 0.5386997815639667
```

```
13. 2997 Loss: 0.5386852683612686
14. 2998 Loss: 0.538670761570461
15. 2999 Loss: 0.5386562611886281
```

这次虽然损失随着迭代逐渐下降了，但是迭代到了最大次数 3000，算法依然没有收敛，最终损失（在训练集上的 MSE）为 0.539，距离之前用最小二乘法计算出的最小值 0.417 还差很远，并且发现后面每次迭代损失下降得非常小。这种状况主要是由于 $\boldsymbol{X}$ 中各特征尺寸相差较大造成的，观察 $\boldsymbol{X}$ 中各特征的均值：

```
1.  >>> X.mean(axis=0)
2.  array([ 8.31963727,  0.52782051,  0.27097561,  2.5388055 ,
0.08746654,
3.       15.87492183,  46.46779237,  0.99674668,  3.3111132 ,
0.65814884,
4.       10.42298311])
```

可看出各特征尺寸差距确实很大，有的特征间相差了好几个数量级。以两个特征为例，如果 $\boldsymbol{x_1}$ 特征尺寸比 $\boldsymbol{x_2}$ 的小很多（如图 1-5 所示），通常 $\boldsymbol{x_2}$ 的变化对损失函数值影响更大，梯度下降时就会先沿着接近 $\boldsymbol{x_2}$ 轴的方向下山，再沿着 $\boldsymbol{x_1}$ 轴进入一段长长的几乎平坦的山谷，用下山时谨慎的小步走平地，速度慢得像蜗牛爬，虽然最终也可以抵达最小值点，但需要更多的迭代次数，花费更长时间。

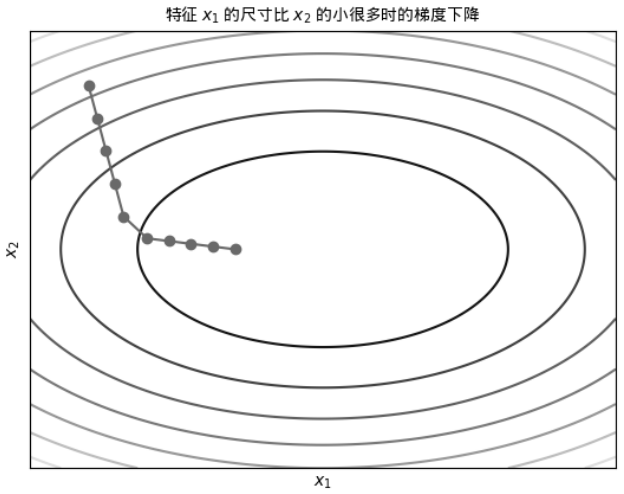

图 1-5

相反，如果 $\boldsymbol{x_1}$ 和 $\boldsymbol{x_2}$ 特征尺寸相同（见图 1-6），梯度下降时将更直接地走向最小值点，算法收敛更快。

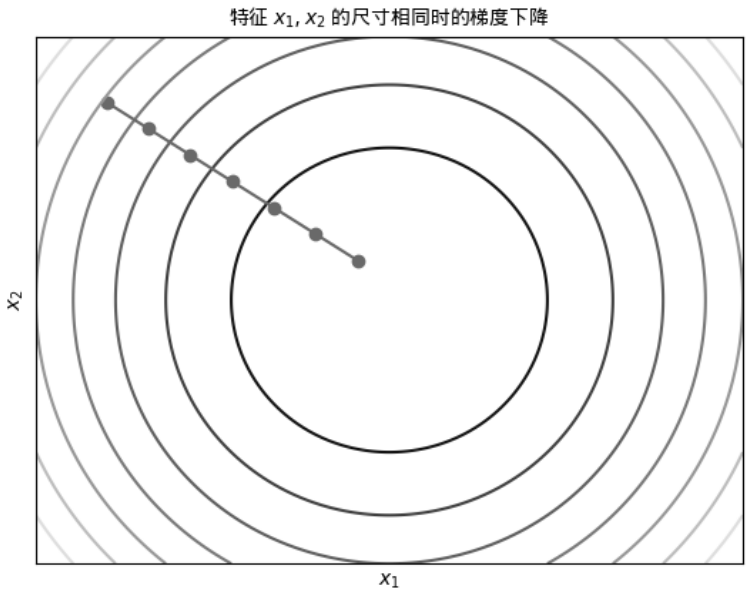

图 1-6

接下来我们把 $\boldsymbol{X}$ 各特征缩放到相同尺寸，然后重新训练模型。将特征缩放到相同尺寸有两种常用方法：归一化（Normalization）和标准化（Standardization）。

归一化是指使用 min-max 缩放将各特征的值缩放至 [0, 1] 区间。对于第 i 个实例的第 j 个特征 $x_i^{(j)}$，归一化转换公式为：

$$x_{i\_Norm}^{(j)} = \frac{x_i^{(j)} - x_{min}^{(j)}}{x_{max}^{(j)} - x_{min}^{(j)}}$$

其中，$x_{max}^{(j)}$ 和 $x_{min}^{(j)}$ 分别为所有实例第 j 个特征的最大值和最小值。调用 sklearn 中的 MinMaxScaler 函数可以完成归一化转换。

标准化是指将各特征的均值设置为 0，方差设置为 1。对于第 i 个实例的第 j 个特征 $x_i^{(j)}$，标准化转换公式为：

$$x_{i\_Std}^{(j)} = \frac{x_i^{(j)} - \mu_x^{(j)}}{\sigma_x^{(j)}}$$

其中，$\mu_x^{(j)}$ 和 $\sigma_x^{(j)}$ 分别为所有实例第 j 个特征的均值和标准差。调用 sklearn 中的 StandardScaler 函数可以完成标准化转换。

对于大多数机器学习算法而言，标准化更加实用，因为标准化保持了异常值所蕴含的有用信息。这里我们调用 sklearn 中的 StandardScaler 函数对各特征进行缩放：

```
>>> from sklearn.preprocessing import StandardScaler
>>> ss = StandardScaler()
>>> ss.fit(X_train)
StandardScaler(copy=True, with_mean=True, with_std=True)
>>> X_train_std = ss.transform(X_train)
>>> X_test_std = ss.transform(X_test)
>>> X_train_std[:3]
array([[ 2.06525234, -0.52177354,  2.00384591,  0.41731378,
         0.43302079, -0.54794448, -0.71530754,  1.69254258,
        -1.04504699,  1.06645239, -0.00577349],
       [-0.54836368,  0.42987536, -0.07611659, -0.31883207,
        -0.11252578,  0.11774751,  1.37454564, -0.33359707,
        -0.11983048, -0.58506021, -0.56891899],
       [-0.19988155, -0.01795942, -0.12811565, -0.39244666,
        -0.21743858, 0.30794522,  0.39108532, -0.3547027 ,
        -0.05374359, -1.04064989, -0.19348865]])
>>> X_test_std[:3]
array([[-1. 41956902,  0.14997863, -0.90810159, -0.24521749,
         -0.55315954,2.20992233,  0.14522024, -0.88234323,
         1.40016808,  0.553914,  0.74508718],
       [ 0.55516308,  -0.2418768 ,  0.0798806 , -0.31883207,
        0.81070687,  -0.9283399 , -0.93043948,  0.00409287,
         -0.58243874,  0.32611916,  0.74508718],
       [-0.43220297,  0.68178242, -1.27209503, -0.46606124,
        -0.0495781,  -0.16754906,  0.32961905,  0.05685692,
        0.40886467, -1.15454731, -0.09963107]])
```

这里需要注意，在以上的代码中，StandardScaler 只对训练集进行拟合（计算均值 μ 和标准差σ），然后使用相同的拟合参数对训练集和测试集进行转换，因为在预测时测试集对于我们是未知的。

现在各特征值被缩放到了相同的尺寸。接下来重新创建模型，并使用已缩放的数据进行训练：

```
>>> gd_lr = GDLinearRegression(n_iter=3000, eta=0.05, tol=0.00001)
>>> gd_lr.train(X_train_std, y_train)
    0 Loss: 32.   37093970975737
    1 Loss: 29.   238242349047354
    2 Loss: 26.   4144603303325
    3 Loss: 23.   868826231616428
```

```
    4 Loss: 21.   573695276485843
    5 Loss: 19.   50421695549397
    6 Loss: 17.   63804348681383
     ...
128 Loss: 0.4288433323855949
129 Loss: 0.428828969543995
130 Loss: 0.42881542387707694
131 Loss: 0.4288026302147262
132 Loss: 0.4287905293029377
133 Loss: 0.42877906724161524
134 Loss: 0.4287681949766629
135 Loss: 0.4287578678410973
136 Loss: 0.42874804514041764
```

这次我们将 eta 大幅提高到了 0.05，经过 136 次迭代后算法收敛，目前损失（在训练集上的 MSE）为 0.428，已接近用最小二乘法计算出的最小值 0.417。

最后使用已训练好的模型对测试集中的实例进行预测，并评估性能：

```
>>> y_pred = gd_lr.predict(X_test_std)
>>> mse = mean_squared_error(y_test, y_pred)
>>> mse
0.39311865138396274
>>> mae = mean_absolute_error(y_test, y_pred)
>>> mae
0.49190905290250364
```

此时 MSE 为 0.393，MAE 为 0.492，与之前使用 OLSLinearRegression 的性能差不多。读者可以继续调整超参数进行优化，但性能不会有太明显的提升，毕竟线性回归是非常简单的模型。

至此，线性回归的项目就完成了。

第2章

Logistic 回归与 Softmax 回归

Logistic 回归这个名字可能会引起误会，虽然名字中带有“回归”，但它是一个分类算法，用于处理二元分类问题。Softmax 回归同样是分类算法，它是在 Logistic 回归的基础上进行推广得到的，用于处理多元分类问题。

2.1 Logistic 回归

2.1.1 线性模型

Logistic 回归是一种广义线性模型，它使用线性判别式函数对实例进行分类。举一个例子，图 2-1 中有两种类别的实例，o 表示正例，x 表示负例。

我们可以找到一个超平面将两类实例分隔开，即正确分类，假设超平面方程为：

$$w^{\mathrm{T}}x+b=0$$

其中，$w\in\mathbf{R}^{\mathbf{n}}$ 为超平面的法向量，$b\in\mathbf{R}$ 为偏置。

超平面上方的点都满足：

$$w^{\mathrm{T}}x+b>0$$

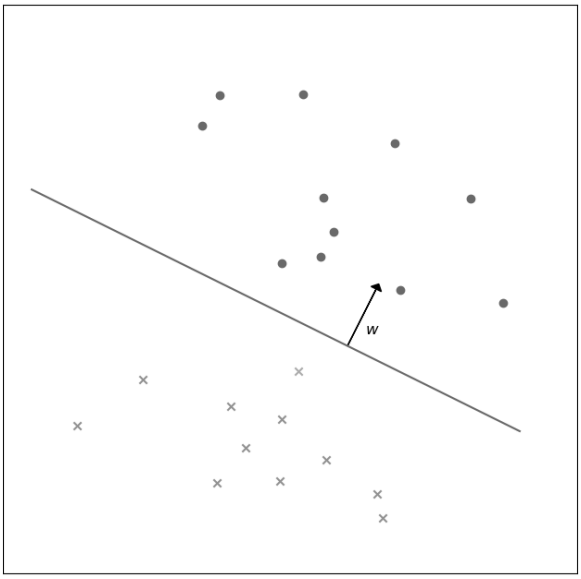

图 2-1

而超平面下方的点都满足：

$$w^{\mathrm{T}}x+b<0$$

这意味着，我们可以根据以下 x 的线性函数的值（与 0 的比较结果）判断实例的类别：

$$z=g(x)=w^{\mathrm{T}}x+b$$

分类函数以 z 为输入，输出预测的类别：

$$c=H(z)=H(g(x))$$

以上便是线性分类器的基本模型。

2.1.2　logistic 函数

显然，最理想的分类函数为单位阶跃函数：

$$H(z)=\begin{cases}1, & z\geqslant 0\\ 0, & z<0\end{cases}$$

但单位阶跃函数作为分类函数有一个严重缺点：它不连续，所以不是处处可微，这使得一些算法不能得以应用（如梯度下降）。我们希望找到一个在输入输出特性上

与单位阶跃函数类似，并且单调可微的函数来替代阶跃函数，logistic 函数便是一种常用替代函数。

logistic 回归函数定义为：

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

其函数图像如图 2-2 所示。

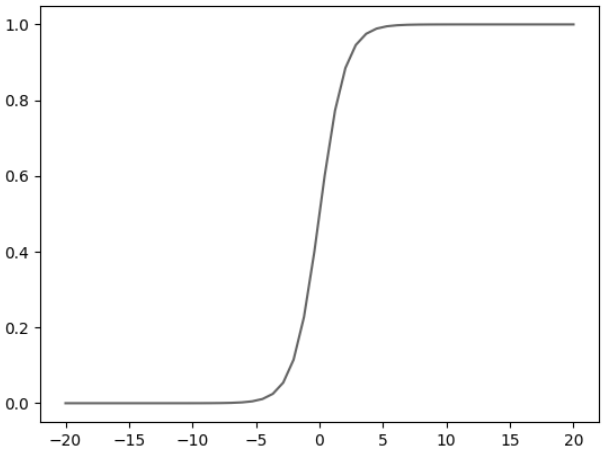

图 2-2

logistic 函数是一种 Sigmoid 函数（S 型）。从图 2-2 可以看出，logistic 函数的值域在(0, 1)之间连续，函数的输出可视为 x 条件下实例为正例的条件概率，即：

$$P(y=1|x) = \sigma(g(x)) = \frac{1}{1+e^{-(w^{\mathrm{T}}x+b)}}$$

那么，x 条件下实例为负例的条件概率为：

$$P(y=0|x) = 1-\sigma(g(x)) = \frac{1}{1+e^{(w^{\mathrm{T}}x+b)}}$$

以上概率的意义是什么呢？实际上，logistic 函数是对数概率函数的反函数。一个事件的概率（odds）指该事件发生的概率 p 与该事件不发生的概率 $1-p$ 的比值。那么，对数概率为：

$$\log\frac{p}{1-p}$$

对数概率大于 0 表明是正例的概率大，小于 0 表明是负例的概率大。

Logistic 回归模型假设一个实例为正例的对数概率是输入 x 的线性函数，即：

$$\log \frac{p}{1-p} = w^{\mathrm{T}}x + b$$

反求上式中的 p 便可得出：

$$p = \sigma(g(x)) = \frac{1}{1 + e^{-(w^{\mathrm{T}}x+b)}}$$

理解上述 logistic 函数概率的意义，是后面使用极大似然法的基础。

另外，logistic 函数还有一个很好的数学特性，$\sigma(z)$ 的一阶导数形式简单，并且是 $\sigma(z)$ 的函数：

$$\frac{\mathrm{d}\sigma(z)}{\mathrm{d}z} = \sigma(z)\,(1 - \sigma(z))$$

2.1.3　Logistic 回归模型

Logistic 回归模型假设函数为：

$$h_{w,b}(x) = \sigma(g(x)) = \frac{1}{1 + e^{-(w^{\mathrm{T}}x+b)}}$$

为了方便，通常将 b 纳入权向量 w，作为 w_0，同时为输入向量 x 添加一个常数 1 作为 x_0：

$$w = (b, w_1, w_2, \ldots w_n)^{\mathrm{T}}$$
$$x = (1, x_1, x_2, \ldots x_n)^{\mathrm{T}}$$

此时：

$$z = g(x) = w^{\mathrm{T}}x$$

假设函数为：

$$h_w(x) = \sigma(g(x)) = \frac{1}{1 + e^{-w^{\mathrm{T}}x}}$$

$h_w(x)$ 的输出是预测 x 为正例的概率，如果通过训练确定了模型参数 w，便可构建二元分类函数：

$$H(h_w(x)) = \begin{cases} 1, & h_w(x) \geqslant 0.5 \\ 0, & h_w(x) < 0.5 \end{cases}$$

2.1.4 极大似然法估计参数

确定了假设函数，接下来训练模型参数 w。对于给定的包含 m 个样本的数据集 D，可以使用极大似然估计法来估计 w。

根据 $h_w(x)$ 的概率意义，有：

$$P(y=1|x)=h_w(x)$$
$$P(y=0|x)=1-h_w(x)$$

综合上述二式可得出，训练集 D 中某样本 (x_i,y_i)，模型将输入实例 x_i 预测为类别 y_i 的概率为：

$$P(y=y_i|x_i;w)=h_w(x_i)^{y_i}(1-h_w(x_i))^{1-y_i}$$

训练集 D 中各样本独立同分布，因此我们定义似然函数 $L(w)$ 来描述训练集中 m 个样本同时出现的概率为：

$$\begin{aligned}L(w)&=\prod_{i=1}^{m}P(y=y_i|x_i;w)\\&=\prod_{i=1}^{m}h_w(x_i)^{y_i}(1-h_w(x_i))^{1-y_i}\end{aligned}$$

极大似然法估计参数 w 的核心思想是：选择参数 w，使得当前已经观测到的数据（训练集中的 m 个样本）最有可能出现（概率最大），即：

$$\hat{w}=\arg\max_{w}L(w)$$

$L(w)$是一系列项之积，求导比较麻烦，不容易找出其最大值点（即求出最大值）。ln函数是单调递增函数，因此可将问题转化为找出对数似然函数 $\ln(L(w))$ 的最大值点，即：

$$\hat{w}=\arg\max_{w}\ln(L(w))$$

根据定义，对数似然函数为：

$$l(w)=\ln(L(w))=\sum_{i=1}^{m}y_i\ln(h_w(x_i))+(1-y_i)\ln(1-h_w(x_i))$$

经观察可看出，以上对数似然函数是一系列项之和，求导简单，容易找到最大值点，即求出最大值。

2.1.5　梯度下降更新公式

习惯上，我们通常定义模型的损失函数，并求其最小值（找出最小值点）。对于 Logistic 回归模型，可以定义其损失函数为：

$$J(w) = -\frac{1}{m} l(w) = -\frac{1}{m} \sum_{i=1}^{m} y_i \ln(h_w(x_i)) + (1-y_i)\ln(1-h_w(x_i))$$

此时，求出损失函数最小值与求出对数似然函数最大值等价。求损失函数最小值，依然可以使用梯度下降算法，最终估计出模型参数 $\hat{w}$。

下面计算损失函数 $J(w)$ 的梯度，从而推出梯度下降算法中 w 的更新公式。

计算 $J(w)$ 对分量 w_j 的偏导数：

$$\begin{aligned}
\frac{\partial}{\partial w_j} J(w) &= -\frac{1}{m} \frac{\partial}{\partial w_j} \sum_{i=1}^{m} y_i \ln h_w(x_i) + (1-y_i)\ln(1-h_w(x_i)) \\
&= -\frac{1}{m} \sum_{i=1}^{m} y_i \frac{\partial}{\partial w_j} \ln h_w(x_i) + (1-y_i) \frac{\partial}{\partial w_j} \ln(1-h_w(x_i)) \\
&= -\frac{1}{m} \sum_{i=1}^{m} y_i \frac{1}{h_w(x_i)} \frac{\partial h_w(x_i)}{\partial z_i} \frac{\partial z_i}{w_j} + (1-y_i) \frac{1}{1-h_w(x_i)} - \frac{\partial h_w(x_i)}{\partial z_i} \frac{\partial z_i}{w_j} \\
&= -\frac{1}{m} \sum_{i=1}^{m} \left(y_i \frac{h_w(x_i)(1-h_w(x_i))}{h_w(x_i)} - (1-y_i) \frac{h_w(x_i)(1-h_w(x_i))}{1-h_w(x_i)} \right) \frac{\partial z_i}{w_j} \\
&= -\frac{1}{m} \sum_{i=1}^{m} (y_i - h_w(x_i)) \frac{\partial z_i}{w_j} \\
&= \frac{1}{m} \sum_{i=1}^{m} (h_w(x_i) - y_i) x_{ij}
\end{aligned}$$

其中，$h_w(x_i) - y_i$ 可解释为模型预测 x_i 为正例的概率与其实际类别之间的误差。

由此可推出梯度 $\nabla J(w)$ 计算公式为：

$$\nabla J(w) = \frac{1}{m} \sum_{i=1}^{m} (h_w(x_i) - y_i) x_i$$

对于随机梯度下降算法，每次只使用一个样本来计算梯度（m=1），相应梯度 $\nabla J(w)$ 计算公式为：

$$\nabla J(w) = (h_w(x_i) - y_i) x_i$$

假设梯度下降（或随机梯度下降）算法的学习率为 η，模型参数 w 的更新公式为：

$$w := w - \eta \nabla J(w)$$

2.2 Softmax 回归

Logistic 回归只能处理二元分类问题，在其基础上推广得到的 Softmax 回归可处理多元分类问题。Softmax 回归也被称为多元 Logistic 回归。

2.2.1 Softmax 函数

假设分类问题有 K 个类别，Softmax 对实例 x 的类别进行预测时，需分别计算 x 为每一个类别的概率，因此每个类别拥有各自独立的线性函数$g_j(x)$：

$$z_j = g_j(x) = w_j^{\mathrm{T}} x$$

这就意味着 w_j 有 K 个，它们构成一个矩阵：

$$W = \begin{bmatrix} w_1^{\mathrm{T}} \\ w_2^{\mathrm{T}} \\ \vdots \\ w_K^{\mathrm{T}} \end{bmatrix}$$

可定义 Softmax 回归的 $g(x)$ 函数为：

$$z = g(x) = Wx = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_K \end{bmatrix}$$

与 Logistic 回归的 logistic 函数相对应，Softmax 回归使用 softmax 函数来预测概率。

softmax 函数的输出为一个向量：

$$\sigma(z)=\begin{bmatrix}\sigma(z)_1\\ \sigma(z)_2\\ \vdots\\ \sigma(z)_K\end{bmatrix}$$

其中的分量 $\sigma(z)_j$ 即是模型预测 x 为第 j 个类别的概率。$\sigma(z)_j$ 定义如下：

$$\sigma(z)_j=\frac{e^{z_j}}{\sum_{k=1}^{K}e^{z_k}}$$

经观察可发现，logistic 函数实际上是 softmax 函数的特例：K=2 时，softmax 函数、分子分母同时除以 e^{z_j}，便是 logistic 函数的形式。

2.2.2　Softmax 回归模型

Softmax 回归模型假设函数为：

$$h_W(x)=\sigma(g(x))=\frac{1}{\sum_{k=1}^{K}e^{w_k^{\mathrm{T}}x}}\begin{bmatrix}e^{w_1^{\mathrm{T}}x}\\ e^{w_2^{\mathrm{T}}x}\\ \vdots\\ e^{w_K^{\mathrm{T}}x}\end{bmatrix}$$

$h_W(x)$ 的输出是模型预测 x 为各类别的概率，如果通过训练确定了模型参数 W，便可构建出多元分类函数：

$$H(h_W(x))=\arg\max_k h_W(x)_k=\arg\max_k(w_k^{\mathrm{T}}x)$$

2.2.3　梯度下降更新公式

Softmax 回归模型的损失函数被称为交叉熵，定义如下：

$$J(W)=-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{K}I(y_i=j)\ln h_W(x_i)_j$$

其中，I 为指示函数，当 $y_i=j$ 时为 1，否则为 0。经观察可发现，Logistic 回归的损失函数是 K = 2 时的交叉熵。

下面推导梯度下降算法中参数 W 的更新公式。W为矩阵，更新W即更新其中每

一个w_j，这就需要计算 $J(W)$ 对每一个 w_j 的梯度。推导过程与 Logistic 回归类似，这里直接给出计算公式：

$$\nabla_{w_j} J(W) = \frac{1}{m} \sum_{i=1}^{m} (h_W(x_i)_j - I(y_i = j)) x_i$$

其中，$h_W(x_i)_j - I(y_i = j)$ 可解释为模型预测 x_i 为第 j 类别的概率与其实际是否为第 j 类别（是为 1，不是为 0）之间的误差。

对于随机梯度下降算法，每次只使用一个样本来计算梯度（m=1），相应梯度 $\nabla_{w_j} J(w)$ 计算公式为：

$$\nabla_{w_j} J(W) = (h_W(x_i)_j - I(y_i = j)) x_i$$

假设梯度下降（或随机梯度下降）算法学习率为 η，w_j 的更新公式为：

$$w_j := w_j - \eta \nabla_{w_j} J(W)$$

最终得出，模型参数 W 的更新公式为：

$$W := W - \eta \begin{bmatrix} \nabla_{w_1} J(w)^{\mathrm{T}} \\ \nabla_{w_2} J(w)^{\mathrm{T}} \\ \vdots \\ \nabla_{w_K} J(w)^{\mathrm{T}} \end{bmatrix}$$

2.3 编码实现

2.3.1 Logistic 回归

我们基于梯度下降实现一个 Logistic 回归分类器，代码如下：

```
import numpy as np

class LogisticRegression:
    def __init__(self, n_iter=200, eta=1e-3, tol=None):
        # 训练迭代次数
        self.n_iter = n_iter
        # 学习率
        self.eta = eta
        # 误差变化阈值
```

```
        self.tol = tol
        # 模型参数 w(训练时初始化)
        self.w = None

    def _z(self, X, w):
        '''g(x)函数：计算 x 与 w 的内积.'''
        return np.dot(X, w)

    def _sigmoid(self, z):
        '''Logistic 函数'''
        return 1. / (1.   + np.exp(-z))

    def _predict_proba(self, X, w):
        '''h(x)函数：预测为正例(y=1)的概率.'''
        z = self._z(X, w)
        return self._sigmoid(z)

    def _loss(self, y, y_proba):
        '''计算损失'''
        m = y.size
        p = y_proba * (2 * y - 1) + (1 - y)
        return -np.sum(np.log(p)) / m

    def _gradient(self, X, y, y_proba):
        '''计算梯度'''
        return np.matmul(y_proba - y, X) / y.size

    def _gradient_descent(self, w, X, y):
        '''梯度下降算法'''

        # 若用户指定 tol，则启用早期停止法
        if self.tol is not None:
            loss_old = np.inf

        # 使用梯度下降，至多迭代 n_iter 次，更新 w
        for step_i in range(self.n_iter):
            # 预测所有点为 1 的概率
            y_proba = self._predict_proba(X, w)
```

```
            # 计算损失
            loss = self._loss(y, y_proba)
            print('%4i Loss: %s' % (step_i, loss))

            # 早期停止法
            if self.tol is not None:
                # 如果损失下降小于阈值，则终止迭代
                if loss_old - loss < self.tol:
                    break
                loss_old = loss

            # 计算梯度
            grad = self._gradient(X, y, y_proba)
            # 更新参数 w
            w -= self.eta * grad

    def _preprocess_data_X(self, X):
        '''数据预处理'''

        # 扩展 X，添加 x0 列并设置为 1
        m, n = X.shape
        X_ = np.empty((m, n + 1))
        X_[:, 0] = 1
        X_[:, 1:] = X

        return X_

    def train(self, X_train, y_train):
        '''训练'''

        # 预处理 X_train(添加 x0=1)
        X_train = self._preprocess_data_X(X_train)

        # 初始化参数向量 w
        _, n = X_train.shape
        self.w = np.random.random(n) * 0.05

        # 执行梯度下降训练 w
```

```
        self._gradient_descent(self.w, X_train, y_train)

    def predict(self, X):
        '''预测'''

        # 预处理 X_test(添加 x0=1)
        X = self._preprocess_data_X(X)

        # 预测为正例(y=1)的概率
        y_pred = self._predict_proba(X, self.w)

        # 根据概率预测类别，p>=0.5 为正例，否则为负例
        return np.where(y_pred >= 0.5, 1, 0)
```

上述代码简要说明如下（详细内容参看代码注释）：

- __init__()方法：构造器，保存用户传入的超参数。
- _z()方法：实现线性函数$g(x)$，计算w与x的内积（即点积，或称为数量积）。
- _sigmoid()方法：实现 logistic 函数 $\sigma(z)$。
- _predict_proba()方法：实现概率预测函数 $h_w(x)$，计算x 为正例的概率。
- _loss()方法：实现损失函数$J(w)$，计算当前w下的损失，该方法有以下两个用途。
  - 供早期停止法使用：如果用户通过超参数 tol 启用早期停止法，则调用该方法计算损失。
  - 方便调试：迭代过程中可以每次打印出当前损失，观察变化的情况。
- _gradient()方法：计算当前梯度 $\nabla J(w)$。
- _gradient_descent()方法：实现批量梯度下降算法。
- _preprocess_data_X()方法：对 X 进行预处理，添加x_0 列并设置为 1。
- train()方法：训练模型。该方法由 3 部分构成：
  - 对训练集的 X_train 进行预处理，添加 x_0 列并设置为 1。
  - 初始化模型参数 w，赋值较小的随机数。
  - 调用_gradient_descent()方法训练模型参数 w。
- predict()方法：预测。对于 X 中每个实例，若模型预测其为正例的概率大于等于 0.5，则判为正例，否则判为负例。

2.3.2 Softmax 回归

我们再基于随机梯度下降实现一个 Softmax 回归分类器，代码如下：

```
import numpy as np

class SoftmaxRegression:
    def __init__(self, n_iter=200, eta=1e-3, tol=None):
        # 训练迭代次数
        self.n_iter = n_iter
        # 学习率
        self.eta = eta
        # 误差变化阈值
        self.tol = tol
        # 模型参数 W(训练时初始化)
        self.W = None

    def _z(self, X, W):
        '''g(x)函数：计算 x 与 w 的内积.'''
        if X.ndim == 1:
            return np.dot(W, X)
        return np.matmul(X, W.T)

    def _softmax(self, Z):
        '''softmax 函数'''
        E = np.exp(Z)
        if Z.ndim == 1:
            return E / np.sum(E)
        return E / np.sum(E, axis=1, keepdims=True)

    def _predict_proba(self, X, W):
        '''h(x)函数：预测 y 为各个类别的概率.'''
        Z = self._z(X, W)
        return self._softmax(Z)

    def _loss(self, y, y_proba):
        '''计算损失'''
        m = y.size
```

```
        p = y_proba[range(m), y]
        return -np.sum(np.log(p)) / m

    def _gradient(self, xi, yi, yi_proba):
        '''计算梯度'''
        K = yi_proba.size
        y_bin = np.zeros(K)
        y_bin[yi] = 1

        return (yi_proba - y_bin)[:, None] * xi

    def _stochastic_gradient_descent(self, W, X, y):
        '''随机梯度下降算法'''

        # 若用户指定 tol，则启用早期停止法
        if self.tol is not None:
            loss_old = np.inf
            end_count = 0

        # 使用随机梯度下降，至多迭代 n_iter 次，更新 w
        m = y.size
        idx = np.arange(m)
        for step_i in range(self.n_iter):
            # 计算损失
            y_proba = self._predict_proba(X, W)
            loss = self._loss(y, y_proba)
            print('%4i Loss: %s' % (step_i, loss))

            # 早期停止法
            if self.tol is not None:
                # 随机梯度下降的 loss 曲线不像批量梯度下降那么平滑(上下起伏)，
                # 因此连续多次(而非一次)下降到小于阈值，才终止迭代
                if loss_old - loss < self.tol:
                    end_count += 1
                    if end_count == 5:
                        break
                else:
                    end_count = 0
```

```
73.
74.                 loss_old = loss
75.
76.             # 每一轮迭代之前，随机打乱训练集
77.             np.random.shuffle(idx)
78.             for i in idx:
79.                 # 预测 xi 为各类别的概率
80.                 yi_proba = self._predict_proba(X[i], W)
81.                 # 计算梯度
82.                 grad = self._gradient(X[i], y[i], yi_proba)
83.                 # 更新参数 w
84.                 W -= self.eta * grad
85.
86.
87.     def _preprocess_data_X(self, X):
88.         '''数据预处理'''
89.
90.         # 扩展 X，添加 x0 列并设置为 1
91.         m, n = X.shape
92.         X_ = np.empty((m, n + 1))
93.         X_[:, 0] = 1
94.         X_[:, 1:] = X
95.
96.         return X_
97.
98.     def train(self, X_train, y_train):
99.         '''训练'''
100.
101.            # 预处理 X_train(添加 x0=1)
102.            X_train = self._preprocess_data_X(X_train)
103.
104.            # 初始化参数向量 W
105.            k = np.unique(y_train).size
106.            _, n = X_train.shape
107.            self.W = np.random.random((k, n)) * 0.05
108.
109.            # 执行随机梯度下降训练 W
```

```
110.            self._stochastic_gradient_descent(self.W, X_train,
                                                  y_train)
111.
112.        def predict(self, X):
113.            '''预测'''
114.
115.            # 预处理 X_test(添加 x0=1)
116.            X = self._preprocess_data_X(X)
117.
118.            # 对每个实例计算向量 z
119.            Z = self._z(X, self.W)
120.
121.            # 以向量 z 中最大分量的索引作为预测的类别
122.            return np.argmax(Z, axis=1)
```

上述代码简要说明如下（详细内容参看代码注释）。

- __init__()方法：构造器，保存用户传入的超参数。
- _z()方法：实现线性函数 $g(x)$，计算各个 w_j 与 x 的内积。
- _softmax()方法：实现 softmax 函数 $\sigma(z)$。
- _predict_proba()方法：实现概率预测函数 $h_W(x)$，计算 x 为各个类别的概率。
- _loss()方法：实现损失函数 $J(w)$，计算当前w下的损失。该方法有以下两个用途：
  - 供早期停止法使用：如果用户通过超参数 tol 启用早期停止法，则调用该方法计算损失。
  - 方便调试：迭代过程中可以每次打印出当前损失，观察变化的情况。
- _gradient()方法：计算当前梯度 $\nabla J(w)$。
- _stochastic_gradient_descent()方法：实现随机梯度下降算法。
- _preprocess_data_X()方法：对 X 进行预处理，添加 x_0 列并设置为 1。
- train()方法：训练模型。该方法由 3 部分构成：
  - 对训练集的 X_train 进行预处理，添加 x_0 列并设置为 1。
  - 初始化模型参数 w，赋值较小的随机数。
  - 调用_stochastic_gradient_descent()方法训练模型参数 W。
- predict()方法：预测。对于 X 中每个实例，计算由各线性函数值 z_j 构成的向量 z，以其最大分量的索引作为预测类别。

2.4 项目实战

最后，我们分别来做一个 Logistic 回归和一个 Softmax 回归的实战项目：使用 Logistic 回归和 Softmax 回归分别来鉴别红酒的种类，如表 2-1 所示。

表2-1 红酒数据集（https://archive.ics.uci.edu/ml/datasets/wine）

列号	列名	含义	特征/类标记	可取值
1	class	葡萄酒类别	类标记	1,2,3
2	Alcohol	酒精度	特征	实数
3	Malic acid	苹果酸	特征	实数
4	Ash	灰	特征	实数
5	Alcalinity of ash	灰分的碱度	特征	实数
6	Magnesium	镁含量	特征	实数
7	Total phenols	总酚类	特征	实数
8	Flavonoids	黄烷类	特征	实数
9	Nonflavonoid phenols	非类黄烷酚	特征	实数
10	Proanthocyanins	原花青素	特征	实数
11	Color intensity	颜色强度	特征	实数
12	Hue	色相	特征	实数
13	OD280/OD315 of diluted wines	稀释葡萄酒的 OD280/ OD315	特征	实数
14	Proline	脯氨酸	特征	实数

数据集总共有 178 条数据，其中每一行包含一个红酒样本的类标记以及 13 个特征，这些特征是酒精度、苹果酸浓度等化学指标。红酒的种类有 3 种，Softmax 回归可以处理多元分类问题，而 Logistic 回归只能处理二元分类问题，因此在做 Logistic 回归项目时，我们从数据集中去掉其中的一类红酒样本，使用剩下的两类红酒样本作为训练数据。

读者可使用任意方式将红酒数据集文件 letter-recognition.data 下载到本地。此文件所在的 URL 为：https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data。

2.4.1 Logistic 回归

1. 准备数据

首先，调用 Numpy 的 genfromtxt 函数加载数据集：

```
>>> import numpy as np
>>> X = np.genfromtxt('wine.data', delimiter=',', usecols=range(1,
14))
>>> X
array([[1.  423e+01, 1.710e+00, 2.430e+00, ..., 1.040e+00,
        3.920e+00, 1.065e+03],
       [1.320e+01, 1.780e+00, 2.140e+00, ..., 1.050e+00, 3.400e+00,
         1.050e+03],
       [1.316e+01, 2.360e+00, 2.670e+00, ..., 1.030e+00, 3.170e+00,
         1.185e+03],
       ...,
       [1.327e+01, 4.280e+00, 2.260e+00, ..., 5.900e-01, 1.560e+00,
        8.350e+02],
       [1.317e+01, 2.590e+00, 2.370e+00, ..., 6.000e-01, 1.620e+00,
        8.400e+02],
       [1.413e+01, 4.100e+00, 2.740e+00, ..., 6.100e-01, 1.600e+00,
        5.600e+02]])
>>> y = np.genfromtxt('wine.data', delimiter=',', usecols=0)
>>> 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., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2., 2., 2., 2., 2.,
2.,
       2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
2.,
       2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
2.,
       2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 3., 3., 3., 3.,
3.,
       3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
3.,
```

```
    28.       3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
3.,
    29.       3., 3., 3., 3., 3., 3., 3., 3.])
```

在这个项目中，我们使用 Logistic 回归鉴别第 1 类和第 2 类红酒，因此将数据集中第 3 类红酒样本去除：

```
    1.  >>> idx = y != 3
    2.  >>> X = X[idx]
    3.  >>> X
    4.  array([[1.423e+01, 1.710e+00, 2.430e+00, ..., 1.040e+00, 3.920e+00,
    5.         1.065e+03],
    6.        [1.320e+01, 1.780e+00, 2.140e+00, ..., 1.050e+00, 3.400e+00,
    7.         1.050e+03],
    8.        [1.316e+01, 2.360e+00, 2.670e+00, ..., 1.030e+00, 3.170e+00,
    9.         1.185e+03],
    10.       ...,
    11.       [1.179e+01, 2.130e+00, 2.780e+00, ..., 9.700e-01, 2.440e+00,
    12.        4.660e+02],
    13.       [1.237e+01, 1.630e+00, 2.300e+00, ..., 8.900e-01, 2.780e+00,
    14.        3.420e+02],
    15.       [1.204e+01, 4.300e+00, 2.380e+00, ..., 7.900e-01, 2.570e+00,
    16.        5.800e+02]])
    17. >>> y = y[idx]
    18. >>> y
    19. array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1., 1.,
    20.       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1.,
    21.       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1.,
    22.       1., 1., 1., 1., 1., 1., 1., 1., 2., 2., 2., 2., 2., 2., 2.,
2.,
    23.       2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
2.,
    24.       2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
2.,
    25.       2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
2.,
```

```
    26.       2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.])
```

另外，目前 y 中的类标记为 1 和 2，转换为算法所使用的 0 和 1：

```
    1.  >>> y -= 1
    2.  >>> y
    3.  array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0.,
    4.        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
    5.        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0.,
    6.        0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1.,
1.,
    7.        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1.,
    8.        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1.,
    9.        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
1.,
    10.       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

至此，数据准备完毕。

2. 模型训练与测试

LogisticRegression 的超参数有：

（1）梯度下降最大迭代次数 n_iter

（2）学习率 eta

（3）损失降低阈值 tol（tol 不为 None 时，开启早期停止法）

先以超参数（n_iter=2000，eta=0.01，tol=0.0001）创建模型：

```
    1.  >>> from logistic_regression import LogisticRegression
    2.  >>> clf = LogisticRegression(n_iter=2000, eta=0.01, tol=0.0001)
```

然后，调用 sklearn 中的 train_test_split 函数将数据集切分为训练集和测试集（比例为 7:3）：

```
    1.  >>> from sklearn.model_selection import train_test_split
    2.  >>> X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3)
```

在第 1 章中曾讨论过，应用梯度下降算法时，应保证各特征值相差不大。观察下面的数据集特征均值及方差：

```
>>> X.mean(axis=0)
array([1.29440769e+01, 1.96807692e+00, 2.34046154e+00,
1.87853846e+01,
       9.99000000e+01, 2.52269231e+00, 2.49000000e+00,
3.30230769e-01,
       1.75238462e+00, 4.19476923e+00, 1.05889231e+00,
2.95438462e+00,
       7.90092308e+02])
>>> X.var(axis=0)
array([7.83834917e-01, 7.68387840e-01, 8.76259408e-02,
1.14741710e+01,
       2.34766923e+02, 2.95165828e-01, 5.40110769e-01,
1.18084083e-02,
       2.88898160e-01, 2.62283572e+00, 2.82373576e-02,
2.24046160e-01,
       1.23309545e+05])
```

发现其中一些特征值差别较大，因此调用 sklearn 中的 StandardScaler 函数对各特征值进行缩放：

```
>>> from sklearn.preprocessing import StandardScaler
>>> ss = StandardScaler()
>>> ss.fit(X_train)
StandardScaler(copy=True, with_mean=True, with_std=True)
>>> X_train_std = ss.transform(X_train)
>>> X_test_std = ss.transform(X_test)
>>> X_train_std[:3]
array([[ 0.11880613,  0.14021041,  2.90295463,  1.6313308 ,
1.44203794,
         0.10847775,  0.18001387,  1.28632362,  0.25373657,
-0.38491269,
         0.44287056,  0.52346046,  0.09647931],
       [-0.61993088,  0.71134502, -0.21844373,  0.81967906,
-0.65855782,
        -1.71247403, -0.96286489,  3.04585965, -0.63984036,
-0.91480453,
        -1.32357907,  0.75116075, -1.33937419],
```

```
    14.       [-0.36195923, -0.64795535, -1.03986435, -0.58718395,
-0.04073554,
    15.        -1.06076497, -1.54790997,  1.84196658, -2.06956344,
0.92175241,
    16.        -0.53849034, -3.  14251427, -0.96298541]])
    17. >>> X_test_std[:3]
    18. array([[-0.72546473, -0.94494535, -0.1855869 , -0.80362441,
0.02104669,
    19.        -1.00326123, -1.98329235,  2.76803817, -2.44486575,
-0.57157914,
    20.         1.22795929, -2.96035404, -0.32173045],
    21.       [ 1.06861084, -0.47661497,  1.09582927,  1.6313308 ,
-0.90568673,
    22.         0.72185098,  0.42491646, -1.12146252,  0.16437888,
-0.50534266,
    23.         1.94762395,  0.43238034, -1.07450801],
    24.       [-0.51439702, -0.22531574, -1.17129164,  0.41385319,
-0.96746896,
    25.        -0.71574253, -0.85401929, -0.10278377, -0.53261113,
-0.77028858,
    26.        -0.14594598,  1.36595154, -0.34403497]])
```

接下来，训练模型：

```
    1.  >>> clf.train(X_train_std, y_train)
    2.     0 Loss: 0.7330723153684124
    3.     1 Loss: 0.72438967420864
    4.     2 Loss: 0.7158883537077279
    5.     3 Loss: 0.7075647055742007
    6.     4 Loss: 0.699415092738375
    7.     5 Loss: 0.691435894577802
    8.      ...
    9.   710 Loss: 0.12273875001717284
    10.  711 Loss: 0.12263819712826703
    11.  712 Loss: 0.12253785223988992
    12.  713 Loss: 0.1224377146473229
    13.  714 Loss: 0.12233778364922131
```

经过 700 多次迭代后算法收敛。图 2-3 所示为训练过程中的损失（loss）曲线。

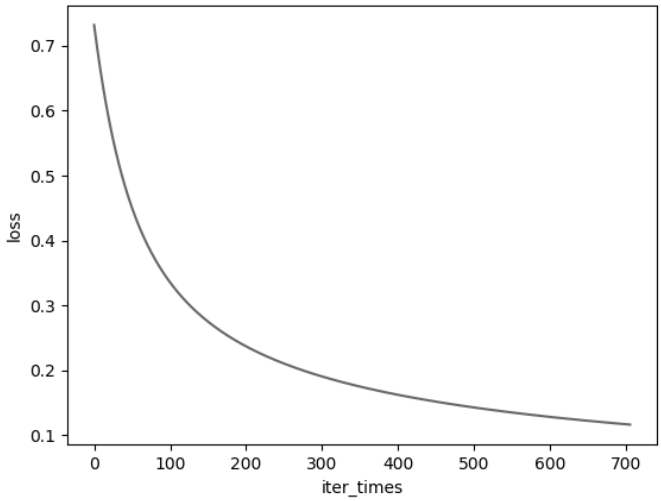

图 2-3

使用已训练好的模型对测试集中的实例进行预测，并调用 sklearn 中的 accuracy_score 函数计算预测的准确率：

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = clf.predict(X_test_std)
>>> accuracy = accuracy_score(y_test, y_pred)
>>> accuracy
1.0
```

单次测试一下，预测的准确率为 100%，再进行多次（50 次）反复测试，观察平均的预测准确率：

```
>>> def test(X, y):
...     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
...
...     ss = StandardScaler()
...     ss.fit(X_train)
...     X_train_std = ss.transform(X_train)
...     X_test_std = ss.transform(X_test)
...
...     clf = LogisticRegression(n_iter=2000, eta=0.01, tol=0.0001)
...     clf.train(X_train_std, y_train)
```

```
11. ...
12. ...     y_pred = clf.predict(X_test_std)
13. ...     accuracy = accuracy_score(y_test, y_pred)
14. ...     return accuracy
15. ...
16. >>> accuracy_mean = np.mean([test(X, y) for _ in range(50)])
17. >>> accuracy_mean
18. 0.9805128205128206
```

50 次测试平均的预测准确率为 98.05%，这表明几乎只有一个实例被预测错误，结果令人满意。读者还可以尝试使用其他超参数的组合创建模型，但该分类问题比较简单，性能提升空间不大。

至此，Logistic 回归项目就完成了。

2.4.2　Softmax 回归

1. 准备数据

除了无须去掉第 3 类红酒样本外，Softmax 回归项目的数据准备工作与 Logistic 回归项目的数据准备工作完全相同。

首先，调用 Numpy 的 genfromtxt 函数加载数据集：

```
1.  >>> import numpy as np
2.  >>> X = np.genfromtxt('wine.data', delimiter=',', usecols=range(1,
14))
3.  >>> y = np.genfromtxt('wine.data', delimiter=',', usecols=0)
```

然后，将目前 y 中的类标记为(1,2,3)，转换为算法所使用的(0,1,2)：

```
1.  >>> y -= 1
```

至此，数据准备完毕。

2. 模型训练与测试

Softmax 回归项目中的模型训练与测试过程与之前 Logistic 回归项目中的完全相同，以下叙述中某些细节不再重复。

SoftmaxRegression 的超参数与 LogisticRegression 相同：

（1）梯度下降最大迭代次数 n_iter

（2）学习率 eta

（3）损失降低阈值 tol（tol 不为 None 时，开启早期停止法）

我们依然使用超参数（n_iter=2000，tol=0.01，eta=0.0001）创建模型：

```
>>> from softmax_regression import SoftmaxRegression
>>> clf = SoftmaxRegression(n_iter=2000, eta=0.01, tol=0.0001)
```

将数据集切分为训练集和测试集（比例为 7:3）：

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3)
```

对各特征值进行缩放：

```
>>> from sklearn.preprocessing import StandardScaler
>>> ss = StandardScaler()
>>> ss.fit(X_train)
StandardScaler(copy=True, with_mean=True, with_std=True)
>>> X_train_std = ss.transform(X_train)
>>> X_test_std = ss.transform(X_test)
```

训练模型：

```
>>> clf.train(X_train_std, y_train)
   0 Loss: 1.1483823399828617
   1 Loss: 0.3360318473642378
   2 Loss: 0.22362742655678353
   3 Loss: 0.17673512423650206
   4 Loss: 0.1500298205757405
   5 Loss: 0.13187232998549944
   ...
 124 Loss: 0.016775944169047555
 125 Loss: 0.016676511693757688
 126 Loss: 0.01657807933519901
 127 Loss: 0.016480527435067803
 128 Loss: 0.01638475036830267
 129 Loss: 0.016289674417589398
```

使用已训练好的模型对测试集进行预测，并计算预测的准确率：

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = clf.predict(X_test_std)
>>> accuracy = accuracy_score(y_test, y_pred)
```

```
>>> accuracy
0.9814814814814815
```

单次测试一下，预测的准确率为 98.15%，同样，再进行多次（50 次）反复测试，观察平均的预测准确率：

```
>>> def test(X, y):
...     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
...
...     ss = StandardScaler()
...     ss.fit(X_train)
...     X_train_std = ss.transform(X_train)
...     X_test_std = ss.transform(X_test)
...
...     clf = SoftmaxRegression(n_iter=2000, eta=0.01, tol=0.0001)
...     clf.train(X_train_std, y_train)
...
...     y_pred = clf.predict(X_test_std)
...     accuracy = accuracy_score(y_test, y_pred)
...     return accuracy
...
>>> accuracy_mean = np.mean([test(X, y) for _ in range(50)])
>>> accuracy_mean
0.9803703703703703
```

50 次测试平均的预测准确率为 98.04%，与之前的 Logistic 回归性能几乎相同。

至此，Softmax 回归项目也完成了。