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，$w \in \mathbf{R}^n$ 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：

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

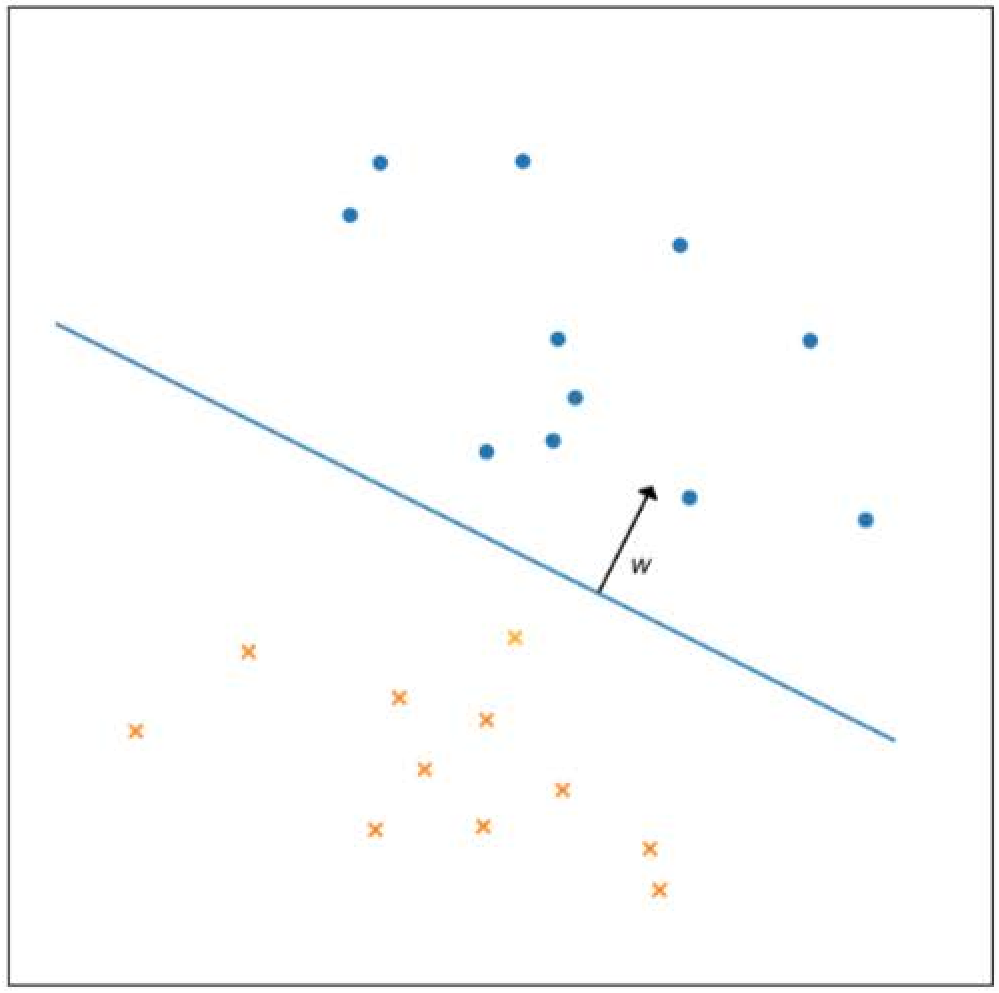

图 2-1

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。

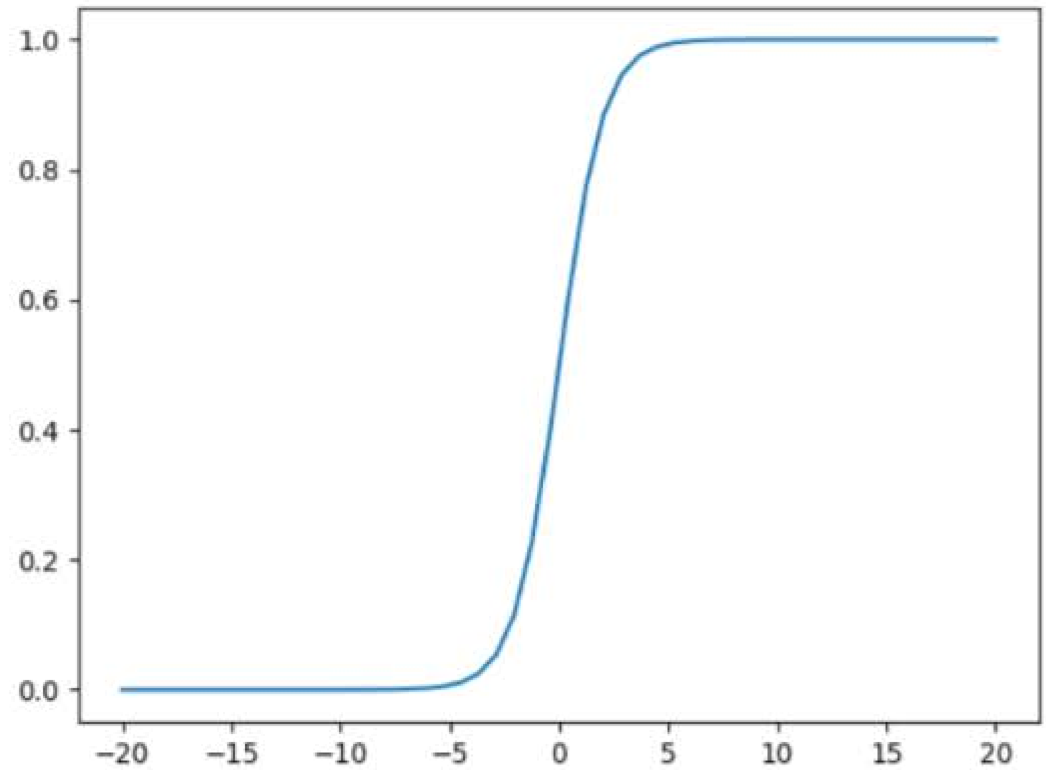

图 2-2

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 $\boldsymbol{x}$ 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，$\boldsymbol{x}$ 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 $\boldsymbol{p}$ 与该事件不发生的概率 $1-\boldsymbol{p}$ 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：

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

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 $\boldsymbol{w}$。对于给定的包含 m 个样本的数据集 $\boldsymbol{D}$，可以使用极大似然估计法来估计 $\boldsymbol{w}$。

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 $\boldsymbol{x}$ 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 $\boldsymbol{x}$ 为各类别的概率，如果通过训练确定了模型参数 $\boldsymbol{W}$，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 $\boldsymbol{W}$ 的更新公式。$\boldsymbol{W}$为矩阵，更新$\boldsymbol{W}$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
```

```python
        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
```

```

                loss_old = loss

            # 每一轮迭代之前，随机打乱训练集
            np.random.shuffle(idx)
            for i in idx:
                # 预测 xi 为各类别的概率
                yi_proba = self._predict_proba(X[i], W)
                # 计算梯度
                grad = self._gradient(X[i], y[i], yi_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
            k = np.unique(y_train).size
            _, n = X_train.shape
            self.W = np.random.random((k, n)) * 0.05

            # 执行随机梯度下降训练 W
```

```
            self._stochastic_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)

        # 对每个实例计算向量 z
        Z = self._z(X, self.W)

        # 以向量 z 中最大分量的索引作为预测的类别
        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., 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., 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., 3., 3.,
```

```
       3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3., 3.,
       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：

```
>>> idx = y != 3
>>> X = X[idx]
>>> 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],
       ...,
       [1.179e+01, 2.130e+00, 2.780e+00, ..., 9.700e-01, 2.440e+00,
        4.660e+02],
       [1.237e+01, 1.630e+00, 2.300e+00, ..., 8.900e-01, 2.780e+00,
        3.420e+02],
       [1.204e+01, 4.300e+00, 2.380e+00, ..., 7.900e-01, 2.570e+00,
        5.800e+02]])
>>> y = y[idx]
>>> 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., 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., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
```

```
       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：

```
>>> y -= 1
>>> y
array([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., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 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., 1., 1., 1.,
       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：

```
>>> from logistic_regression import LogisticRegression
>>> 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）：

```
>>> 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)
```

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],
```

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

接下来，训练模型：

```
>>> clf.train(X_train_std, y_train)
  0 Loss: 0.733072315368412
  1 Loss: 0.72438967420864
  2 Loss: 0.7158883537077279
  3 Loss: 0.7075647055742007
  4 Loss: 0.699415092738375
  5 Loss: 0.691435894577802
   ...
710 Loss: 0.12273875001717284
711 Loss: 0.12263819712826703
712 Loss: 0.12253785223988992
713 Loss: 0.1224377146473229
714 Loss: 0.1223377836492213
```

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。

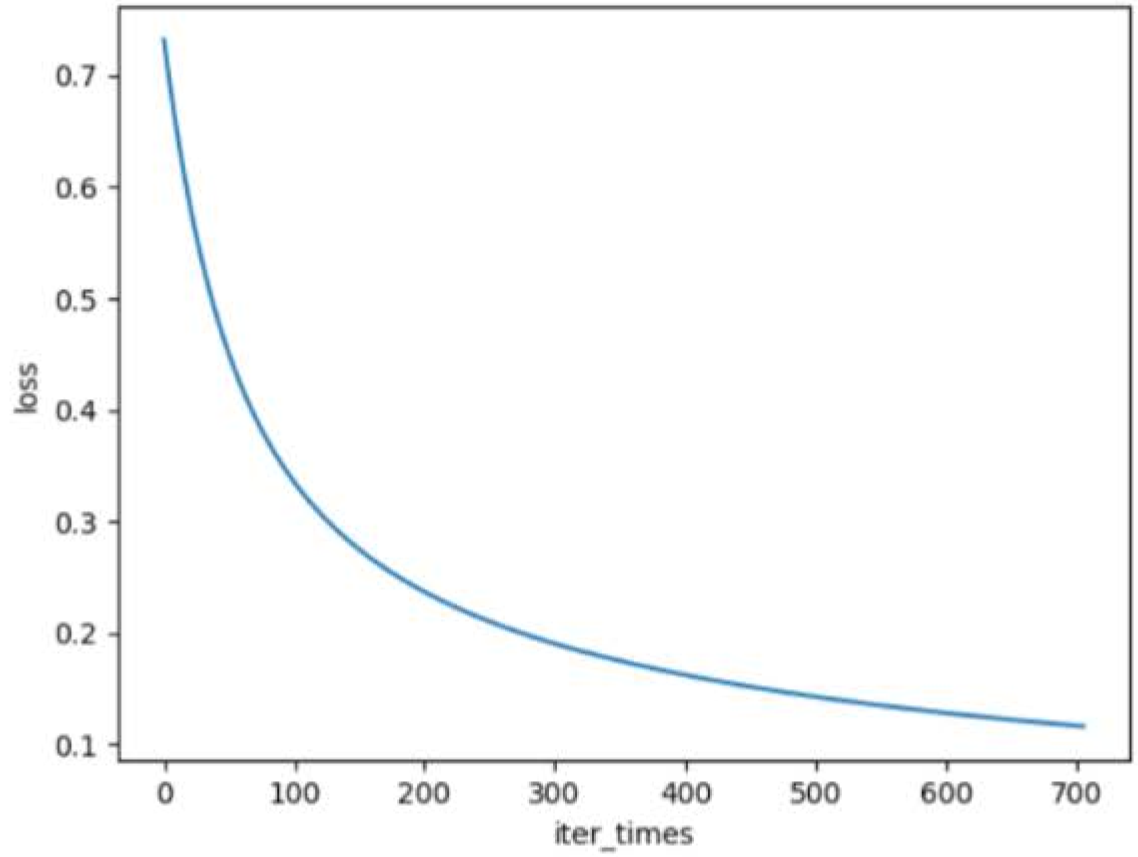

图 2-3

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)
```

```
...
...     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
0.9805128205128206
```

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：

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

```
>>> 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。