

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：

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

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 ：

$$\begin{aligned} \mathbf{w} &= (b, w_1, w_2, \dots, w_n)^T \\ \mathbf{x} &= (1, x_1, x_2, \dots, x_n)^T \end{aligned}$$

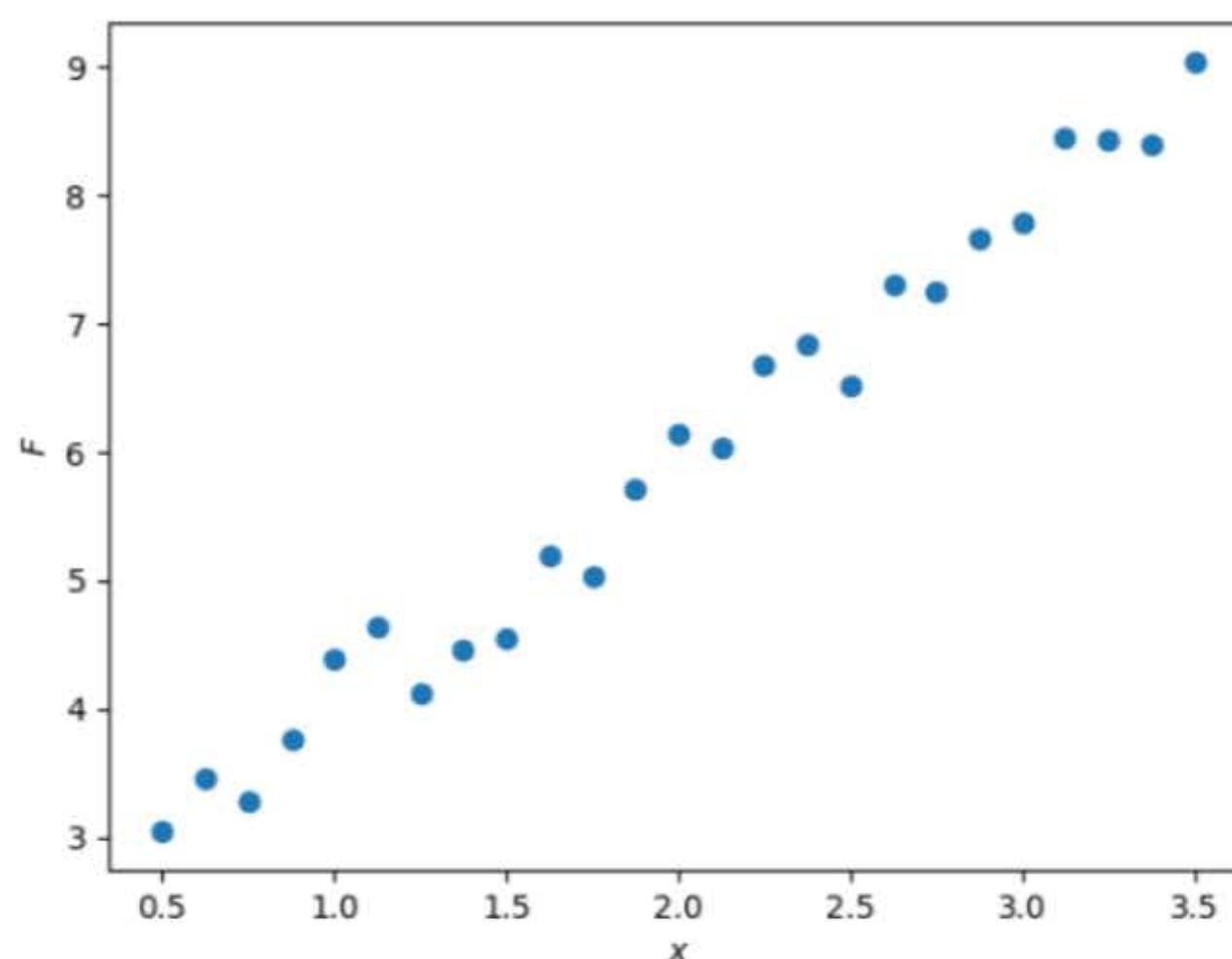


图 1-1

此时，假设函数为：

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

其中， $\mathbf{w} \in \mathbf{R}^{n+1}$ ，通过训练确定模型参数 \mathbf{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 \mathbf{D} 有 m 个样本，均方误差损失函数定义为：

$$\begin{aligned} J(\mathbf{w}) &= \frac{1}{2m} \sum_{i=1}^m (h_{\mathbf{w}}(\mathbf{x}_i) - y_i)^2 \\ &= \frac{1}{2m} \sum_{i=1}^m (\mathbf{w}^T \mathbf{x}_i - y_i)^2 \end{aligned}$$

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 \mathbf{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：

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

以上公式使用矩阵运算描述形式更为简洁，设：

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

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

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

那么，梯度计算公式可写为：

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

令梯度为 0，解得：

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

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 $X^T X$ 求了逆矩阵，这要求 $X^T X$ 是满秩的。然而实际应用中， $X^T X$ 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。

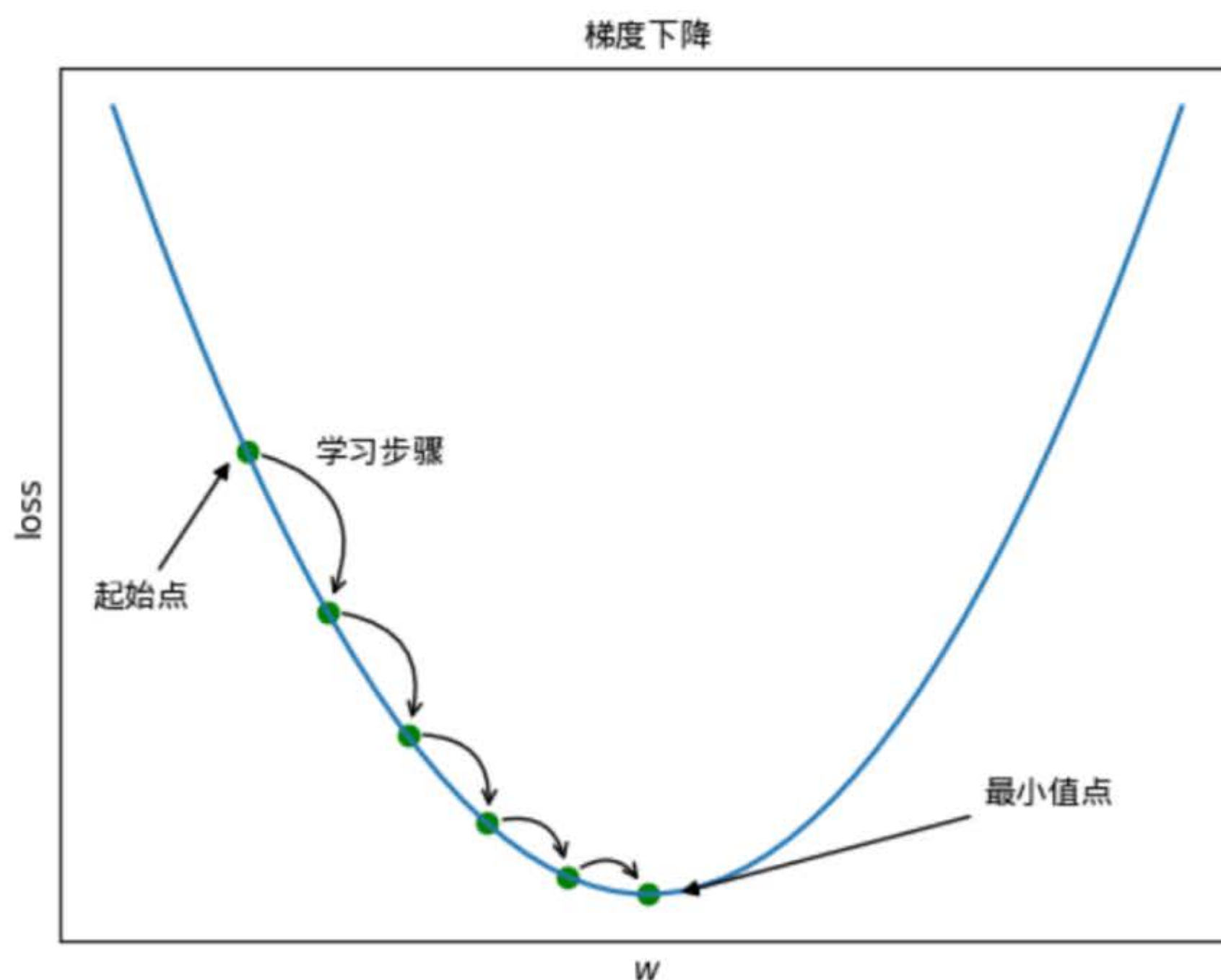


图 1-2

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。

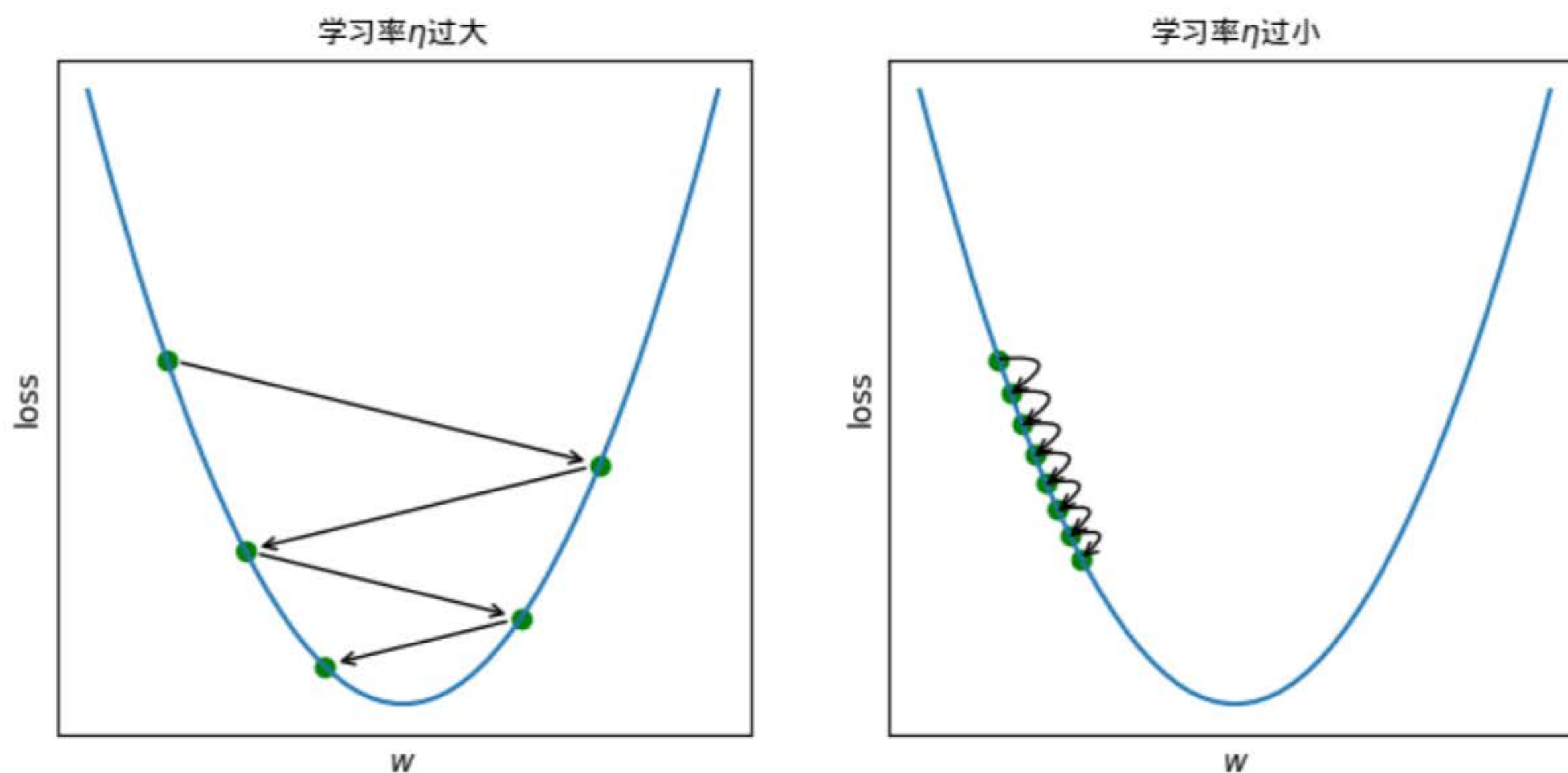


图 1-3

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。

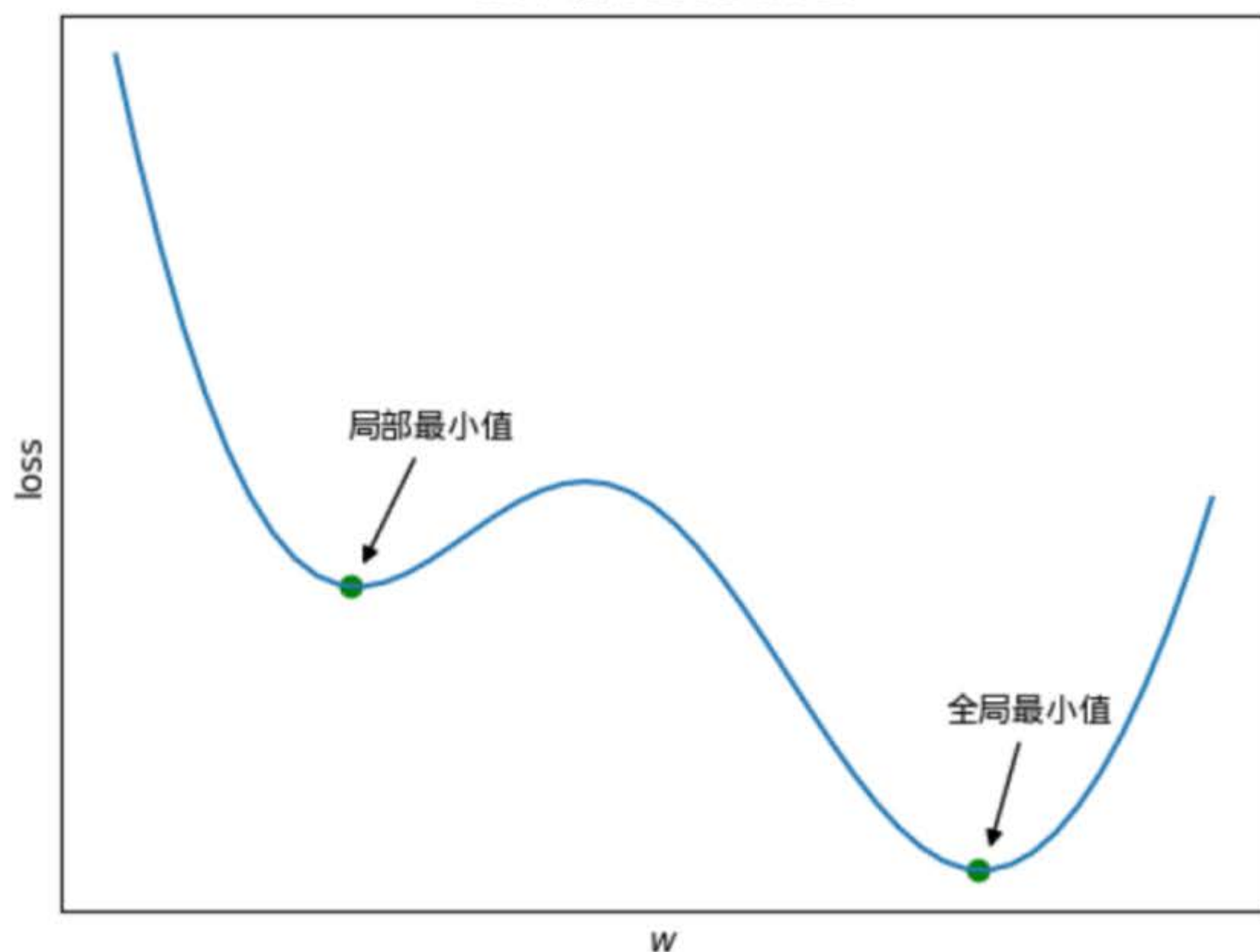


图 1-4

在 1.2 节中，我们计算出线性回归损失函数的梯度为：

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

设学习率为 η ，梯度下降算法的参数更新公式为：

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

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 \mathbf{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：

$$\nabla J(w) = (w^T x_i - y_i) x_i$$

设学习率为 η ，随机梯度下降算法的参数更新公式为：

$$w := w - \eta (w^T x_i - y_i) x_i$$

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：

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

设学习率为 η ，小批量梯度下降算法的参数更新公式为：

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

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：


```

1. import numpy as np
2.
3. class OLSLinearRegression:
4.
5.     def _ols(self, X, y):
6.         '''最小二乘法估算 w'''
7.         tmp = np.linalg.inv(np.matmul(X.T, X))
8.         tmp = np.matmul(tmp, X.T)
9.         return np.matmul(tmp, y)
10.
11.         # 若使用较新的 Python 和 Numpy 版本, 可使用如下实现
12.         # return np.linalg.inv(X.T @ X) @ X.T @ y
13.
14.     def _preprocess_data_X(self, X):
15.         '''数据预处理'''
16.
17.         # 扩展 X, 添加 x0 列并设置为 1
18.         m, n = X.shape
19.         X_ = np.empty((m, n + 1))
20.         X_[:, 0] = 1
21.         X_[:, 1:] = X
22.
23.         return X_
24.
25.     def train(self, X_train, y_train):
26.         '''训练模型'''
27.
28.         # 预处理 X_train (添加 x0=1)
29.         X_train = self._preprocess_data_X(X_train)
30.
31.         # 使用最小二乘法估算 w
32.         self.w = self._ols(X_train, y_train)
33.
34.     def predict(self, X):
35.         '''预测'''
36.         # 预处理 X_train (添加 x0=1)
37.         X = self._preprocess_data_X(X)
38.         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 $\hat{\mathbf{w}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$ 。
- `_preprocess_data_X()`方法：对 \mathbf{X} 进行预处理，添加 \mathbf{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 \mathbf{w} ，并保存。
- `predict()`方法：预测，实现函数 $h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$ ，对 \mathbf{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：

```
1. import numpy as np
2.
3. class GDLinearRegression:
4.
5.     def __init__(self, n_iter=200, eta=1e-3, tol=None):
6.         # 训练迭代次数
7.         self.n_iter = n_iter
8.         # 学习率
9.         self.eta = eta
10.        # 误差变化阈值
11.        self.tol = tol
12.        # 模型参数 w (训练时初始化)
13.        self.w = None
14.
15.    def _loss(self, y, y_pred):
16.        '''计算损失'''
17.        return np.sum((y_pred - y) ** 2) / y.size
18.
19.    def _gradient(self, X, y, y_pred):
20.        '''计算梯度'''
21.        return np.matmul(y_pred - y, X) / y.size
22.
23.    def _gradient_descent(self, w, X, y):
24.        '''梯度下降算法'''
25.
26.        # 若用户指定 tol，则启用早期停止法
27.        if self.tol is not None:
28.            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 $h_w(x) = w^T x$ 。
- `_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)
```



```

3. >>> X = data[:, :-1]
4. >>> X
5. array([[ 7.4 ,  0.7 ,  0.   , ...,  3.51 ,  0.56 ,  9.4  ],
6.        [ 7.8 ,  0.88 ,  0.   , ...,  3.2 ,  0.68 ,  9.8  ],
7.        [ 7.8 ,  0.76 ,  0.04 , ...,  3.26 ,  0.65 ,  9.8  ],
8.        ...,
9.        [ 6.3 ,  0.51 ,  0.13 , ...,  3.42 ,  0.75 , 11.   ],
10.       [ 5.9 ,  0.645,  0.12 , ...,  3.57 ,  0.71 , 10.   2 ],
11.       [ 6.   ,  0.31 ,  0.47 , ...,  3.39 ,  0.66 , 11.   ]])
12. >>> y = data[:, -1]
13. >>> y
14. 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：

```

1. >>> from linear_regression import OLSLinearRegression
2. >>> 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）：

```

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)

```

接下来，训练模型：

```

1. >>> 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：


```

1. >>> y_pred = ols_lr.predict(X_test)
2. >>> y_pred
3. array([5.97884966, 5.97298391, 5.24300126, 5.06622202, 5.34749778,
4.         5.75578547, 5.12227758, 5.42068169, 5.45180575, 6.13254774,
5.         5.34770062, 5.83609291, 6.05261885, 6.12793756, 6.02340132,
6.         ...
7.         5.57593107, 6.52179897, 5.96058307, 5.34186329, 5.72550139,
8.         5.22740437, 5.07311142, 5.78794577, 5.9205651 , 5.87093099,
9.         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:

```

1. >>> mse = mean_squared_error(y_test, y_pred)
2. >>> mse
3. 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:

```

1. >>> y_train_pred = ols_lr.predict(X_train)
2. >>> mse_train = mean_squared_error(y_train, y_train_pred)
3. >>> mse_train
4. 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。



注意

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:

```

1. >>> from sklearn.metrics import mean_absolute_error
2. >>> mae = mean_absolute_error(y_test, y_pred)
3. >>> mae
4. 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：

```
1. >>> from linear_regression import GDLinearRegression
2. >>> 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：

```
1. >>> gd_lr.train(X_train, y_train)
2.      0 Loss: 12.200517575720156
3.      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：

```
1. >>> gd_lr = GDLinearRegression(n_iter=3000, eta=0.0001, tol=0.00001)
2. >>> gd_lr.train(X_train, y_train)
3.      0 Loss: 16.023304785489813
4.      1 Loss: 11.497026121171373
5.      2 Loss: 9.592720523188712
6.      3 Loss: 8.75383172330532
7.      4 Loss: 8.348468678827183
8.      5 Loss: 8.120102508707241
9.      ...
10. 2994 Loss: 0.5387288272167047
11. 2995 Loss: 0.5387143011814725
12. 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 \mathbf{X} 中各特征尺寸相差较大造成的，观察 \mathbf{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 x_1 特征尺寸比 x_2 的小很多（如图 1-5 所示），通常 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 x_2 轴的方向下山，再沿着 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。

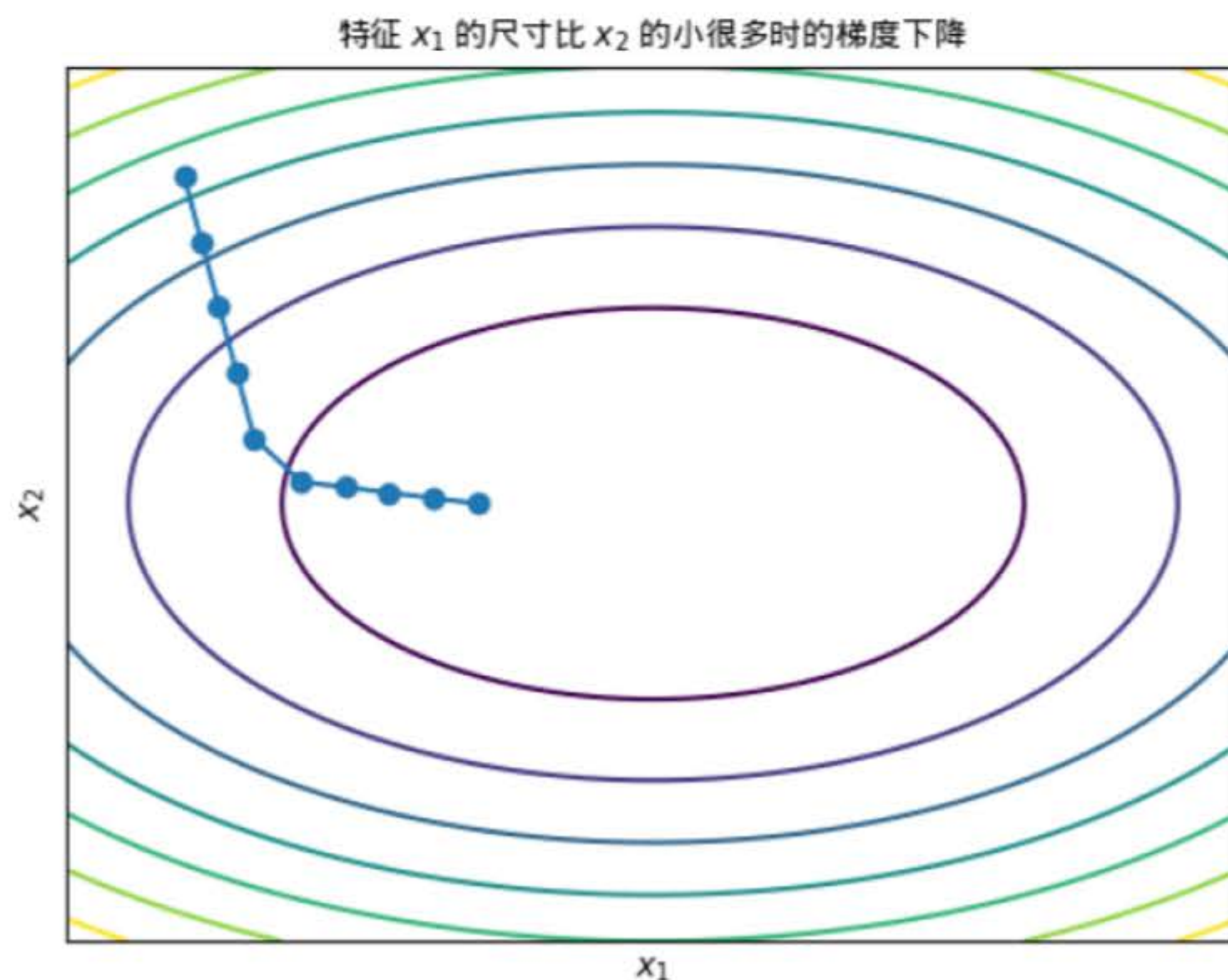


图 1-5

相反，如果 x_1 和 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。

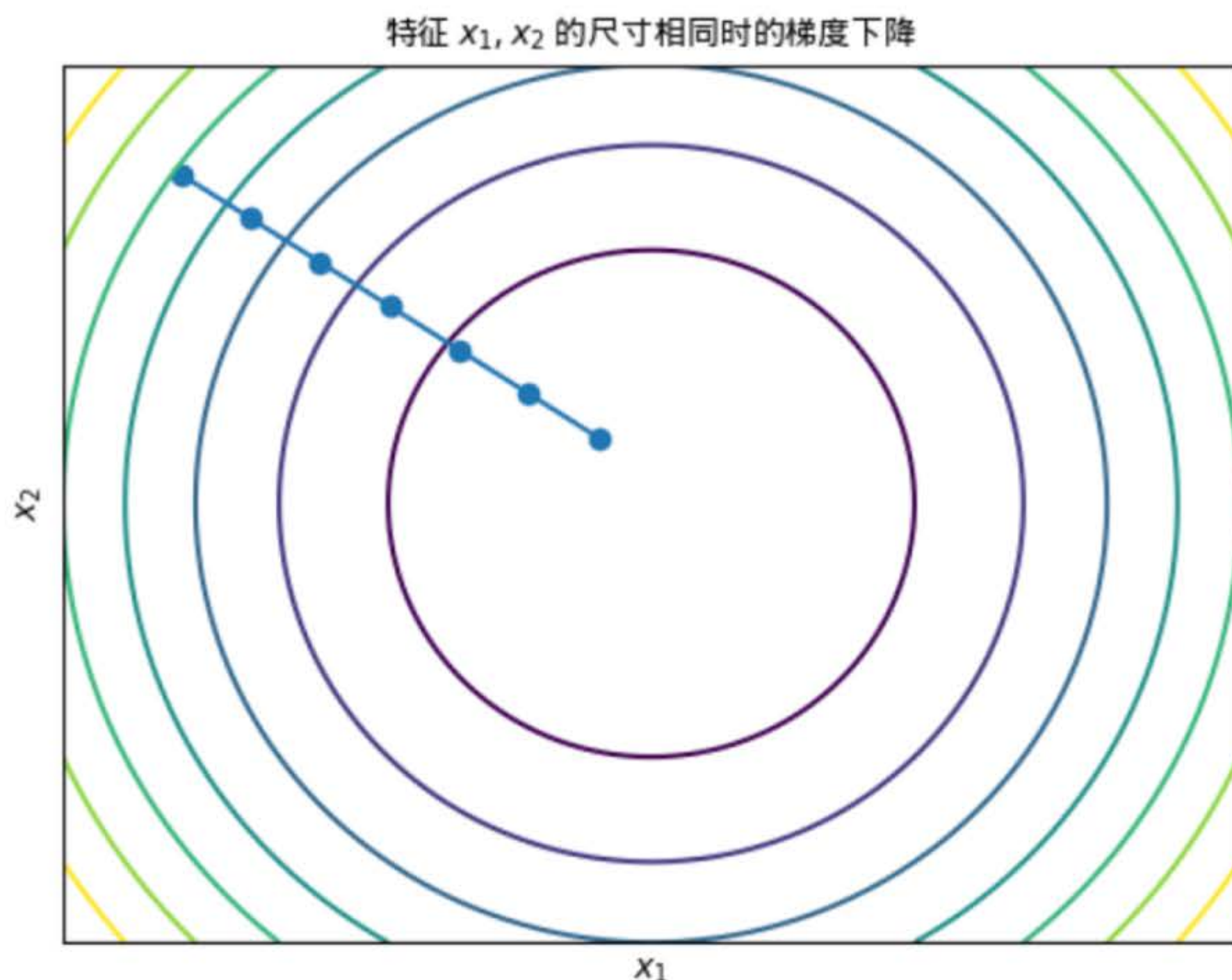


图 1-6

接下来我们把 \mathbf{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：


```

1. >>> from sklearn.preprocessing import StandardScaler
2. >>> ss = StandardScaler()
3. >>> ss.fit(X_train)
4. StandardScaler(copy=True, with_mean=True, with_std=True)
5. >>> X_train_std = ss.transform(X_train)
6. >>> X_test_std = ss.transform(X_test)
7. >>> X_train_std[:3]
8. array([[ 2.06525234, -0.52177354,  2.00384591,  0.41731378,
9.          0.43302079, -0.54794448, -0.71530754,  1.69254258,
10.         -1.04504699,  1.06645239, -0.00577349],
11.        [-0.54836368,  0.42987536, -0.07611659, -0.31883207,
12.         -0.11252578,  0.11774751,  1.37454564, -0.33359707,
13.         -0.11983048, -0.58506021, -0.56891899],
14.        [-0.19988155, -0.01795942, -0.12811565, -0.39244666,
15.         -0.21743858,  0.30794522,  0.39108532, -0.3547027 ,
16.         -0.05374359, -1.04064989, -0.19348865]])
17. >>> X_test_std[:3]
18. array([[ -1.41956902,  0.14997863, -0.90810159, -0.24521749,
19.          -0.55315954,  2.20992233,  0.14522024, -0.88234323,
20.          1.40016808,  0.553914,  0.74508718],
21.        [ 0.55516308, -0.2418768 ,  0.0798806 , -0.31883207,
22.          0.81070687, -0.9283399 , -0.93043948,  0.00409287,
23.         -0.58243874,  0.32611916,  0.74508718],
24.        [-0.43220297,  0.68178242, -1.27209503, -0.46606124,
25.         -0.0495781, -0.16754906,  0.32961905,  0.05685692,
26.          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未知。

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：

```

1. >>> gd_lr = GDLinearRegression(n_iter=3000, eta=0.05, tol=0.00001)
2. >>> gd_lr.train(X_train_std, y_train)
3.    0 Loss: 32.    37093970975737
4.    1 Loss: 29.    238242349047354
5.    2 Loss: 26.    4144603303325
6.    3 Loss: 23.    868826231616428

```



```
7.      4 Loss: 21.    573695276485843
8.      5 Loss: 19.    50421695549397
9.      6 Loss: 17.    63804348681383
10.     ...
11.   128 Loss: 0.4288433323855949
12.   129 Loss: 0.428828969543995
13.   130 Loss: 0.42881542387707694
14.   131 Loss: 0.4288026302147262
15.   132 Loss: 0.4287905293029377
16.   133 Loss: 0.42877906724161524
17.   134 Loss: 0.4287681949766629
18.   135 Loss: 0.4287578678410973
19.   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：

```
1. >>> y_pred = gd_lr.predict(X_test_std)
2. >>> mse = mean_squared_error(y_test, y_pred)
3. >>> mse
4. 0.39311865138396274
5. >>> mae = mean_absolute_error(y_test, y_pred)
6. >>> mae
7. 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。