第3章

分类问题

3.1 K 近邻算法

K 近邻(K-nearest neighbor,KNN)算法于 1968 年由 Cover 和 Hart 提出,是机器学习算法中一种基本的分类与回归方法,以输入为实例的特征向量,通过计算新数据与训练数据特征值之间的距离,然后选取 $k(k \geqslant 1)$个距离中最近的邻居进行分类判断(投票法)或者回归。如果 $k=1$,那么新数据被简单地分配给其近邻的类。

- 分类问题。分类问题的输出为实例的类别。分类时,对于新的实例,根据其 k 个最近邻的训练实例的类别,通过多数表决规则进行预测。
- 回归问题。回归问题的输出为实例的值。回归时,对于新的实例,取其 k 个最近邻的训练实例的平均值为预测值。

K 近邻算法的直观理解是,给定一个训练数据集,对于新的输入实例,在训练集中找到与该实例最邻近的 k 个实例。这 k 个实例的多数属于某个类别,则该输入实例就划分为这个类别。

K 近邻算法不包含显式的学习过程,而是直接进行预测。实际上它是利用训练数据集对特征向量空间进行划分,作为其分类的“模型”。

3.1.1 原理简介

K 近邻算法的三要素分别是 k 值选择、距离度量和分类决策规则。

1. k 值选择

当 $k=1$ 时,K 近邻算法又称为最近邻算法,此时会将训练集中与 $\boldsymbol{x}$ 最近的点的类别作为 $\boldsymbol{x}$ 的分类。

k 值的选择会对 K 近邻算法的结果产生重大影响。

(1) 若 k 值选取得较小,就相当于用较小的邻域中的训练实例进行预测。

- 只有与输入实例较近的(相似的)训练实例才会对预测结果起作用。
- 学习的估计误差会增大,预测结果会对近邻的实例点非常敏感。若近邻的训练实例点刚好是噪声,则预测会出错。即 k 值的减小意味着模型整体变复杂,易发生过拟合。

（2）若 k 值选取得较大，就相当于用较大的领域中的训练实例进行预测。

■ 能减少学习的估计误差。

■ 学习的近似误差会增大。这时与输入实例较远的（不相似的）训练实例也会对预测结果起作用，使预测产生错误，即 k 值的增大意味着模型整体变简单。当 $k=N$ 时，无论输入实例是什么，都将其预测为训练实例中最多的类（即预测结果是一个常量），此时模型过于简单，完全忽略了训练实例中大量有用的信息。

应用中，k 值一般选取一个较小的数值，通常采用交叉验证法来选取最优的 k 值。即比较不同 k 值时的交叉验证平均误差率，选择误差率最小的那个 k 值。例如，选择 $k=1,2,3,\cdots$，对每个 $k=i$ 做若干次交叉验证，计算出平均误差率，然后比较并选出最小的那个。

2. 距离度量

K 近邻算法要求数据的所有特征都可以做可比较的量化。若在数据特征中存在非数值的类型，则必须采取手段将其量化为数值。例如，如果样本特征中包含颜色（红、黑、蓝）一项，颜色之间是没有距离可言的，可通过将颜色转换为灰度值来实现距离计算。另外，样本有多个参数，每个参数都有自己的定义域和取值范围，它们对距离计算的影响也就不一样，如取值较大的影响力会盖过取值较小的参数。为了公平，样本参数必须做一些归一化处理，最简单的方式就是所有特征的数值都进行归一化处置。

特征空间中两个实例点的距离是两个实例点相似程度的反映，K 近邻模型的特征空间一般是 n 维实数向量空间 $\mathbb{R}^n$。常使用的距离是欧氏距离，但有时也可以是其他距离，如曼哈顿距离。一般的距离公式表示为 L_p 距离（L_p distance）。

$$L_p(\boldsymbol{x}_i,\boldsymbol{x}_j)=\left(\sum_{l=1}^{n}|x_i^{(l)}-x_j^{(l)}|^p\right)^{\frac{1}{p}},\quad \begin{cases}\boldsymbol{x}_i,\boldsymbol{x}_j\in\boldsymbol{\chi}\subseteq\mathbb{R}^n\\ \boldsymbol{x}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)})^{\mathrm{T}}\\ \boldsymbol{x}_j=(x_j^{(1)},x_j^{(2)},\cdots,x_j^{(n)})^{\mathrm{T}}\\ p\geqslant 1\end{cases}\tag{3.1}$$

上式中，当 $p=1$ 时，为曼哈顿距离（Manhattan distance），即

$$L_1(\boldsymbol{x}_i,\boldsymbol{x}_j)=\sum_{l=1}^{n}|x_i^{(l)}-x_j^{(l)}|\tag{3.2}$$

当 $p=2$ 时，为欧氏距离（Euclidean distance），即

$$L_2(\boldsymbol{x}_i,\boldsymbol{x}_j)=\left(\sum_{l=1}^{n}|x_i^{(l)}-x_j^{(l)}|^2\right)^{\frac{1}{2}}\tag{3.3}$$

当 $p=\infty$ 时，为各维度距离中的最大值，即

$$L_\infty(\boldsymbol{x}_i,\boldsymbol{x}_j)=\max_l|x_i^{(l)}-x_j^{(l)}|\tag{3.4}$$

不同的距离度量所确定的最近邻点是不同的，需要根据具体应用场景确定使用哪一种度量方式。一般情况下，欧氏距离适用于连续变量，曼哈顿距离适用于计算棋盘格局相似性、机器人避障路径选择、城市道路长度计算等问题。

3. 分类决策规则

K 近邻算法的分类决策通常采用多数表决规则，即由输入实例的 k 个邻近的训练实例中的多数类决定输入实例的类。也可以基于距离的远近进行加权投票，距离越近的样本实

例权重越大。

多数表决规则等价于经验风险最小化，设分类的损失函数为0-1损失函数（指预测值与损失值不相等为1，否则为0），分类函数为 $f:\mathbb{R}^n \rightarrow \{c_1, c_2, \cdots, c_K\}$，误分类概率为

$$P(Y \neq f(X)) = 1 - P(Y = f(X)) \tag{3.5}$$

给定输入实例 $\boldsymbol{x} \in \boldsymbol{\chi}$，其最邻近的 k 个样本实例构成集合 $N_k(\boldsymbol{x})$。设涵盖 $N_k(\boldsymbol{x})$ 区域的类别为 c_j（是一个待求的未知量，但它肯定是 $c_1, c_2, \cdots, c_K$ 之一），则误分类概率为

$$\frac{1}{k}\sum_{\boldsymbol{x}_i \in N_k(\boldsymbol{x})} I(y_i \neq c_j) = 1 - \frac{1}{k}\sum_{\boldsymbol{x}_i \in N_k(\boldsymbol{x})} I(y_i = c_j), \quad \begin{cases} i = 1, 2, \cdots, N \\ j = 1, 2, \cdots, K \end{cases} \tag{3.6}$$

误分类概率就是训练数据的经验风险，要使误分类概率最小，即经验风险最小，就要使 $\sum_{\boldsymbol{x}_i \in N_k(\boldsymbol{x})} I(y_i = c_j)$ 最大。即

$$c_j = \underset{c_j}{\operatorname{argmax}} \sum_{\boldsymbol{x}_i \in N_k(\boldsymbol{x})} I(y_i = c_j), \quad \begin{cases} i = 1, 2, \cdots, N \\ j = 1, 2, \cdots, K \end{cases} \tag{3.7}$$

3.1.2　算法步骤

K 近邻算法的步骤如下。

（1）输入：训练数据集 $T = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \cdots, (\boldsymbol{x}_N, y_N)\}$，$\boldsymbol{x}_i \in \boldsymbol{\chi} \subseteq \mathbb{R}^n$ 为样本实例，$y_i \in \gamma = \{c_1, c_2, \cdots, c_K\}$ 为样本实例的类别，$i = 1, 2, \cdots, N$。给定输入实例 $\boldsymbol{x}(\boldsymbol{x} \in \boldsymbol{\chi})$。

（2）输出：输入实例 $\boldsymbol{x}$ 的类别 y。

（3）算法步骤。

① 根据选取的距离度量，在 T 中寻找与输入实例 $\boldsymbol{x}$ 最近邻的 k 个样本实例点 $\boldsymbol{x}_i$。涵盖这 k 个样本实例点的邻域记作 $N_k(\boldsymbol{x})$。

② 从 $N_k(\boldsymbol{x})$ 中，根据分类决策规则（如多数表决规则）决定输入实例 $\boldsymbol{x}$ 的类别 y：

$$y = \underset{c_j}{\operatorname{argmax}} \sum_{\boldsymbol{x}_i \in N_k(\boldsymbol{x})} I(y_i = c_j), \quad \begin{cases} i = 1, 2, \cdots, N \\ j = 1, 2, \cdots, K \end{cases} \tag{3.8}$$

其中，$I()$ 为指示函数，$I(\text{true}) = 1$，$I(\text{false}) = 0$。上式中，对于 $y_i, i = 1, 2, \cdots, N$，只有 $\boldsymbol{x} \in N_k(\boldsymbol{x})$ 中的样本点时才考虑。

3.1.3　实战

1. 数据集

首先导入包：

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import neighbors, datasets, model_selection
```

然后给出加载数据集的函数：

```
def load_classification_data():
  '''
  加载分类模型使用的数据集。
```

```
    :return: 一个元组,依次为训练样本集、测试样本集、训练样本的标记、测试样本的标记
    '''
    # 使用 scikit - learn 自带的手写识别数据集 Digit Dataset
    digits = datasets.load_digits()
    X_train = digits.data
    y_train = digits.target
    # 进行分层采样拆分,测试集大小占 1/4
    return model_selection.train_test_split(X_train, y_train,test_size = 0.25,
        random_state = 0,stratify = y_train)
```

其中,load_classification_data()函数使用的是 scikit-learn 自带的手写识别数据集 Digit Dataset。该数据集由 1797 张样本图片组成。每张样本图片都是一个 8×8 像素的手写数字位图。

2. Sklearn 实现

1)K 近邻分类

scikit-learn 中提供了一个 KNeighborsClassifier 类来实现 K 近邻算法的分类模型,其原型如下:

```
class sklearn.neighbors.KNeighborsClassifier(n_neighbors = 5, *, weights = 'uniform',
algorithm = 'auto', leaf_size = 30, p = 2, metric = 'minkowski', metric_params = None, n_jobs =
None, ** kwargs)
```

(1)参数。

① n_neighbors:一个整数,用于指定 k 值。

② weights:一个字符串或者可调用对象,用于指定投票权重类型。即这些邻居投票权可以相同或者不同。可以有以下取值。

■ 'uniform':本节点的所有邻居节点的投票权重都相等。

■ 'distance':本节点的所有邻居节点的投票权重与距离成反比。即越近的节点,其投票权重越大。

■ [callable]:一个可调用对象。它传入距离的数组,返回同样形状的权重数组。

③ algorithm:一个字符串,用于指定计算最近邻的算法,可以为如下取值。

■ 'ball_tree':使用 BallTree 算法。

■ 'kd_tree':使用 KDTree 算法。

■ 'brute':使用暴力搜索法。

■ 'auto':自动决定最合适的算法。

④ leaf_size:一个整数,用于指定 BallTree 或者 KDTree 叶节点规模。它影响树的构建和查询速度。

⑤ metric:一个字符串,用于指定距离度量。默认为'minkowski'距离。

⑥ p:整数值,用于指定在'minkowski'度量上的指数。如果 p=1,对应曼哈顿距离;如果 p=2,对应欧氏距离。

⑦ metric_params:度量功能的其他关键字参数。字典类型,默认参数值为 None。

⑧ n_jobs:用于指定并行性。默认为-1,表示派发任务到所有计算机的 CPU 上。

(2)方法。

① fit(X,y):训练模型。

② predict(X)：使用模型来预测，返回待预测样本的标记。

③ score(X,y)：返回在(X,y)上预测的准确率(accuracy)。

④ predict_proba(X)：返回样本为每种标记的概率。

⑤ kneighbors([X，n_neighbors，return_distance])：返回样本点的 k 个近邻点。如果 return_distance＝True，同时还返回这些近邻点的距离。

⑥ kneighbors_graph([X，n_neighbors，mode])：返回样本点的连接图。

首先使用 KNeighborsClassifier，给出如下测试函数：

```
def test_KNeighborsClassifier( * data):
    '''
    测试 KNeighborsClassifier 的用法

    :param data: 可变参数.它是一个元组,这里要求其元素依次为训练样本集、测试样本集、训练样本
的标记、测试样本的标记
    :return: None
    '''
    X_train,X_test,y_train,y_test = data
    clf = neighbors.KNeighborsClassifier()
    clf.fit(X_train,y_train)
    print("Training Score: % f" % clf.score(X_train,y_train))
    print("Testing Score: % f" % clf.score(X_test,y_test))
```

然后调用 test_KNeighborsClassifier()函数：

```
X_train,X_test,y_train,y_test = load_classification_data() # 获取分类模型的数据集
test_KNeighborsClassifier(X_train,X_test,y_train,y_test)  #调用 test_KNeighborsClassifier
#输出
Training Score:0.991091
Testing Score:0.980000
```

可以看出，K 近邻算法对测试集的数据预测准确率高达 98.0000％，对训练集的拟合准确率高达 99.1091％。

然后考察 k 值以及投票策略对预测性能的影响，给出测试函数：

```
def test_KNeighborsClassifier_k_w( * data):
    '''
    测试 KNeighborsClassifier 中 n_neighbors 和 weights 参数的影响

    :param data: 可变参数.它是一个元组,这里要求其元素依次为训练样本集、测试样本集、训练样本
的标记、测试样本的标记
    :return: None
    '''
    X_train,X_test,y_train,y_test = data
    Ks = np.linspace(1,y_train.size,num = 100,endpoint = False,dtype = 'int')
    weights = ['uniform','distance']

    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    ### 绘制不同 weights 下, 预测得分随 n_neighbors 变化的曲线
    for weight in weights:
```

```
        training_scores = [ ]
        testing_scores = [ ]
        for K in Ks:
            clf = neighbors.KNeighborsClassifier(weights = weight,n_neighbors = K)
            clf.fit(X_train,y_train)
            testing_scores.append(clf.score(X_test,y_test))
            training_scores.append(clf.score(X_train,y_train))
    ax.plot(Ks,testing_scores,label = "testing score:weight = % s" % weight)
    ax.plot(Ks,training_scores,label = "training score:weight = % s" % weight)
    ax.legend(loc = 'best')
    ax.set_xlabel("k")
    ax.set_ylabel("score")
    ax.set_ylim(0,1.05)
    ax.set_title("KNeighborsClassifier")
    plt.show()
```

同样调用 test_KNeighborsClassifier_k_w()函数,运行结果如图 3.1 所示。

```
test_KNeighborsClassifier_k_w(X_train,X_test,y_train,y_test)
```

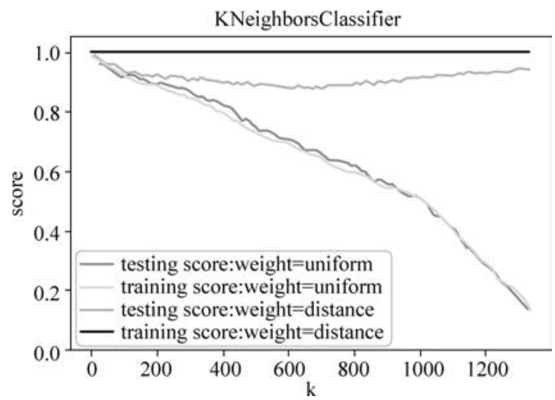


图 3.1　调用 test_KNeighborsClassifier_k_w()函数的运行结果

从图 3.1 中可以看出,在使用 uniform 投票策略的情况下(即投票权重都相同),分类器随着 k 的增长,预测性能稳定下降。这是因为当 k 增大时,距输入实例较远的训练实例也会对预测起作用,使预测发生错误。

在使用 distance 投票策略的情况下(即投票权重与距离成反比),分类器随着 k 的增长,对测试集的预测性能相对比较稳定。这是因为虽然 k 增大时,距输入实例较远的训练实例也会对预测起作用,但因为距离较远,其影响小得多(权重很小)。

然后考察 p 值(即距离函数的形式)对预测性能的影响,给出测试函数:

```
def test_KNeighborsClassifier_k_p( * data):
    '''
    测试 KNeighborsClassifier 中 n_neighbors 和 p 参数的影响

    :param data: 可变参数.它是一个元组,这里要求其元素依次为训练样本集、测试样本集、训练样本的标记、测试样本的标记
    :return: None
```

```
'''
X_train,X_test,y_train,y_test = data
Ks = np.linspace(1,y_train.size,endpoint = False,dtype = 'int')
Ps = [1,2,10]

fig = plt.figure()
ax = fig.add_subplot(1,1,1)
### 绘制不同 p 值下，预测得分随 n_neighbors 变化的曲线
for P in Ps:
    training_scores = []
    testing_scores = []
    for K in Ks:
        clf = neighbors.KNeighborsClassifier(p = P,n_neighbors = K)
        clf.fit(X_train,y_train)
        testing_scores.append(clf.score(X_test,y_test))
        training_scores.append(clf.score(X_train,y_train))
    ax.plot(Ks,testing_scores,label = "testing score:p = %d" % P)
    ax.plot(Ks,training_scores,label = "training score:p = %d" % P)
ax.legend(loc = 'best')
ax.set_xlabel("k")
ax.set_ylabel("score")
ax.set_ylim(0,1.05)
ax.set_title("KNeighborsClassifier")
plt.show()
```

同样地调用 test_KNeighborsClassifier_k_p()函数，运行结果如图 3.2 所示。

```
test_KNeighborsClassifier_k_p(X_train,X_test,y_train,y_test)
```

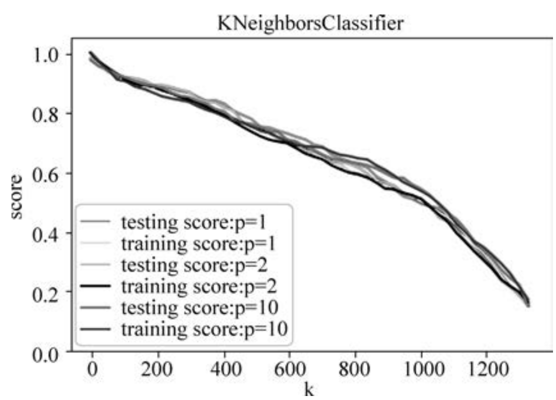


图 3.2 调用 test_KNeighborsClassifier_k_p()函数的运行结果

可以看到，参数 p 对分类器的预测性能没有任何影响。由于有

$$L_p(\boldsymbol{x}_i,\boldsymbol{x}_j)=\left(\sum_{l=1}^{n}|x_i^{(l)}-x_j^{(l)}|^p\right)^{\frac{1}{p}}$$

因此，当 $p=1$ 时，$\boldsymbol{x}_j$ 是 $\boldsymbol{x}_i$ 的最近的点；当 p 为其他值时，该结论也成立。

2）K 近邻回归

scikit-learn 中提供了一个 KNeighborsRegressor 类来实现 K 近邻算法的回归模型，其原型如下：

```
class sklearn.neighbors.KNeighborsRegressor(n_neighbors = 5, *, weights = 'uniform',
algorithm = 'auto', leaf_size = 30, p = 2, metric = 'minkowski', metric_params = None, n_jobs =
None)
```

（1）参数。

① n_neighbors：一个整数，用于指定 k 值。

② weights：一个字符串或者可调用对象，用于指定投票权重类型。即这些邻居投票权可以相同或者不同。可以有以下取值。

- 'uniform'：本节点所有邻居节点的投票权重都相等。
- 'distance'：本节点的所有邻居节点的投票权重与距离成反比。即越近的节点，其投票权重越大。
- [callable]：一个可调用对象。它传入距离的数组，返回同样形状的权重数组。

③ algorithm：一个字符串，用于指定计算最近邻的算法，可以有以下取值。

- 'ball_tree'：使用 BallTree 算法。
- 'kd_tree'：使用 KDTree 算法。
- 'brute'：使用暴力搜索法。
- 'auto'：自动决定最合适的算法。

④ leaf_size：一个整数，用于指定 BallTree 或者 KDTree 叶节点规模。它影响树的构建和查询速度。

⑤ metric：一个字符串，用于指定距离度量。默认为'minkowski'距离。

⑥ p：一个整数值，用于指定在'minkowski'度量上的指数。如果 p=1，对应曼哈顿距离；如果 p=2，对应欧氏距离。

⑦ n_jobs：一个整数，用于指定并行性。默认为−1，表示派发任务到所有计算机的 CPU 上。

（2）方法。

① fit(X,y)：训练模型。

② predict(X)：使用模型来预测，返回待预测样本的标记。

③ score(X,y)：返回预测性能得分。设预测集为 T_{test}，真实值为 y_i，真实值的均值为 $\bar{y}$，预测值为 $\hat{y}$，则有

$$\text{score}=1-\frac{\sum_{T_{\text{test}}}(y_i-\hat{y})^2}{(y_i-\bar{y})^2} \tag{3.9}$$

- score 不超过 1，但是可能为负值（预测效果太差）。
- score 越大，预测性能越好。

④ kneighbors([X,n_neighbors,return_distance])：返回样本点的 k 个近邻点。如果 return.distance=True，则同时还返回该样本到这些近邻点的距离值。

⑤ kneighbors_graph([X,n_neighbors,mode])：返回样本点的连接图。其参数意义以及实例方法与 KNeighborsClassifier 几乎完全相同。两者区别在于回归分析分类决策的不同。

- KNeighborsClassifier 将待预测样本点最近邻的 k 个训练样本点中出现次数最多的

分类作为待预测样本点的分类。

■ KNeighborsRegressor 将待预测样本点最近邻的 k 个训练样本点的平均值作为待预测样本点的值。

首先使用 KNeighborsRegressor，给出测试函数：

```
def test_KNeighborsRegressor( * data):
  '''
  测试 KNeighborsRegressor 的用法

  :param data: 可变参数.它是一个元组,这里要求其元素依次为训练样本集、测试样本集、训练样本
的值、测试样本的值
  :return: None
  '''
  X_train, X_test, y_train, y_test = data
  regr = neighbors.KNeighborsRegressor()
  regr.fit(X_train, y_train)
  print("Training Score: %f" % regr.score(X_train, y_train))
print("Testing Score: %f" % regr.score(X_test, y_test))
```

然后调用 test_KNeighborsRegressor()函数：

```
X_train, X_test, y_train, y_test = create_regression_data(1000) # 获取回归模型的数据集
test_KNeighborsRegressor(X_train, X_test, y_train, y_test) # 调用 test_KNeighborsRegressor()
```

这里生成了 1000 个样本数据。结果如下：

```
Training Score:0.972194
Testing Score:0.969817
```

可以看到，回归器对于测试集的预测得分为 0.969817，对于训练集的预测得分为 0.972194。

然后考察 k 值以及投票策略对预测性能的影响，给出如下测试函数：

```
def test_KNeighborsRegressor_k_w( * data):
  '''
  测试 KNeighborsRegressor 中 n_neighbors 和 weights 参数的影响
  :param data: 可变参数。它是一个元组,这里要求其元素依次为训练样本集、测试样本集、训练样
本的值、测试样本的值
  :return: None
  '''
  X_train, X_test, y_train, y_test = data
  Ks = np.linspace(1, y_train.size, num=100, endpoint=False, dtype='int')
  weights = ['uniform', 'distance']

  fig = plt.figure()
  ax = fig.add_subplot(1, 1, 1)
  ### 绘制不同 weights 下, 预测得分随 n_neighbors 变化的曲线
  for weight in weights:
     training_scores = []
     testing_scores = []
     for K in Ks:
```

```
            regr = neighbors.KNeighborsRegressor(weights = weight, n_neighbors = K)
            regr.fit(X_train, y_train)
            testing_scores.append(regr.score(X_test, y_test))
            training_scores.append(regr.score(X_train, y_train))
        ax.plot(Ks, testing_scores, label = "testing score:weight = %s" % weight)
        ax.plot(Ks, training_scores, label = "training score:weight = %s" % weight)
    ax.legend(loc = 'best')
    ax.set_xlabel("k")
    ax.set_ylabel("score")
    ax.set_ylim(0, 1.05)
    ax.set_title("KNeighborsRegressor")
plt.show()
```

调用 test_KNeighborsRegressor_k_w()函数：

```
X_train, X_test, y_train, y_test = create_regression_data(1000)  # 获取回归模型的数据集
test_KNeighborsRegressor_k_w(X_train,X_test,y_train,y_test)
                                              # 调用 test_KNeighborsRegressor_k_w
```

运行结果如图 3.3 所示。

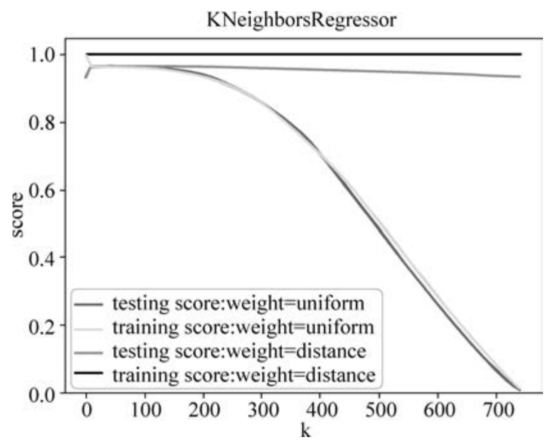


图 3.3 调用 test_KNeighborsRegressor_k_w()函数的运行结果

然后考察 p 值(即距离函数的形式)对预测性能的影响,给出如下测试函数：

```
def test_KNeighborsRegressor_k_p(*data):
    '''
    测试 KNeighborsRegressor 中 n_neighbors 和 p 参数的影响

    :param data: 可变参数。它是一个元组,这里要求其元素依次为训练样本集、测试样本集、训练样
本的值、测试样本的值
    :return: None
    '''
    X_train, X_test, y_train, y_test = data
    Ks = np.linspace(1, y_train.size, endpoint = False, dtype = 'int')
    Ps = [1, 2, 10]

    fig = plt.figure()
```

```
    ax = fig.add_subplot(1, 1, 1)
    ### 绘制不同 p 值下，预测得分随 n_neighbors 变化的曲线
    for P in Ps:
        training_scores = []
        testing_scores = []
        for K in Ks:
            regr = neighbors.KNeighborsRegressor(p = P, n_neighbors = K)
            regr.fit(X_train, y_train)
            testing_scores.append(regr.score(X_test, y_test))
            training_scores.append(regr.score(X_train, y_train))
        ax.plot(Ks, testing_scores, label = "testing score:p = %d" % P)
        ax.plot(Ks, training_scores, label = "training score:p = %d" % P)
    ax.legend(loc = 'best')
    ax.set_xlabel("k")
    ax.set_ylabel("score")
    ax.set_ylim(0, 1.05)
    ax.set_title("KNeighborsRegressor")
plt.show()
```

调用 test_KNeighborsRegressor_k_p()函数，运行结果如图 3.4 所示。

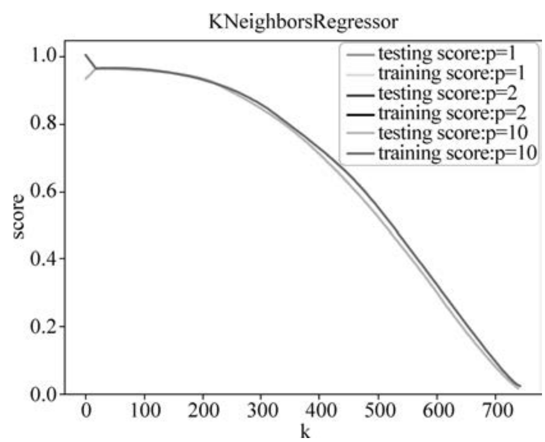


图 3.4　调用 test_KNeighborsRegressor_k_p()函数的运行结果

3. 算法实现

首先，数据集选择经典的 Iris 鸢尾花数据集。该数据集的详细介绍见 2.1.4 节。Iris 数据集整体构成一个 150 行、5 列的二维表，表 3.1 展示了其中的 5 个样本。

表 3.1　Iris 数据集示例

序号	sepal length	sepal width	petal length	petal width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5	3.6	1.4	0.2	Iris-setosa

下载后的数据集存放在./data/目录下。代码如下：

```
import numpy as np
import pandas as pd

data = pd.read_csv('./data/iris.data', header = None)
data.columns = ['sepal length', 'sepal width', 'petal length', 'petal width', 'species']
```

将 3 个类别的数据分别提取出来，setosa、versicolour、virginica 分别用 0、1、2 来表示。代码如下：

```
X = data.iloc[0:150, 0:4].values
y = data.iloc[0:150, 4].values
y[y == 'Iris-setosa'] = 0                           # Iris-setosa 输出 label 用 0 表示
y[y == 'Iris-versicolour'] = 1                      # Iris-versicolour 输出 label 用 1 表示
y[y == 'Iris-virginica'] = 2                        # Iris-virginica 输出 label 用 2 表示
X_setosa, y_setosa = X[0:50], y[0:50]               # Iris-setosa 的 4 个特征
X_versicolour, y_versicolour = X[50:100], y[50:100] # Iris-versicolour 的 4 个特征
X_virginica, y_virginica = X[100:150], y[100:150]   # Iris-virginica 的 4 个特征
```

接下来看一下 3 个类别不同特征的空间分布。为了可视性，选择 sepal length 和 petal length 两个特征，在二维平面上作图。代码如下：

```
import matplotlib.pyplot as plt

plt.scatter(X_setosa[:, 0], X_setosa[:, 2], colour = 'red', marker = 'o', label = 'setosa')
plt.scatter(X_versicolour[:, 0], X_versicolour[:, 2], colour = 'blue', marker = '^', label =
'versicolour')
plt.scatter(X_virginica[:, 0], X_virginica[:, 2], colour = 'green', marker = 's', label =
'virginica')
plt.xlabel('sepal length')
plt.ylabel('petal length')
plt.legend(loc = 'upper left')
plt.show()
```

运行结果如图 3.5 所示。

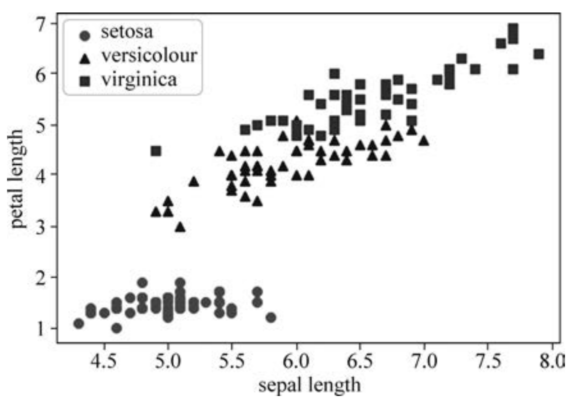


图 3.5　3 个类别在不同特征下的空间分布

由图 3.5 可见，3 个类别之间有较明显的区别。

接下来将每个类别的所有样本分成训练集(training set)、验证集(validation set)和测

试集(test set),各占所有样本的比例分别为 60%、20%、20%。代码如下:

```
# 训练集
X_setosa_train = X_setosa[:30, :]
y_setosa_train = y_setosa[:30]
X_versicolour_train = X_versicolour[:30, :]
y_versicolour_train = y_versicolour[:30]
X_virginica_train = X_virginica[:30, :]
y_virginica_train = y_virginica[:30]
X_train = np.vstack([X_setosa_train, X_versicolour_train, X_virginica_train])
y_train = np.hstack([y_setosa_train, y_versicolour_train, y_virginica_train])

# 验证集
X_setosa_val = X_setosa[30:40, :]
y_setosa_val = y_setosa[30:40]
X_versicolour_val = X_versicolour[30:40, :]
y_versicolour_val = y_versicolour[30:40]
X_virginica_val = X_virginica[30:40, :]
y_virginica_val = y_virginica[30:40]
X_val = np.vstack([X_setosa_val, X_versicolour_val, X_virginica_val])
y_val = np.hstack([y_setosa_val, y_versicolour_val, y_virginica_val])

# 测试集
X_setosa_test = X_setosa[40:50, :]
y_setosa_test = y_setosa[40:50]
X_versicolour_test = X_versicolour[40:50, :]
y_versicolour_test = y_versicolour[40:50]
X_virginica_test = X_virginica[40:50, :]
y_virginica_test = y_virginica[40:50]
X_test = np.vstack([X_setosa_test, X_versicolour_test, X_virginica_test])
y_test = np.hstack([y_setosa_test, y_versicolour_test, y_virginica_test])
class KNearestNeighbor(object):
  def __init__(self):
     pass

  # 训练函数
  def train(self, X, y):
     self.X_train = X
     self.y_train = y

  # 预测函数
  def predict(self, X, k=1):
     # 计算 L2 距离
     num_test = X.shape[0]
     num_train = self.X_train.shape[0]
     dists = np.zeros((num_test, num_train))          # 初始化距离函数
     # because(X - X_train)*(X - X_train) = -2X*X_train + X*X + X_train*X_train, so
     d1 = -2 * np.dot(X, self.X_train.T)              # shape (num_test, num_train)
     d2 = np.sum(np.square(X), axis=1, keepdims=True)# shape (num_test, 1)
     d3 = np.sum(np.square(self.X_train), axis=1)     # shape (1, num_train)
     dist = np.sqrt(d1 + d2 + d3)
     # 根据 k 值,选择最可能属于的类别
     y_pred = np.zeros(num_test)
```

```
    for i in range(num_test):
        dist_k_min = np.argsort(dist[i])[:k]      # 最近邻 k 个实例位置
        y_kclose = self.y_train[dist_k_min]       # 最近邻 k 个实例对应的标签
        y_pred[i] = np.argmax(np.bincount(y_kclose.tolist()))   # 找出 k 个标签中从属类别
#最多的作为预测类别

    return y_pred
```

创建一个 KnearestNeighbor 实例对象，然后，在验证集上进行 k-fold 交叉验证。选择不同的 k 值，根据验证结果，选择最佳的 k 值。通过实验发现，k 值取 3 的时候，验证集的准确率最高。此例中，由于总体样本数据量不够多，所以验证结果并不明显。但是使用 k-fold 交叉验证来选择最佳 k 值是最常用的方法之一。选择完合适的 k 值之后，就可以对测试集进行预测分析了。代码如下：

```
KNN = KNearestNeighbor()
KNN.train(X_train, y_train)
y_pred = KNN.predict(X_test, k = 3)
accuracy = np.mean(y_pred == y_test)
print('测试集预测准确率: %f' % accuracy)
#输出
测试集预测准确率:1.000000
```

最终结果显示，测试集预测准确率为 100%。

最后，将预测结果绘图表示。仍然只选择 sepal length 和 petal length 两个特征，在二维平面上作图。代码如下：

```
# 训练集
plt.scatter(X_setosa_train[:, 0], X_setosa_train[:, 2], colour = 'red', marker = 'o', label =
'setosa_train')
plt.scatter(X_versicolour_train[:, 0], X_versicolour_train[:, 2], colour = 'blue', marker = '^',
label = 'versicolour_train')
plt.scatter(X_virginica_train[:, 0], X_virginica_train[:, 2], colour = 'green', marker = 's',
label = 'virginica_train')
# 测试集
plt.scatter(X_setosa_test[:, 0], X_setosa_test[:, 2], colour = 'y', marker = 'o', label =
'setosa_test')
plt.scatter(X_versicolour_test[:, 0], X_versicolour_test[:, 2], colour = 'y', marker = '^',
label = 'versicolour_test')
plt.scatter(X_virginica_test[:, 0], X_virginica_test[:, 2], colour = 'y', marker = 's', label =
'virginica_test')

plt.xlabel('sepal length')
plt.ylabel('petal length')
plt.legend(loc = 4)
plt.show()
```

运行如上代码后，得到的预测结果如图 3.6 所示。

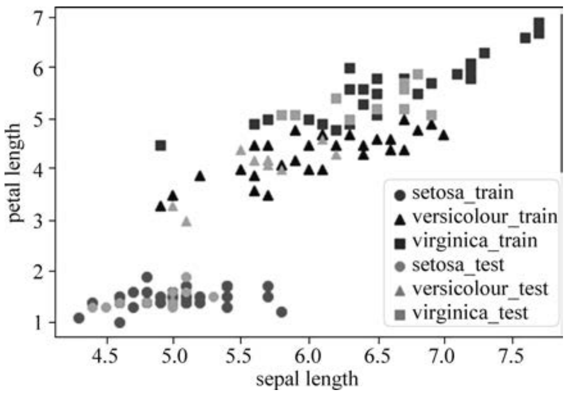


图 3.6 预测结果

3.1.4 实验

1. 实验目的

理解 K 近邻分类算法的原理,掌握 k 值选择、距离度量和分类决策的设置方法,并分别利用 scikit-learn 的相关包、Python 语言编程来实现该算法。

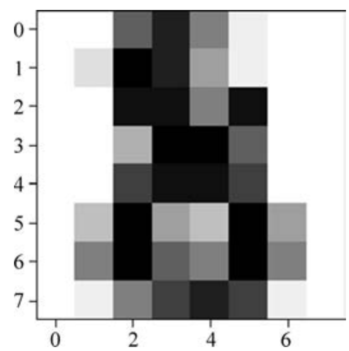


图 3.7 手写数字位图

2. 实验数据

实验数据为 scikit-learn 自带的手写识别数据集 Digit Dataset。该数据集由 1797 张样本图片组成。每张样本图片都是一个 8×8 像素大小的手写数字位图,如图 3.7 所示。为了使用这样的 8×8 像素图形,必须首先将其转换为长度为 64 的特征向量。

实现图 3.7 的代码如下:

```
# - * - coding: UTF - 8 - * -
"""
============================================================
手写数字数据集
============================================================
from sklearn import datasets

import matplotlib.pyplot as plt

# 导入手写数字数据集
digits = datasets.load_digits()

# 显示最后一个手写数字位图
plt.figure(1, figsize = (3, 3))
plt.imshow(digits.images[ - 1], cmap = plt.cm.gray_r, interpolation = "nearest")
plt.show()
```

3. 实验要求

利用 Python 程序设计语言编程实现如下功能。

（1）数据预处理：主要做好加载数据、交叉验证、归一化等功能，以实现数据预处理。

（2）模型训练：利用 Python 来实现训练算法。

（3）模型验证：编程实现对算法的验证，以评估算法的有效性。

3.2 逻辑回归算法

3.2.1 原理简介

逻辑回归是一种常见的广义线性模型，线性模型中的“线性”就是一系列一次特征的线性组合，在二维空间中是一条直线，在三维空间中是一个平面，如果推广到 n 维空间，就可以理解为广义线性模型。线性模型(linear model)的形式为

$$f(\boldsymbol{x})=\boldsymbol{w}\cdot\boldsymbol{x}+b \tag{3.10}$$

其中，$\boldsymbol{x}=(x^{(1)},x^{(2)},\cdots,x^{(n)})^{\mathrm{T}}$ 是用列向量表示的样本，该样本有 n 种特征，用 $\boldsymbol{x}^{(i)}$ 表示样本 $\boldsymbol{x}$ 的第 i 个特征。$\boldsymbol{w}=(w^{(1)},w^{(2)},\cdots,w^{(n)})^{\mathrm{T}}$ 为每个特征对应的权重生成的权重向量，权重向量直观地表达了各个特征在预测中的重要性。

1. 普通线性回归

线性回归是一种回归分析技术，回归分析本质上就是一个函数估计的问题(函数估计包括参数估计和非参数估计两类)，就是找出因变量和自变量之间的因果关系。回归分析的因变量应该是连续变量，若因变量为离散变量，则问题转换为分类问题，回归分析是一个有监督学习的问题。

给定数据集 $\boldsymbol{T}=\{(\boldsymbol{x}_1,y_1),(\boldsymbol{x}_2,y_2),\cdots,(\boldsymbol{x}_N,y_N)\}$，$\boldsymbol{x}_i\in\boldsymbol{\chi}\subseteq\mathbb{R}^n$，$y_i\in Y\subseteq\mathbb{R}$，$i=1,2,\cdots,N$，其中 $\boldsymbol{x}_i=[x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)}]^{\mathrm{T}}$，学习的模型为

$$f(\boldsymbol{x})=\boldsymbol{w}\cdot\boldsymbol{x}+b \tag{3.11}$$

下面根据已知的数据集 $\boldsymbol{T}$ 来计算参数 $\boldsymbol{w}$ 和 b。

对于给定的样本 $\boldsymbol{x}_i$，其预测值为 $\hat{y}_i=f(\boldsymbol{x}_i)=\boldsymbol{w}\cdot\boldsymbol{x}_i+b$。采用平方损失函数，则在训练集 $\boldsymbol{T}$ 上，模型的损失函数为

$$L(f)=\sum_{i=1}^{N}(\hat{y}_i-y_i)^2=\sum_{i=1}^{N}(\boldsymbol{w}\cdot\boldsymbol{x}_i+b-y_i)^2 \tag{3.12}$$

要使损失函数最小，即

$$(\boldsymbol{w}^*,b^*)=\underset{\boldsymbol{w},b}{\operatorname{argmin}}\sum_{i=1}^{N}(\boldsymbol{w}\cdot\boldsymbol{x}_i+b-y_i)^2 \tag{3.13}$$

可以用梯度下降法来求解上述最优化问题的数值解，同时要对特征进行归一化处理，并利用最小二乘法来求解解析解。

令

$$\tilde{\boldsymbol{w}}=(w^{(1)},w^{(2)},\cdots,w^{(n)},b)^{\mathrm{T}}=(\boldsymbol{w}^{\mathrm{T}},b)^{\mathrm{T}}$$
$$\tilde{\boldsymbol{x}}=(x^{(1)},x^{(2)},\cdots,x^{(n)},1)^{\mathrm{T}}=(\boldsymbol{x}^{\mathrm{T}},1)^{\mathrm{T}}$$
$$\boldsymbol{y}=(y_1,y_2,\cdots,y_N)^{\mathrm{T}}$$

则有

$$\sum_{i=1}^{N}(\boldsymbol{w}\cdot\boldsymbol{x}_i+b-y_i)^2=(\boldsymbol{y}-(\tilde{\boldsymbol{x}}_1,\tilde{\boldsymbol{x}}_2,\cdots,\tilde{\boldsymbol{x}}_N)^{\mathrm{T}}\tilde{\boldsymbol{w}})^{\mathrm{T}}(\boldsymbol{y}-(\tilde{\boldsymbol{x}}_1,\tilde{\boldsymbol{x}}_2,\cdots,\tilde{\boldsymbol{x}}_N)^{\mathrm{T}}\tilde{\boldsymbol{w}}) \tag{3.14}$$

令

$$\boldsymbol{x}=(\tilde{\boldsymbol{x}}_1,\tilde{\boldsymbol{x}}_2,\cdots,\tilde{\boldsymbol{x}}_N)^{\mathrm{T}}=\begin{bmatrix}\tilde{\boldsymbol{x}}_1^{\mathrm{T}}\\ \tilde{\boldsymbol{x}}_2^{\mathrm{T}}\\ \vdots\\ \tilde{\boldsymbol{x}}_N^{\mathrm{T}}\end{bmatrix}=\begin{bmatrix}x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(n)} & 1\\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(n)} & 1\\ \vdots & \vdots & \ddots & \vdots & 1\\ x_N^{(1)} & x_N^{(2)} & \cdots & x_N^{(n)} & 1\end{bmatrix}$$

则有

$$\tilde{\boldsymbol{w}}^*=\underset{\tilde{\boldsymbol{w}}}{\operatorname{argmin}}(\boldsymbol{y}-\boldsymbol{x}\tilde{\boldsymbol{w}})^{\mathrm{T}}(\boldsymbol{y}-\boldsymbol{x}\tilde{\boldsymbol{w}}) \tag{3.15}$$

令 $E_{\tilde{\boldsymbol{w}}}=(\boldsymbol{y}-\boldsymbol{x}\tilde{\boldsymbol{w}})^{\mathrm{T}}(\boldsymbol{y}-\boldsymbol{x}\tilde{\boldsymbol{w}})$，求它的极小值。对 $\tilde{\boldsymbol{w}}$ 求导，令导数为零，得到如下解析解：

$$\frac{\partial E_{\tilde{\boldsymbol{w}}}}{\partial\tilde{\boldsymbol{w}}}=2\boldsymbol{x}^{\mathrm{T}}(\boldsymbol{x}\tilde{\boldsymbol{w}}-\boldsymbol{y})=\boldsymbol{0}\Rightarrow\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}\tilde{\boldsymbol{w}}=\boldsymbol{x}^{\mathrm{T}}\boldsymbol{y} \tag{3.16}$$

(1) 当 $\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}$ 为满秩矩阵或者正定矩阵时，可得

$$\tilde{\boldsymbol{w}}^*=(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x})^{-1}\boldsymbol{x}^{\mathrm{T}}\boldsymbol{y} \tag{3.17}$$

其中，$(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x})^{-1}$ 为 $\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}$ 的逆矩阵。于是得到多元线性回归模型为

$$f(\tilde{\boldsymbol{x}}_i)=\tilde{\boldsymbol{x}}_i^{\mathrm{T}}\tilde{\boldsymbol{w}}^* \tag{3.18}$$

(2) 当 $\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}$ 不是满秩矩阵时，比如 $N<n$(样本数量小于特征种类的数量)，根据 $\boldsymbol{x}$ 的秩小于或等于(N,n)中的最小值，即小于或等于 N(矩阵的秩一定小于或等于矩阵的行数和列数)；而矩阵 $\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}$ 是 $n\times n$ 大小的，它的秩一定小于或等于 N，因此不是满秩矩阵。此时存在多个解析解。常见的做法是引入正则化项，如 L_1 正则化或者 L_2 正则化。以 L_2 正则化为例：

$$\tilde{\boldsymbol{w}}^*=\underset{\tilde{\boldsymbol{w}}}{\operatorname{argmin}}[(\boldsymbol{y}-\boldsymbol{x}\tilde{\boldsymbol{w}})^{\mathrm{T}}(\boldsymbol{y}-\boldsymbol{x}\tilde{\boldsymbol{w}})+\lambda\|\tilde{\boldsymbol{w}}\|_2^2] \tag{3.19}$$

其中，$\lambda>0$ 时调整正则化项与均方误差的比例，$\|\cdots\|_2$ 为 L_2 范数。

2. 广义线性模型

考虑单调可导函数 $h(\cdot)$，令 $h(y)=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}+b$，这样得到的模型称为广义线性模型(generalized linear model)。广义线性模型的一个典型的例子就是对数线性回归。当 $h(\cdot)=\ln(\cdot)$时的广义线性模型就是对数线性回归，即

$$\ln y=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}+b \tag{3.20}$$

它是通过 $\exp(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}+b)$来拟合 y 的。它虽然被称为广义线性回归，但实质上是非线性的。

3. 逻辑回归

前述的学习方法都是使用线性模型进行回归学习的，而线性模型也可以用于分类。考虑二分类问题，给定数据集 $\boldsymbol{T}=\{(\boldsymbol{x}_1,y_1),(\boldsymbol{x}_2,y_2),\cdots,(\boldsymbol{x}_N,y_N)\}$，$\boldsymbol{x}_i\in\boldsymbol{\chi}\subseteq\mathbb{R}^n$，$y_i\in\{0,1\}$，$i=1,2,\cdots,N$，其中 $\boldsymbol{x}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(n)})^{\mathrm{T}}$。需要知道 $P(y|\boldsymbol{x})$，这里用条件概率的原因是：预测时都是已知 $\boldsymbol{x}$，然后需要判断此时对应的 y 值。

考虑到 $\boldsymbol{w}\cdot\boldsymbol{x}+b$ 取值是连续的，因此它不能拟合离散变量。可以考虑用它来拟合条件

概率 $P(y=1|\boldsymbol{x})$，因为概率的取值也是连续的。但是对于 $\boldsymbol{w}\neq\boldsymbol{0}$（若等于零向量则没有求解的价值），$\boldsymbol{w}\cdot\boldsymbol{x}+b$ 的取值范围为 $-\infty\sim+\infty$，不符合概率取值在范围$[0,1]$的要求，因此考虑采用广义线性模型，最理想的是单位阶跃函数：

$$P(y=1\mid\boldsymbol{x})=\begin{cases}0, & z<0\\0.5, & z=0\\1, & z>0\end{cases},\quad z=\boldsymbol{w}\cdot\boldsymbol{x}+b \tag{3.21}$$

但是阶跃函数不满足单调可导的性质。因此，需要寻找一个可导的、与阶跃函数相似的函数。对数概率函数（logistic function）就是这样的一个替代函数：

$$P(y=1\mid\boldsymbol{x})=\frac{1}{1+\mathrm{e}^{-z}},z=\boldsymbol{w}\cdot\boldsymbol{x}+b \tag{3.22}$$

由于 $P(y=0|\boldsymbol{x})=1-P(y=1|\boldsymbol{x})$，则有

$$\ln\frac{P(y=1\mid\boldsymbol{x})}{P(y=0\mid\boldsymbol{x})}=z=\boldsymbol{w}\cdot\boldsymbol{x}+b$$

$\dfrac{P(y=1|\boldsymbol{x})}{P(y=0|\boldsymbol{x})}$表示样本为正例的可能性与为反例的可能性之比，称为概率（odds），反映了样本作为正例的相对可能性。概率的对数称为对数概率（log odds，又称为 logit）。

下面给出逻辑回归模型参数估计。给定训练数据集 $\boldsymbol{T}=\{(\boldsymbol{x}_1,y_1),(\boldsymbol{x}_2,y_2),\cdots,(\boldsymbol{x}_N,y_N)\}$，其中 $\boldsymbol{x}_i\in\mathbb{R}^n$，$y_i\in\{0,1\}$。模型估计的原理是用极大似然法估计模型参数。

为了便于讨论，将参数 b 吸收进 $\boldsymbol{w}$ 中，即令

$$\tilde{\boldsymbol{w}}=(w^{(1)},w^{(2)},\cdots,w^{(n)},b)^{\mathrm{T}}\in\mathbb{R}^{n+1}$$
$$\tilde{\boldsymbol{x}}=(x^{(1)},x^{(2)},\cdots,x^{(n)},1)^{\mathrm{T}}\in\mathbb{R}^{n+1}$$
$$P(Y=1\mid\tilde{\boldsymbol{x}})=\pi(\tilde{\boldsymbol{x}})=\frac{\exp(\tilde{\boldsymbol{w}}\cdot\tilde{\boldsymbol{x}})}{1+\exp(\tilde{\boldsymbol{w}}\cdot\tilde{\boldsymbol{x}})}$$
$$P(Y=0\mid\tilde{\boldsymbol{x}})=1-\pi(\tilde{\boldsymbol{x}})$$

则似然函数为

$$\prod_{i=1}^{N}[\pi(\tilde{\boldsymbol{x}}_i)]^{y_i}[1-\pi(\tilde{\boldsymbol{x}}_i)]^{1-y_i} \tag{3.23}$$

对数似然函数为

$$\begin{aligned}L(\tilde{\boldsymbol{w}})&=\sum_{i=1}^{N}[y_i\log\pi(\tilde{\boldsymbol{x}})+(1-y_i)\log(1-\pi(\tilde{\boldsymbol{x}}))]\\&=\sum_{i=1}^{N}\left[y_i\log\frac{\pi(\tilde{\boldsymbol{x}})}{1-\pi(\tilde{\boldsymbol{x}})}+\log(1-\pi(\tilde{\boldsymbol{x}}))\right]\end{aligned} \tag{3.24}$$

又由于

$$\pi(\tilde{\boldsymbol{x}})=\frac{\exp(\tilde{\boldsymbol{w}}\cdot\tilde{\boldsymbol{x}})}{1+\exp(\tilde{\boldsymbol{w}}\cdot\tilde{\boldsymbol{x}})}$$

因此有

$$L(\tilde{\boldsymbol{w}})=\sum_{i=1}^{N}[y_i(\tilde{\boldsymbol{w}}\cdot\tilde{\boldsymbol{x}}_i)-\log(1+\exp(\tilde{\boldsymbol{w}}\cdot\tilde{\boldsymbol{x}}_i))] \tag{3.25}$$

对 $L(\tilde{\boldsymbol{w}})$求极大值，得到 $\tilde{\boldsymbol{w}}$ 的估计值。设估计值为 $\tilde{\boldsymbol{w}}^*$，则逻辑回归模型为

$$P(Y=1 \mid X=\tilde{\boldsymbol{x}})=\frac{\exp(\tilde{\boldsymbol{w}}^{*}\cdot\tilde{\boldsymbol{x}})}{1+\exp(\tilde{\boldsymbol{w}}^{*}\cdot\tilde{\boldsymbol{x}})} \tag{3.26}$$

$$P(Y=0 \mid X=\tilde{\boldsymbol{x}})=\frac{1}{1+\exp(\tilde{\boldsymbol{w}}^{*}\cdot\tilde{\boldsymbol{x}})} \tag{3.27}$$

以上讨论的都是二分类的逻辑回归模型,可以推广到多分类逻辑回归模型。设离散型随机变量 Y 的取值集合为$\{1,2,\cdots,K\}$,则多分类逻辑回归模型为

$$P(Y=k \mid \tilde{\boldsymbol{x}})=\frac{\exp(\tilde{\boldsymbol{w}}_k\cdot\tilde{\boldsymbol{x}})}{1+\sum_{k=1}^{K-1}\exp(\tilde{\boldsymbol{w}}_k\cdot\tilde{\boldsymbol{x}})},\quad k=1,2,\cdots,K-1 \tag{3.28}$$

$$P(Y=K \mid \tilde{\boldsymbol{x}})=\frac{1}{1+\sum_{k=1}^{K-1}\exp(\tilde{\boldsymbol{w}}_k\cdot\tilde{\boldsymbol{x}})},\quad \tilde{\boldsymbol{x}}\in\mathbb{R}^{n+1},\tilde{\boldsymbol{w}}_k\in\mathbb{R}^{n+1} \tag{3.29}$$

其参数估计方法与二分类逻辑回归模型类似。

3.2.2　算法步骤

输入：数据集 $T=\{(\boldsymbol{x}_1,y_1),(\boldsymbol{x}_2,y_2),\cdots,(\boldsymbol{x}_N,y_N)\}$,$\boldsymbol{x}_i\in\boldsymbol{\chi}\subseteq\mathbb{R}^n$,$y_i\in Y\subseteq\mathbb{R}$,$i=1,2,\cdots,N$,正则化项系数 $\lambda>0$。

输出：

$$f(\boldsymbol{x})=\boldsymbol{w}\cdot\boldsymbol{x}+b$$

算法步骤：

令

$$\tilde{\boldsymbol{w}}=(w^{(1)},w^{(2)},\cdots,w^{(n)},b)^{\mathrm{T}}=(\boldsymbol{w}^{\mathrm{T}},b)^{\mathrm{T}}$$
$$\tilde{\boldsymbol{x}}=(x^{(1)},x^{(2)},\cdots,x^{(n)},1)^{\mathrm{T}}=(\boldsymbol{x}^{\mathrm{T}},1)^{\mathrm{T}}$$
$$\boldsymbol{y}=(y_1,y_2,\cdots,y_N)^{\mathrm{T}}$$

计算

$$\boldsymbol{x}=(\tilde{\boldsymbol{x}}_1,\tilde{\boldsymbol{x}}_2,\cdots,\tilde{\boldsymbol{x}}_N)^{\mathrm{T}}=\begin{bmatrix}\tilde{\boldsymbol{x}}_1^{\mathrm{T}}\\ \tilde{\boldsymbol{x}}_2^{\mathrm{T}}\\ \vdots\\ \tilde{\boldsymbol{x}}_N^{\mathrm{T}}\end{bmatrix}=\begin{bmatrix}x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(n)} & 1\\ x_2^{(1)} & x_2^{(2)} & \cdots & x_2^{(n)} & 1\\ \vdots & \vdots & \ddots & \vdots & 1\\ x_N^{(1)} & x_N^{(2)} & \cdots & x_N^{(n)} & 1\end{bmatrix}$$

优化求解

$$\tilde{\boldsymbol{w}}^{*}=\underset{\tilde{\boldsymbol{w}}}{\arg\min}\left[(\boldsymbol{y}-\boldsymbol{x}\tilde{\boldsymbol{w}})^{\mathrm{T}}(\boldsymbol{y}-\boldsymbol{x}\tilde{\boldsymbol{w}})+\lambda\|\tilde{\boldsymbol{w}}\|_2^2\right]$$

最终得到模型

$$f(\tilde{\boldsymbol{x}}_i)=\tilde{\boldsymbol{x}}_i^{\mathrm{T}}\tilde{\boldsymbol{w}}^{*}$$

3.2.3　实战

1. 数据集

1）线性回归

在线性回归问题中,使用的数据集是 scikit-learn 自带的一个糖尿病病人的数据集。该

数据集从糖尿病病人采样并整理后，特点如下：

- 数据集有 442 个样本。
- 每个样本有 10 个特征。
- 每个特征都是浮点数，数据的范围为−0.2～0.2。
- 样本的目标为 25～346 的整数。

这里给出加载数据集的函数：

```
def load_data():
  diabetes = datasets.load_diabetes()
  return model_selection.train_test_split(diabetes.data,diabetes.target,
        test_size=0.25,random_state=0)
```

使用 scikit-learn 自带的一个糖尿病病人的数据集 diabetes()，返回值是一个元组，元组依次是：训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

load_data()函数加载数据集并随机切分数据集为两部分，其中 test_size 指定了测试集为原始数据集的大小（比例）。本示例代码中，将数据集拆分成训练集和测试集，测试集大小 test_size 为原始数据集大小的 1/4。

2）逻辑回归

为了测试逻辑回归模型的分类性能，此处选用经典的数据集：鸢尾花数据集，该数据集的详细介绍见 2.1.4 节。

2. Sklearn 实现

1）线性回归

LinearRegression 是 scikit-learn 提供的线性回归模型，其原型如下：

```
class sklearn.linear_model.LinearRegression( * , fit_intercept = True, normalize = 'deprecated',
copy_X = True, n_jobs = None, positive = False)
```

（1）参数。

fit_intercept：一个布尔值，用于指定是否需要计算 b 值。如果为 False，那么不计算 b 值。当

$$\tilde{\boldsymbol{w}}=(w^{(1)},w^{(2)},\cdots,w^{(n)},b)^{\mathrm{T}}=(\boldsymbol{w}^{\mathrm{T}},b)^{\mathrm{T}}$$

$$\tilde{\boldsymbol{x}}=(x^{(1)},x^{(2)},\cdots,x^{(n)},1)^{\mathrm{T}}=(\boldsymbol{x}^{\mathrm{T}},1)^{\mathrm{T}}$$

时，可以设置 fit_intercept=False。

normalize：一个布尔值。如果为 True，那么训练样本会在回归之前被归一化。

copy_X：一个布尔值。如果为 True，则会复制 X。

n_jobs：一个正数。任务并行时指定的 CPU 数量。如果为−1 则使用所有可用的 CPU。

（2）属性。

coef_：权重向量。

ointercept_：b 值。

（3）方法。

fit(X,y[, sample_weight])：训练模型。

predict(X)：用模型进行预测，返回预测值。

score(X,y[, sample_weight])：返回预测性能得分。设预测集为 T_{test}，真实值为 y_i，真实值的平均值为 $\bar{y}$，预测值为 $\hat{y}$，则

$$\text{score} = 1 - \frac{\sum_{T_{\text{test}}} (y_i - \hat{y})^2}{(y_i - \bar{y})^2}$$

■ score 不超过 1，但是可能为负值(预测效果太差)。

■ score 越大，预测性能越好。

首先导入包：

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model, model_selection
```

LinearRegression()函数如下：

```
def test_LinearRegression( * data):
  X_train,X_test,y_train,y_test = data
  regr = linear_model.LinearRegression()
  regr.fit(X_train, y_train)
  print('Coefficients: % s, intercept % .2f' % (regr.coef_,regr.intercept_))
  print("Residual sum of squares: % .2f" % np.mean((regr.predict(X_test) - y_test) ** 2))
  print('Score: % .2f' % regr.score(X_test, y_test))
```

其中，参数 data 依次指定了训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

调用如下函数简单地从训练数据集中学习，然后从测试数据集中预测。

```
X_train,X_test,y_train,y_test = load_data()
  test_LinearRegression(X_train,X_test,y_train,y_test)
```

输出结果如下：

```
Coefficients: [ -43.26774487 -208.67053951593.39797213302.89814903 -560.27689824
261.47657106 -8.83343952135.937151560703.22658427—28.34844354],intercept
153.07Residual sum of squares: 3180.20
Score: 0.36
```

可以看出，测试集中预测结果的均方误差为 3180.20，预测性能得分仅为 0.36(该值越大越好，1.0 为最好)。

2) 逻辑回归

在 scikit-learn 中，LogisticRegression 实现了逻辑回归模型，其原型如下：

```
class sklearn.linear_model.LogisticRegression(penalty = 'l2',  * , dual = False, tol = 0.0001,
C = 1.0, fit_intercept = True, intercept_scaling = 1, class_weight = None, random_state = None,
solver = 'lbfgs', max_iter = 100, multi_class = 'auto', verbose = 0, warm_start = False, n_jobs =
None, l1_ratio = None)
```

(1) 参数。

penalty：一个字符串，用于指定正则化策略。

■ 如果为'L_2'，则优化目标函数为$\frac{1}{2}\|\tilde{\boldsymbol{w}}\|_2^2+CL(\boldsymbol{w}),C>0,L(\boldsymbol{w})$为极大似然函数。

■ 如果为'L_1'，则优化目标函数为$\|\tilde{\boldsymbol{w}}\|_1+CL(\boldsymbol{w}),C>0,L(\boldsymbol{w})$为极大似然函数。

tol：一个浮点数，用于指定判断迭代收敛与否的阈值。

dual：一个布尔值。如果为 True，则求解对偶形式（只在 penalty='L_2'且 solver='liblinear'时有对偶形式）；如果为 False，则求解原始形式。

C：一个浮点数。用于指定罚项系数的倒数。它的值越小，则正则化项越大。

fit_intercept：一个布尔值，用于指定是否需要计算 b 值。如果值为 False，则不会计算 b 值（模型会假设你的数据已经中心化）。当

$$\tilde{\boldsymbol{w}}=(w^{(1)},w^{(2)},\cdots,w^{(n)},b)^{\mathrm{T}}=(\boldsymbol{w}^{\mathrm{T}},b)^{\mathrm{T}}$$
$$\tilde{\boldsymbol{x}}=(x^{(1)},x^{(2)},\cdots,x^{(n)},1)^{\mathrm{T}}=(\boldsymbol{x}^{\mathrm{T}},1)^{\mathrm{T}}$$

时，可以设置 fit_intercept 为 False。

intercept_scaling：一个浮点数。只有当 solver='liblinear'时才有意义。当采用 fit_intercept 时，相当于人造一个特征出来，该特征恒为 1，其权重为 b。在计算正则化项时，该人造特征也被考虑了。因此为了降低该人造特征的影响，需要提供 intercept_scaling。

class_weight：一个字典或者字符串'balanced'。

■ 如果为字典，则字典给出了每个分类的权重，如{class_label：weight}。

■ 如果为字符串'balanced'，则每个分类的权重与该分类在样本集中出现的频率成反比。

■ 如果未指定，则每个分类的权重都为 1。

max_iter：一个整数，用于指定最大迭代次数。

random_state：一个整数或者一个 RandomState 实例，或者 None。

■ 如果为整数，则它指定了随机数生成器的种子。

■ 如果为 RandomState 实例，则指定了随机数生成器。

■ 如果为 None，则使用默认的随机数生成器。

solver：一个字符串，用于指定求解最优化问题的算法，可以为如下值。

■ 'newton-cg'：使用牛顿法。

■ 'lbfgs'：使用 L-BFGs 拟牛顿法。

■ 'liblinear'：使用 liblinear。

■ 'sag'：使用 SAG(Stochastic Average Gradient，随机平均梯度下降)算法。

注意：对于规模小的数据集，'liblinear'比较适用；对于规模大的数据集，'sag'比较适用。'newton-cg'、'lbfgs'、'sag'只处理 penalty='12'的情况。

multi_class：一个字符串，用于指定多分类问题的策略，可以为如下值。

■ 'ovr'：采用 one-vs-rest 策略。

■ 'multinomial'：直接采用多分类逻辑回归策略。

auto：如果数据是二分类的，或者如果 solver='liblinear'，则'auto'选择'ovr'，否则选择'multionmial'。

verbose：一个正数。用于开启/关闭迭代中间输出日志功能。

warm_start：一个布尔值。如果为 True，则使用前一次训练结果继续训练，否则从头开始训练。

n_jobs：一个正数。用于指定任务并行时的 CPU 数量。如果为－1 则使用所有可用的 CPU。

（2）属性。

coef_：权重向量。

intercept_：b 值。

n_iter_：实际迭代次数。

（3）方法。

fit(X,y[,sample_weight])：训练模型。

predict(X)：用模型进行预测，返回预测值。

predict_log_proba(X)：返回一个数组，数组的元素依次是 X 预测为各个类别的概率的对数值。

predict_proba(X)：返回一个数组，数组的元素依次是 X 预测为各个类别的概率值。oscore(X,y[,sample_weight])：返回在(X,y)上预测的准确率(accuracy)。

LogisticRegression()函数如下：

```
def test_LogisticRegression( * data):
  X_train,X_test,y_train,y_test = data
  regr = linear_model.LogisticRegression()
  regr.fit(X_train, y_train)
  print('Coefficients: % s, intercept % s' % (regr.coef_,regr.intercept_))
  print('Score: % .2f' % regr.score(X_test, y_test))
```

其中，参数 data 依次指定了训练样本集、测试样本集、训练样本集对应的标签值、测试样本集对应的标签值。

该函数简单地从训练数据集中学习，然后从测试数据集中预测。这里 LogisticRegression()函数所有的参数都采用默认值。

调用 LogisticRegression()函数：

```
X_train,X_test,y_train,y_test = load_data()
test_LogisticRegression(X_train,X_test,y_train,y_test)
```

输出结果如下：

```
Coefficients:[[ 0.407697191.32793253 - 2.12687162 - 0.96614355][ 0.1932691 - 1.31070419
0.60821724 - 1.19814744]
[ - 1.50100362 - 1.33529511 2.16377642 2.23963779]],
intercept [ 0.244621181.13229922 - 1.08042606]
Score: 0.97
```

可以看出，测试集中的预测结果性能得分为 0.97(即预测准确率为 97%)。

下面考察 multi_class 参数对分类结果的影响。默认采用的是 one-vs-rest 策略，但是逻辑回归模型的原型就支持多分类，给出的测试函数如下：

```
def test_LogisticRegression_multinomial( *data):
  X_train,X_test,y_train,y_test = data
  regr = linear_model.LogisticRegression(multi_class = 'multinomial',solver = 'lbfgs')
  regr.fit(X_train, y_train)
  print('Coefficients: %s, intercept %s' % (regr.coef_,regr.intercept_))
  print('Score: %.2f' % regr.score(X_test, y_test))
```

注意：只有 solver 为牛顿法或者拟牛顿法时才能配合 multi_class＝'multinomial'，否则报错。

调用 LogisticRegression_multinomial()函数：

```
X_train,X_test,y_train,y_test = load_data()
test_LogisticRegression_multinomial(X_train,X_test,y_train,y_test)
```

测试结果如下：

```
Coefficients: [[ - 0.36834533  0.84161813  - 2.27865338  - 0.98934494 ] [ 0.34136192
 - 0.33359843  - 0.031646  - 0.8294743 ]
[ 0.0269834  - 0.5080197  2.31029938  1.81881924 ]], intercept [ 8.77142226  2.34153563
 - 11.11295788]
Score: 1.00
```

可以看出，在这个问题中，多分类策略进一步提升了预测准确率。这里的准确率提升到100％，说明对于测试集的数据，LogisticRegression 分类器完全预测正确。

最后，考察参数 C 对分类模型的预测性能的影响。C 是正则化项系数的倒数，它越小则正则化项的权重越大。给出的测试函数如下：

```
def test_LogisticRegression_C( *data):
  X_train,X_test,y_train,y_test = data
  Cs = np.logspace( - 2,4,num = 100)
  scores = []
  for C in Cs:
     regr = linear_model.LogisticRegression(C = C)
     regr.fit(X_train, y_train)
     scores.append(regr.score(X_test, y_test))
  ## 绘图
  fig = plt.figure()
  ax = fig.add_subplot(1,1,1)
  ax.plot(Cs,scores)
  ax.set_xlabel(r"C")
  ax.set_ylabel(r"score")
  ax.set_xscale('log')
  ax.set_title("LogisticRegression")
  plt.show()
```

测试结果如图 3.8 所示。可以看到随着 C 的增大(即正则化项减小)，LogisticRegression 的预测准确率上升。当 C 增大到一定程度(即正则化项减小到一定程度)时，LogisticRegression 的预测准确率维持在较高的水准保持不变。

3. 算法实现

为了使用逻辑回归模型对鸢尾花进行分类，此处选用经典的鸢尾花数据集，该数据集的

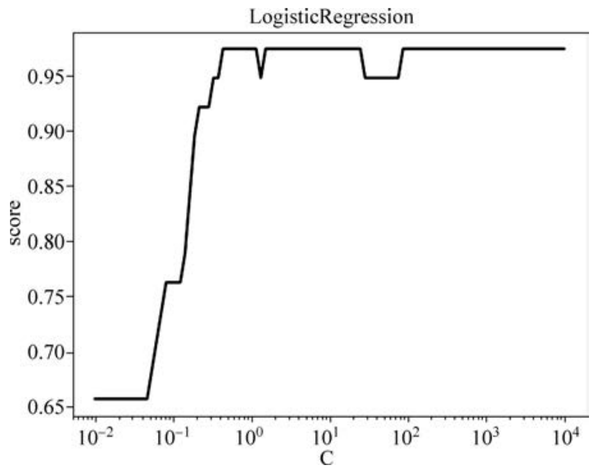


图 3.8　测试结果

详细介绍见 2.1.4 节。

现只取数据集 Iris 中的两个特征 Sepal. Length(花萼长度)和 Petal. Length(花瓣长度),定义为 X(X1, X2),对应 y 分类中的两个类别(0,1),将根据 X(X1,X2)的值对鸢尾花进行分类。首先绘制这两个特征的散点图,代码如下。

```
# 程序名称:logicscatter.py
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import numpy as np

iris = load_iris()
data = iris.data
target = iris.target
# print (data[:10])
# print (target[10:])
X = data[0:100, [0, 2]]
y = target[0:100]
print(X[:5])
print(y[-5:])
label = np.array(y)
index_0 = np.where(label == 0)
plt.scatter(X[index_0, 0], X[index_0, 1], marker = 'x', color = 'b', label = '0', s = 15)
index_1 = np.where(label == 1)
plt.scatter(X[index_1, 0], X[index_1, 1], marker = 'o', color = 'r', label = '1', s = 15)
plt.xlabel('X1')
plt.ylabel('X2')
plt.legend(loc = 'upper left')
plt.show()
```

程序 Logicscatter. py 的运行结果如图 3.9 所示。

接着编写一个逻辑回归模型的类,然后训练测试,计算损失函数(损失函数的本质是衡量“模型预估值”到“实际值”的距离)。注意损失函数值越小,模型越好,而且损失函数尽量是一个凸函数,便于收敛计算。逻辑回归模型预估的是样本属于某个分类的概率,其损失函数可以采用均方差、对数、概率等方法。计算损失函数的程序代码如下。

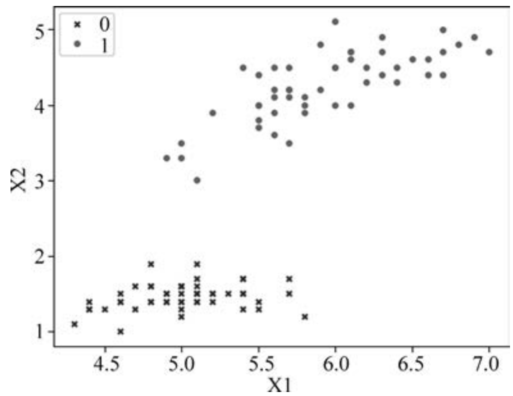


图 3.9 程序 Logicscatter.py 的运行结果

```
# 程序名称:logicregressionloss.py
import matplotlib.pyplot as plt
import numpy as np

class logistic(object):
  def __init__(self):
     self.W = None

  def train(self, X, y, learn_rate = 0.01, num_iters = 5000):
     num_train, num_feature = X.shape
     #初始化权重
     self.W = 0.001 * np.random.randn(num_feature, 1).reshape((-1, 1))
     loss = []
     for i in range(num_iters):
         error, dW = self.compute_loss(X, y)
         self.W += -learn_rate * dW
         loss.append(error)
         if i % 200 == 0:
             print('i = %d,error = %f' % (i, error))
     return loss

  def compute_loss(self, X, y):
     num_train = X.shape[0]
     h = self.output(X)
     loss = -np.sum((y * np.log(h) + (1 - y) * np.log((1 - h))))
     loss = loss / num_train
     dW = X.T.dot((h - y)) / num_train
     return loss, dW

  def output(self, X):
     g = np.dot(X, self.W)
     return self.sigmoid(g)

  def sigmoid(self, X):
     return 1 / (1 + np.exp(-X))

  def predict(self, X_test):
     h = self.output(X_test)
```

```
        y_pred = np.where(h >= 0.5, 1, 0)
        return y_pred

    y = y.reshape((-1, 1))
    # 添加全1的列向量在X矩阵左侧
    one = np.ones((X.shape[0], 1))
    X_train = np.hstack((one, X))
    classify = logistic()
    loss = classify.train(X_train, y)
    print(classify.W)
    plt.plot(loss)
    plt.xlabel('Iteration number')
    plt.ylabel('Loss value')
    plt.show()
```

程序 logicregressionloss.py 的运行结果如图 3.10 所示。

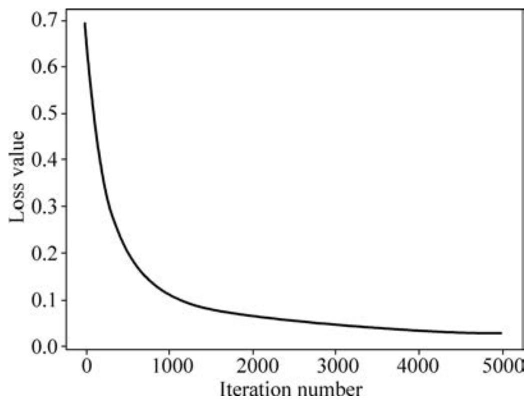


图 3.10　程序 logicregressionloss.py 的运行结果

以绘图的方式对“决策边界”可视化，代码如下。

```
# 程序名称：logicdrawborder.py
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_iris


class logistic(object):
    def __init__(self):
        self.W = None

    def train(self, X, y, learn_rate = 0.01, num_iters = 5000):
        num_train, num_feature = X.shape
        #初始化权重
        self.W = 0.001 * np.random.randn(num_feature, 1).reshape((-1, 1))
        loss = []
        for i in range(num_iters):
            error, dW = self.compute_loss(X, y)
            self.W += -learn_rate * dW
            loss.append(error)
```

```
            if i % 200 == 0:
                print('i = %d,error = %f' % (i, error))
        return loss

    def compute_loss(self, X, y):
        num_train = X.shape[0]
        h = self.output(X)
        loss = -np.sum((y * np.log(h) + (1 - y) * np.log((1 - h))))
        loss = loss / num_train
        dW = X.T.dot((h - y)) / num_train
        return loss, dW

    def output(self, X):
        g = np.dot(X, self.W)
        return self.sigmoid(g)

    def sigmoid(self, X):
        return 1 / (1 + np.exp(-X))

    def predict(self, X_test):
        h = self.output(X_test)
        y_pred = np.where(h >= 0.5, 1, 0)
        return y_pred

    iris = load_iris()
    data = iris.data
    target = iris.target
    # print (data[:10])
    # print (target[10:])
    X = data[0:100, [0, 2]]
    y = target[0:100]
    y = y.reshape((-1, 1))
    one = np.ones((X.shape[0], 1))
    X_train = np.hstack((one, X))
    classify = logistic()
    loss = classify.train(X_train, y)
    label = np.array(y)
    index_0 = np.where(label == 0)
    plt.scatter(X[index_0, 0], X[index_0, 1], marker = 'x', color = 'b', label = '0', s = 15)
    index_1 = np.where(label == 1)
    plt.scatter(X[index_1, 0], X[index_1, 1], marker = 'o', color = 'r', label = '1', s = 15)
    # 绘制分类边界线
    x1 = np.arange(4, 7.5, 0.5)
    x2 = (- classify.W[0] - classify.W[1] * x1) / classify.W[2]
    plt.plot(x1, x2, color = 'black')
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.legend(loc = 'upper left')
    plt.show()
```

程序 logicdrawborder.py 的运行结果如图 3.11 所示，可以看出，最后学习得到的决策边界成功地隔开了两个类别。

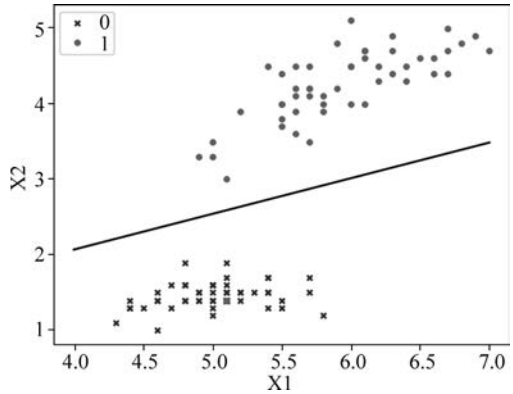


图 3.11　程序 logicdrawborder.py 的运行结果

3.2.4　实验

1. 实验目的

理解逻辑回归的算法原理，掌握损失函数的优化方法，并分别利用 scikit-learn 的相关包、Python 语言编程来实现该算法。

2. 实验数据

数据集选用经典的鸢尾花数据集，该数据集的详细介绍见 2.1.4 节。

3. 实验要求

(1) 实现数据可视化。

通过数据文件导入数据，并使用 Matplotlib 工具建立对应散点图。

(2) 将线性回归参数初始化为 0，计算损失函数(cost function)的初始值，根据算法基本原理中的损失函数计算公式来计算。

(3) 选择以下两种优化方法分别求解逻辑回归参数。

■ 梯度下降法。

■ 牛顿迭代法。

(4) 对验证集进行验证。

(5) 画出分类边界。

3.3　决策树算法

决策树(decision tree)是功能强大而且相当受欢迎的分类和预测方法，它是一种有监督的学习算法，以树状图为基础，其输出结果为一系列简单实用的规则，故得名决策树。决策树就是一系列的 if-then 语句，决策树可以用于分类问题，也可以用于回归问题。在讲解原理时为了表述方便，以分类问题为例。

决策树模型基于特征对实例进行分类，它是一种树状结构。决策树的优点是可读性强，分类速度快。学习决策树时，通常采用损失函数最小化原则。

决策树的经典算法包括 ID3、C4.5 和 CART，其中 ID3 和 C4.5 是基于信息论方法的，

而 CART 是基于最小基尼系数计算方法的。

在本章中，训练集用 $\boldsymbol{D}$ 表示，T 表示一棵决策树。

3.3.1 原理简介

决策树算法是一个贪心算法，即在特性空间上执行递归的二元分割，决策树由节点和有向边组成。内部节点表示一个特征或者属性；叶子节点表示一个分类。使用决策树进行分类时，将实例分配到叶节点的类中，该叶节点所属的类就是该节点的分类。

决策树可以表示给定特征条件下，类别的条件概率分布。将特征空间划分为互不相交的单元 $S_1, S_2, \cdots, S_m$。设某个单元 S_i 内部有 N_i 个样本点，则它定义了一个条件概率分布 $P(y=c_k|X), X \in S_i, c_k(k=1,2,\cdots,K)$ 为第 k 个分类。

- 每个单元对应于决策树的一条路径。
- 所有单元的条件概率分布构成了决策树所代表的条件概率分布。
- 在单元 S_i 内部有 N_i 个样本点，但是整个单元都属于类 $\hat{c}_k$。其中，$\hat{c}_k = \underset{c_k}{\operatorname{argmax}} P(y=c_k|X), X \in S_i$。即单元 S_i 内部的 N_i 个样本点，哪个分类占优，则整个单元都属于该类。

3.3.2 算法步骤

构建决策树通常包括 3 个步骤：特征选择；决策树生成；决策树剪枝。

假设给定训练集 $\boldsymbol{D}=\{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \cdots, (\boldsymbol{x}_N, y_N)\}$，其中 $\boldsymbol{x}_i=(x_i^{(1)}, x_i^{(2)}, \cdots, x_i^{(n)})$ 为输入实例，n 为特征个数；$y_i \in \{1,2,\cdots,K\}$ 为类标记，$i=1,2,\cdots,N$；N 为样本容量。构建决策树的目标是，根据给定的训练数据集学习一个决策树模型。

构建决策树时，通常将正则化的极大似然函数作为损失函数，其学习目标是损失函数为目标函数的最小化。构建决策树的算法通常是递归地选择最优特征，并根据该特征对训练数据进行分割，其步骤如下。

（1）构建根节点，使所有训练样本都位于根节点。

（2）选择一个最优特征。通过该特征将训练数据分割成多个子集，确保各个子集都有最好的分类，但要考虑下列两种情况。

- 若子集已能够被较好地分类，则构建叶节点，并将该子集划分到对应的叶节点。
- 若某个子集不能够被较好地分类，则对该子集继续划分。

（3）递归执行，直至所有的训练样本都被较好地分类，或者没有合适的特征为止。是否被较好地分类，可通过后面介绍的指标来判断。

通过如上步骤生成的决策树对训练样本有很好的分类能力，但是需要的是对未知样本的分类能力。因此通常需要对已生成的决策树进行剪枝，从而使得决策树具有更好的泛化能力。剪枝过程是去掉过于细分的叶节点，从而提高泛化能力。

1. 特征选择

特征选择就是选取有较强分类能力的特征。分类能力通过信息增益或者信息增益比来刻画。选择特征的标准是找出局部最优的特征作为判断进行切分，取决于切分后节点数据集中类别的有序程度（纯度），划分后的分区数据越纯，切分规则越合适。可衡量节点数据集

纯度的有熵、基尼系数和方差。熵和基尼系数是针对分类的,方差是针对回归的。

先给出熵的定义。设 X 是一个离散型随机变量,其概率分布为

$$P(X=\boldsymbol{x}_i)=p_i,\quad i=1,2,\cdots,n \tag{3.30}$$

则随机变量 X 的熵为

$$H(X)=-\sum_{i=1}^{n}p_i\log p_i \tag{3.31}$$

其中,定义 $0\log 0=0$。

当随机变量 X 只取两个值时,X 的分布为

$$P(X=1)=p$$
$$P(X=0)=1-p,\quad 0\leqslant p\leqslant 1$$

此时熵为

$$H(p)=-p\log p-(1-p)\log(1-p),\quad 0\leqslant p\leqslant 1$$

当 $p=0$ 或者 $p=1$ 时,熵最小(为 0),此时随机变量不确定性最小;当 $p=0.5$ 时,熵最大(为 1),此时随机变量不确定性最大。

熵 $H(p)$的函数图像如图 3.12 所示。

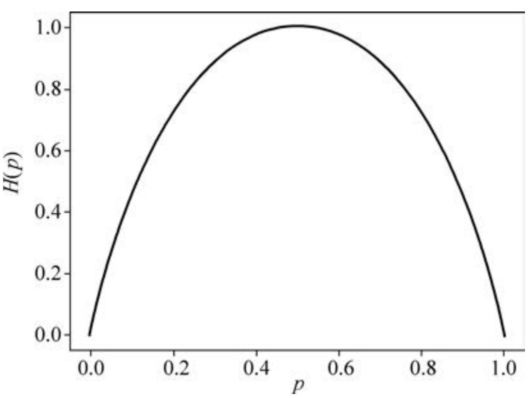


图 3.12　熵 $\boldsymbol{H(p)}$的函数图像

设随机变量为(X,Y),其联合概率分布为

$$P(X=\boldsymbol{x}_i,Y=y_j)=p_{ij},\quad i=1,2,\cdots,n,\quad j=1,2,\cdots,m \tag{3.32}$$

则条件熵 $H(Y|X)$定义为

$$H(Y\mid X)=\sum_{i=1}^{n}P_X(X=\boldsymbol{x}_i)H(Y\mid X=\boldsymbol{x}_i) \tag{3.33}$$

其中

$$P_X(X=\boldsymbol{x}_i)=\sum_{Y}P(X=\boldsymbol{x}_i,Y)$$

当熵中的概率由数据估计得到时,被称为经验熵;当条件熵中的概率由数据估计得到时,被称为经验条件熵。

对于数据集 $\boldsymbol{D}$,通过 $H(Y)$来刻画数据集 $\boldsymbol{D}$ 的不确定程度。当数据集 $\boldsymbol{D}$ 中的所有样本都是同一种类别时,$H(Y)=0$。也将 $H(Y)$记作 $H(\boldsymbol{D})$。给定特征 A 和训练数据集 $\boldsymbol{D}$,定义信息增益 $g(\boldsymbol{D},A)=H(\boldsymbol{D})-H(\boldsymbol{D}|A)$。

信息增益刻画的是由于特征 A 而使得对数据集 $\boldsymbol{D}$ 的分类的不确定性减少的程度。构建决策树选择信息增益大的特征来划分数据集。

这里给出计算信息增益的算法。假设训练数据集为 $\boldsymbol{D}$，N 为其训练数据集容量。假设有 k 个类别，依次为 c_k，$k=1,2,\cdots,K$。设 $|C_k|$ 为属于类 c_k 的样本个数。

设特征 A 是离散的，且有 n 个不同的取值，即取值集合为 $\{a_1,a_2,\cdots,a_n\}$，根据特征 A 的取值将 $\boldsymbol{D}$ 划分出 n 个子集 $\boldsymbol{D}_1,\boldsymbol{D}_2,\cdots,\boldsymbol{D}_n$，$N_i$ 为对应的 $\boldsymbol{D}_i$ 中的样本个数。

设集合 $\boldsymbol{D}_i$ 中属于类 c_k 的样本集合为 $\boldsymbol{D}_{ik}$，其容量为 N_{ik}，信息增益算法如下。

输入：

■ 训练数据集 $\boldsymbol{D}$。

■ 特征 A。

输出：信息增益 $g(\boldsymbol{D},A)$。

算法步骤：

■ 根据式(3.34)计算数据集 $\boldsymbol{D}$ 的经验熵 $\mathrm{H}(\boldsymbol{D})$。它就是训练数据集 $\boldsymbol{D}$ 中，分类 Y 的概率估计 $\hat{P}(Y=c_k)=\dfrac{|C_k|}{N}$ 计算得到的经验熵。

$$H(\boldsymbol{D})=-\sum_{k=1}^{K}\frac{|C_k|}{N}\log\frac{|C_k|}{N} \tag{3.34}$$

■ 根据下式计算特征 A 对于数据集 $\boldsymbol{D}$ 的经验条件熵 $H(\boldsymbol{D}|A)$。它使用了特征 A 的概率估计：$\hat{P}(X^{(A)}=\alpha_i)=\dfrac{|N_i|}{N}$，以及经验条件熵 $\hat{H}(\boldsymbol{D}\mid X^{(A)}=\alpha_i)=\sum\limits_{k=1}^{K}-\left(\dfrac{|N_{ik}|}{N_i}\log\dfrac{|N_{ik}|}{N_i}\right)$（其中使用了条件概率估计 $\hat{P}(Y=c_k|X^{(A)}=\alpha_i)=\dfrac{N_{ik}}{N_i}$，其意义是：在子集 $\boldsymbol{D}_i$ 中，Y 的分布）

$$H(\boldsymbol{D}\mid A)=\sum_{i=1}^{n}\frac{|N_i|}{N}\sum_{k=1}^{K}-\left(\frac{|N_{ik}|}{N_i}\log\frac{|N_{ik}|}{N_i}\right) \tag{3.35}$$

■ 根据下式计算信息增益：

$$g(\boldsymbol{D},A)=H(\boldsymbol{D})-H(\boldsymbol{D}\mid A)$$

熵越大，表示越混乱；熵越小，表示越有序。因此，信息增益表示混乱的减少程度（或者说是有序的增加程度）。

以信息增益作为划分训练集的特征选取方案，存在偏向于选取值较多的特征的问题。公式如下：

$$\begin{aligned}g(\boldsymbol{D},A)&=H(\boldsymbol{D})-H(\boldsymbol{D}\mid A)\\&=H(\boldsymbol{D})-\sum_{i=1}^{n}\frac{|N_i|}{N}\sum_{k=1}^{K}-\left(\frac{|N_{ik}|}{N_i}\log\frac{|N_{ik}|}{N_i}\right)\end{aligned} \tag{3.36}$$

在极限情况下，特征 A 将每个样本一一对应到对应的节点中时（即每个节点中有且仅有一个样本），此时 $\dfrac{N_{ik}}{N_i}=1$，$i=1,2,\cdots,n$，条件熵部分为 0。而条件熵的最小值为 0，这意味着该情况下的信息增益达到了最大值。然而，这个特征 A 显然不是最佳的选择。

可以通过定义信息增益比来解决。特征 A 对训练集 $\boldsymbol{D}$ 的信息增益比 $g_R(\boldsymbol{D},A)$ 定义为

$$g_R(\boldsymbol{D},A)=\frac{g(\boldsymbol{D},A)}{H_A(\boldsymbol{D})} \tag{3.37}$$

$$H_A(\boldsymbol{D})=-\sum_{i=1}^{n}\frac{N_i}{N}\log\frac{N_i}{N} \tag{3.38}$$

$H_A(\boldsymbol{D})$刻画了特征 A 对训练集 $\boldsymbol{D}$ 的分辨能力。但是这不表征它对类别的分辨能力。比如 A 将 $\boldsymbol{D}$ 切分成了两块：$\boldsymbol{D}_1$ 和 $\boldsymbol{D}_2$，则很有可能 $H(\boldsymbol{D})=H(\boldsymbol{D}_1)=H(\boldsymbol{D}_2)$（如每个子集 $\boldsymbol{D}_i$ 中各类别样本的比例与 $\boldsymbol{D}$ 中各类别样本的比例相同）。

2. 决策树生成

基本的决策树生成算法中，典型的有 ID3 生成算法和 C4.5 生成算法，它们生成树的过程大致相似。ID3 采用信息增益作为特征选择的度量，而 C4.5 则采用信息增益比。

1）ID3 生成算法

ID3 生成算法应用信息增益准则选择特征，其算法描述如下。

（1）输入。

- 训练数据集 $\boldsymbol{D}$。
- 特征集合 A。
- 特征信息增益阈值 $\varepsilon>0$。

（2）输出：决策树 T。

（3）算法步骤。

- 若 $\boldsymbol{D}$ 中所有实例均属于同一类 c_k，则 T 为单节点树，并将 c_k 作为该节点的类标记，返回 T。这是一种特殊情况：$\boldsymbol{D}$ 的分类集合只有一个分类。
- 若 $A=\varnothing$，则 T 为单节点树，将 $\boldsymbol{D}$ 中实例数最大的类 c_k 作为该节点的类标记，返回 T（即多数表决）。这也是一种特殊情况：$\boldsymbol{D}$ 的特征集合为空。
- 否则计算 $g(\boldsymbol{D},A_i)$，其中 $A_i\in A$ 为特征集合中的各个特征，选择信息增益最大的特征 A_g。
- 判断 A_g 的信息增益如下。
  - 若 $g(\boldsymbol{D},A_g)<\varepsilon$，则置 T 为单节点树，将 $\boldsymbol{D}$ 中实例数最大的类 c_k 作为该节点的类标记，返回 T。
  - 若 $g(\boldsymbol{D},A_g)\geqslant\varepsilon$，则对 A_g 特征的每个可能取值 α_i，根据 $A_g=\alpha_i$ 将 $\boldsymbol{D}$ 划分为若干非空子集 $\boldsymbol{D}_i$，将 $\boldsymbol{D}_i$ 中实例数最大的类作为标记，构建子节点，由子节点及其子节点构成树 T，返回 T。
- 对第 i 个子节点，以 $\boldsymbol{D}_i$ 为训练集，以 $A-\{A_g\}$为特征集，递归地调用前面的步骤，得到子树 T_i，返回 T_i。

2）C4.5 生成算法

C4.5 生成算法应用信息增益比来选择特征，其算法描述如下。

（1）输入。

- 训练数据集 $\boldsymbol{D}$。
- 特征集 A。
- 特征信息增益阈值 $\varepsilon>0$。

（2）输出：决策树 T。

(3) 算法步骤。

- 若 $\boldsymbol{D}$ 中的所有实例均属于同一类 c_k，则 T 为单节点树，并将 c_k 作为该节点的类标记，返回 T。这是一种特殊情况：$\boldsymbol{D}$ 的分类集合只有一个分类。
- 若 $A=\varnothing$，则 T 为单节点树，将 $\boldsymbol{D}$ 中实例数最大的类 c_k 作为该节点的类标记，返回 T(即多数表决)。这也是一种特殊情况：$\boldsymbol{D}$ 的特征集合为空。
- 否则计算 $g_R(\boldsymbol{D},A_i)$，其中 $A_i \in A$ 为特征集合中的各个特征，选择信息增益最大的特征 A_g。
- 判断 A_g 的信息增益如下。
  - 若 $g_R(\boldsymbol{D},A_i)<\varepsilon$，则置 T 为单节点树，将 $\boldsymbol{D}$ 中实例数最大的类 c_k 作为该节点的类标记(即多数表决)，返回 T。
  - 若 $g_R(\boldsymbol{D},A_g)\geqslant\varepsilon$，则对 A_g 特征的每个可能取值 α_i，根据 $A_g=\alpha_i$ 将 $\boldsymbol{D}$ 划分为若干非空子集 $\boldsymbol{D}_i$，将 $\boldsymbol{D}_i$ 中实例数最大的类作为标记(即多数表决)，构建子节点，由子节点及其子节点构成树 T，返回 T。
- 对第 i 个子节点，以 $\boldsymbol{D}_i$ 为训练集，以 $A-\{A_g\}$ 为特征集，递归地调用前面的步骤，得到子树 T_i，返回 T_i。

关于 C4.5 生成算法的几点说明如下。

- C4.5 算法继承了 ID3 算法的优点，并在以下几方面对 ID3 算法进行了改进。
  - 用信息增益比来选择属性，克服了用信息增益选择属性时偏向选择取值多的属性的不足。
  - 在树的构造过程中进行剪枝。
  - 能够完成对连续属性的离散化处理。
  - 能够对不完整数据进行处理。
- C4.5 算法有如下优点：产生的分类规则易于理解，准确率较高。其缺点是：在构造树的过程中，需要对数据集进行多次的顺序扫描和排序，因而导致算法低效。此外，C4.5 算法只适合于能够驻留于内存的数据集，当训练集大得内存无法容纳时，程序无法运行。
- 决策树可能只用到特征集中的部分特征。
- C4.5 算法和 ID3 算法只有树的生成算法，生成的树容易产生过拟合现象。即对训练集的匹配很好，但是预测测试集的效果较差。

3. 决策树剪枝

决策树需要剪枝的原因是：决策树生成算法生成的树对训练数据的预测很准确，但是对未知数据的分类却很差，这就产生了过拟合的现象。其实，原理都是一样的，决策树的构建是直到没有特征可选或者信息增益很小，这就导致构建的决策树模型过于复杂，而复杂的模型是在训练数据集上建立的，所以对于测试集往往造成分类的不准确，这就是过拟合。发生过拟合是由于决策树太复杂，解决过拟合的方法是控制模型的复杂度，对于决策树来说就是简化模型，通常被称为剪枝。

决策树剪枝的过程是从已生成的决策树上裁掉一些子树或者叶节点。剪枝的目标是通过极小化决策树的整体损失函数或代价函数来实现的。

决策树剪枝的目的是通过剪枝来提高泛化能力。剪枝的思路就是在决策树对训练数据

的预测误差和数据复杂度之间找到一个平衡。

设决策树 T 的叶节点个数为 $|T_f|$，t 为树的叶节点，该叶节点有 N_t 个样本点，其中属于 c_k 类的样本点有 $N_{tk}(k=1,2,\cdots,K)$ 个，则有 $\sum_{k=1}^{K}N_{tk}=N_t$。

令 $H(t)$ 为叶节点 t 上的经验熵，$\alpha\geqslant 0$ 为参数，则决策树 T 的损失函数定义为

$$C_\alpha(T)=\sum_{t=1}^{|T_f|}N_tH(t)+\alpha\mid T_f\mid H(t)=-\sum_{k=1}^{K}\frac{N_{tk}}{N_t}\log\frac{N_{tk}}{N_t} \tag{3.39}$$

令

$$C(T)=\sum_{t=1}^{|T_f|}N_tH(t)=-\sum_{t=1}^{|T_f|}\sum_{k=1}^{K}N_{tk}\log\frac{N_{tk}}{N_t} \tag{3.40}$$

则

$$C_\alpha(T)=C(T)+\alpha\mid T_f\mid$$

其中，$\alpha|T_f|$ 为正则化项，$C(T)$ 表示预测误差。

- $C(T)=0$ 意味着 $N_{tk}=N_t$，即每个叶节点 t 内的样本都是纯的(即单一的分类，而不是杂的)。
- 决策树划分得越细致，则决策树 T 的叶节点越多，$|T_f|$ 越大；$|T_f|$ 小于或等于样本集的数量，当取等号时，决策树 T 的每个叶节点只有一个样本点。
- 参数 α 控制预测误差与模型复杂度之间的关系如下。
  - 较大的 α 会选择较简单的模型。
  - 较小的 α 会选择较复杂的模型。
  - $\alpha=0$ 时只考虑训练数据与模型的拟合程度，不考虑模型复杂度。

剪枝算法的描述如下。

(1) 输入。

- 生成树 T。
- 参数 α。

(2) 输出：剪枝树 T_α。

(3) 算法步骤。

① 计算每个节点的经验熵。

- 递归地从树的叶节点向上回退。
- 设一组叶节点回退到父节点之前与之后的整棵树分别为 T_t 与 T_t'，对应的损失函数值分别为 $C_\alpha(T_t)$ 与 $C_\alpha(T_t')$。若 $C_\alpha(T_t')\leqslant C_\alpha(T_t)$，则进行剪枝，并将父节点变成新的叶节点。

② 递归执行步骤①，直到不能继续为止，得到损失函数最小的子树 T_α。

4. CART 模型

分类与回归树(classification and regression tree，CART)模型也是一种决策树模型，它既可用于分类，也可用于回归。其学习算法分为如下两步。

决策树生成：用训练数据生成决策树，生成树尽可能地大。

决策树剪枝：基于损失函数最小化的标准，用验证数据对生成的决策树剪枝。

分类与回归树模型采用不同的最优化策略。CART 回归生成树用平方误差最小化策

略，CART 分类生成树采用基尼系数最小化策略。

1）决策树生成算法

（1）CART 回归树。

给定训练数据集 $\boldsymbol{D}=\{(\boldsymbol{x}_1,y_1),(\boldsymbol{x}_2,y_2),\cdots,(\boldsymbol{x}_N,y_N)\}$，$y_i\in\mathbb{R}$。设已经将输入空间划分为 M 个单元 $R_1,R_2,\cdots,R_m$，且在单元 R_m 上输出值为 c_m，$m=1,2,\cdots,M$。则回归树模型为

$$f(\boldsymbol{x})=\sum_{m=1}^{M}c_m I(\boldsymbol{x}\in R_m) \tag{3.41}$$

其中，$I(\cdot)$为示性函数。

如果给定输入空间的一个划分，回归树在训练数据集上的误差（平方误差）为 $\sum\limits_{\boldsymbol{x}_i\in R_m}(y_i-f(\boldsymbol{x}_i))^2$。

基于平方误差最小的准则，可以求解出每个单元上的最优输出值 $\hat{c}_m$：

$$\hat{c}_m=\mathrm{ave}(y_i\mid \boldsymbol{x}_i\in R_m)$$

它就是 R_m 上所有输入样本对应的输出 y_i 的平均值。

现在需要找到最佳的划分，使得该划分对应的回归树的平方误差在所有划分中最小。设 $\boldsymbol{x}_i=(x_i^{(1)},x_i^{(2)},\cdots,x_i^{(k)})$，即输入为 k 维。选择第 j 维 $\boldsymbol{x}_i^{(j)}$，它的取值 s 作为切分变量和切分点。定义两个区域：

$$R_1(j,s)=\{\boldsymbol{x}\mid x^{(j)}\leqslant s\},\quad R_2(j,s)=\{\boldsymbol{x}\mid x^{(j)}>s\} \tag{3.42}$$

然后寻求最优切分变量 j 和最优切分点 s。即求解

$$\min_{j,s}\Big[\min_{c_1}\sum_{\boldsymbol{x}_i\in R_1(j,s)}(y_i-c_1)^2+\min_{c_2}\sum_{\boldsymbol{x}_i\in R_2(j,s)}(y_i-c_2)^2\Big] \tag{3.43}$$

对于给定的维度 j 可以找到最优切分点 s。同时有

$$\begin{aligned}\hat{c}_1&=\mathrm{ave}(y_i\mid \boldsymbol{x}_i\in R_1(j,s))\\ \hat{c}_2&=\mathrm{ave}(y_i\mid \boldsymbol{x}_i\in R_2(j,s))\end{aligned} \tag{3.44}$$

问题是如何求解 j 呢？首先遍历所有的维度，找到最优切分维度 j；然后对该维度找到最优切分点 s 构成一个(j,s)对，并将输入空间划分为两个区域。然后在子区域中重复划分过程，直到满足停止条件为止。这样的回归树称为最小二乘回归树。

最小二乘回归树生成算法的描述如下。

① 输入。

■ 训练数据集 $\boldsymbol{D}$。

■ 停止计算条件。

② 输出：CART 回归树 $f(\boldsymbol{x})$。

③ 算法步骤。

■ 选择数据集 $\boldsymbol{D}$ 的最优切分维度 j 和切分点 s。即求解

$$\min_{j,s}\Big[\min_{c_1}\sum_{\boldsymbol{x}_i\in R_1(j,s)}(y_i-c_1)^2+\min_{c_2}\sum_{\boldsymbol{x}_i\in R_2(j,s)}(y_i-c_2)^2\Big]$$

求解方法为遍历 j、s，找到使上式最小的(j,s)对。

■ 用选定的(j,s)划分区域并决定相应的输出值。

$$R_1(j,s)=\{\boldsymbol{x}\mid x^{(j)}\leqslant s\}$$

$$R_2(j,s)=\{\boldsymbol{x} \mid \boldsymbol{x}^{(j)}>s\}$$
$$\hat{c}_1=\operatorname{ave}(y_i \mid \boldsymbol{x}_i \in R_1(j,s))$$
$$\hat{c}_2=\operatorname{ave}(y_i \mid \boldsymbol{x}_i \in R_2(j,s))$$

■ 对子区域 R_1、R_2 递归地调用上面两步，直到满足停止条件为止。

■ 将输入空间划分为 M 个区域 $R_1,R_2,\cdots,R_M$，生成决策树：

$$f(\boldsymbol{x})=\sum_{m=1}^{M}\hat{c}_m I(\boldsymbol{x} \in R_m)$$

④ 停止条件。

通常的停止条件为下列条件之一：

■ 节点中样本个数小于预定值。

■ 样本集的平方误差小于预定值。

■ 没有更多的特征。

(2) CART 分类树。

假设有 K 个分类，样本点属于第 k 类的概率为 $p_k=P(Y=c_k)$。定义概率分布的基尼系数为

$$\operatorname{Gini}(p)=\sum_{k=1}^{K}p_k(1-p_k)=1-\sum_{k=1}^{K}p_k^2 \tag{3.45}$$

对于给定的样本集合 $\boldsymbol{D}$，设属于类 c_k 的样本子集为 C_k，则基尼系数为

$$\operatorname{Gini}(\boldsymbol{D})=1-\sum_{k=1}^{K}\left(\frac{|C_k|}{|\boldsymbol{D}|}\right)^2 \tag{3.46}$$

给定特征 A，根据其是否取某一个可能值 α，样本集 $\boldsymbol{D}$ 被分为两个子集，分别为 $\boldsymbol{D}_1$ 和 $\boldsymbol{D}_2$，其中：

$$\begin{aligned}\boldsymbol{D}_1&=\{(\boldsymbol{x},y)\in \boldsymbol{D} \mid \boldsymbol{x}^{(A)}=\alpha\}\\ \boldsymbol{D}_2&=\{(\boldsymbol{x},y)\in \boldsymbol{D} \mid \boldsymbol{x}^{(A)}\neq\alpha\}=\boldsymbol{D}-\boldsymbol{D}_1\end{aligned} \tag{3.47}$$

定义 $\operatorname{Gini}(\boldsymbol{D},A)$ 如下：

$$\operatorname{Gini}(\boldsymbol{D},A)=\frac{|\boldsymbol{D}_1|}{|\boldsymbol{D}|}\operatorname{Gini}(\boldsymbol{D}_1)+\frac{|\boldsymbol{D}_2|}{|\boldsymbol{D}|}\operatorname{Gini}(\boldsymbol{D}_2) \tag{3.48}$$

它表示在特征 A 的条件下，集合 $\boldsymbol{D}$ 的基尼系数。

CART 分类树采用基尼系数选择最优特征。CART 分类树的生成算法描述如下。

① 输入。

■ 训练数据集 $\boldsymbol{D}$。

■ 停止计算条件。

②输出：CART 决策树。

③ 算法步骤。

■ 对每个特征 A，以及它可能的每个值 α，计算 $\operatorname{Gini}(\boldsymbol{D},A)$。

■ 选取最优特征和最优切分点。在所有特征 A 以及所有的切分点 α 中，基尼系数最小的 A 和 α 就是最优特征和最优切分点。根据最优特征和最优切分点将训练集 $\boldsymbol{D}$ 切分成两个子节点。

■ 对两个子节点递归调用上面两步，直到满足停止条件为止。

■ 最终生成 CART 决策树。

④ 停止条件。

通常的停止条件为下列条件之一：

■ 节点中样本个数小于预定值。

■ 样本集的基尼系数小于预定值。

■ 没有更多的特征。

2）决策树剪枝算法

CART 剪枝是从生成树开始剪掉一些子树，使得决策树变小。剪枝过程由如下两步组成(假设初始的生成树为 T_0)。

(1) 从 T_0 开始不断地剪枝，直到剪成一棵单节点的树。这些剪枝树形成一个剪枝树序列$\{T_0, T_1, \cdots, T_n\}$。

(2) 从该剪枝树序列中挑选出最优剪枝树。方法是通过交叉验证法使用验证数据集对剪枝树序列进行测试。

给出决策树的损失函数为 $C_\alpha(T)=C(T)+\alpha|T|$。其中 $C(T)$为决策树对训练数据的预测误差；$|T|$为决策树的叶节点个数。

对于固定的 α，存在使 $C_\alpha(T)$最小的树。令其为 T_α，可以证明 T_α 是唯一的。

■ 当 α 大时，T_α 偏小(即决策树比较简单)。

■ 当 α 小时，T_α 偏大(即决策树比较复杂)。

■ 当 $\alpha=0$ 时，生成树就是最优的。

■ 当 $\alpha=\infty$时，根组成的一个单节点树就是最优的。

考虑生成树 T_0。对 T_0 内任意节点 t，以 t 为单节点树(记作 $\tilde{t}$)的损失函数为 $C_\alpha()=C()+\alpha$，以 t 为根的子树 T_0 的损失函数为 $C_\alpha(\tilde{t})=C(\tilde{t})+\alpha$。可以证明：

■ 当 $\alpha=0$ 及充分小时，有 $C_\alpha(T_t)<C_\alpha(\tilde{t})$。

■ 当 α 增大到某个值时，有 $C_\alpha(T_t)=C_\alpha(\tilde{t})$。

■ 当 α 再增大时，有 $C_\alpha(T_t)>C_\alpha(\tilde{t})$。

因此，令 $\alpha=\dfrac{C(\tilde{t})-C(T_t)}{|T_t-1|}$，此时 T 与 $\tilde{t}$ 有相同的损失函数值，但是 $\tilde{t}$ 的叶节点更少。于是对 T_t 进行剪枝得到一棵单节点树 $\tilde{t}$。

对 T_0 内部的每个节点 t，定义 $g(t)=\dfrac{C(t)-C(T_t)}{|T_t|-1}$。设 T_0 内 $g(t)$最小的子树为 T_t^*，令该最小值的 $g(t)$为 $\tilde{\alpha}_1$。从 T_0 中剪去 T_t^*，即得到剪枝树 T_1。重复这种“求 $g(t)$− 剪枝”的过程，直到根节点即完成剪枝。在此过程中不断增加 $\tilde{\alpha}_i$ 的值，从而生成剪枝树序列。

CART 剪枝交叉验证过程是通过验证数据集来测试剪枝树序列$\{T_0, T_1, \cdots, T_n\}$中各剪枝树的。CART 回归树用于考查剪枝树的平方误差，平方误差最小的决策树被认为是最优决策树。CART 分类树用于考查剪枝树的基尼系数，基尼系数最小的决策树被认为是最优决策树。

CART 剪枝算法的描述如下。

(1) 输入：CART 生成树 T_0。

（2）输出：CART 剪枝树 T_α。

（3）算法步骤。

■ 令 $k=0, T=T_0, \alpha=\infty$。

■ 自下而上地对树 T 各内部节点 t 计算 $g(t)=\dfrac{C(t)-C(T_t)}{|T_t-1|}$。

■ 对于所有的内部节点，$\tilde{\alpha}_{k+1}=\min\limits_t(g(t))$，令 $t^*=\operatorname*{argmin}\limits_t(g(t))$。对内部节点 t^* 进行剪枝得到树 T_{k+1}。

■ 令 $T=T_{k+1}, k=k+1$。

■ 若 T 不是由根节点单独构成的树，则继续前面的步骤。

■ 采用交叉验证法在剪枝树序列 $T_0, T_1, \cdots, T_n$ 中选取最优剪枝树 T_α。

3.3.3 实战

1. 数据集

为了检验决策树的性能，本节选用经典的数据集：鸢尾花数据集，该数据集的详细介绍见 2.1.4 节。

2. Sklearn 实现

DecisionTreeClassifier()函数实现了分类决策树，用于分类问题。其原型如下：

```
class sklearn.tree.DecisionTreeClassifier( * , criterion = 'gini', splitter = 'best', max_depth =
None, min_samples_split = 2, min_samples_leaf = 1, min_weight_fraction_leaf = 0.0, max_
features = None, random_state = None, max_leaf_nodes = None, min_impurity_decrease = 0.0, class_
weight = None, ccp_alpha = 0.0)
```

1）参数

（1）criterion：一个字符串，用于指定切分质量的评价准则。可以有以下取值。

■ 'gini'：表示切分时评价准则是基尼系数。

■ 'entropy'：表示切分时评价准则是熵。

（2）splitter：一个字符串，用于指定切分原则，可以有以下取值。

■ 'best'：表示选择最优的切分。

■ 'random'：表示随机切分。

（3）max_depth：取值可以为整数或者 None，用于指定树的最大深度。可以有以下取值。

■ 如果为 None，则表示树的深度不限（直到每个叶子都是纯的，即叶节点中所有的样本点都属于一个类，或者叶子中包含小于 min_samples_split 个样本点）。

■ 如果 max_leaf_nodes 为非 None 值，则忽略此选项。

（4）min_samples_split：取值为整数，用于指定每个内部节点（非叶节点）包含的最少的样本数。

（5）min_samples_leaf：取值为整数，用于指定每个叶节点包含的最少的样本数。

（6）min_weight_fraction_leaf：取值为浮点数，用于指定叶节点中样本的最小权重系数。

(7) max_features：取值可以为整数、浮点数、字符串或者 None，用于指定寻找 best split 时考虑的特征数量。可以有以下取值。

■ 如果是整数，则每次切分只考虑 max_features 个特征。

■ 如果是浮点数，则每次切分只考虑 max_features×n_features 个特征(max_features 指定了百分比)。

■ 如果是字符串'auto'或者'sqrt'，则 max_features＝sqrt(n_features)。如果是字符串'log2'，则 max_features＝log2(n_features)。

■ 如果是 None，则 max_features＝n_features。

(8) random_state：取值为一个整数或一个 RandomState 实例或 None。

■ 如果为整数，则指定了随机数生成器的种子。

■ 如果为 RandomState 实例，则指定了随机数生成器。

■ 如果为 None，则使用默认的随机数生成器。

(9) max_leaf_nodes：取值为整数或者 None，用于指定最大的叶节点数量。

■ 如果为 None，此时叶节点数量不限。

■ 如果为非 None，则 max_depth 被忽略。

(10) class_weight：取值为一个字典、字典的列表、字符串'balanced'，或者 None，用于指定分类的权重。权重的形式为{class_label：weight}。

■ 如果为 None，则每个分类的权重都为 1。

■ 字符串'balanced'表示分类的权重是样本中各分类出现的频率的反比。

■ 如果 sample_weight 提供了权重(由 fit()方法提供)，则这些权重都会乘以 sample_weight。

2) 属性

(1) classes_：分类的标签值。

(2) feature_importances_：给出了特征的重要程度。该值越高，则该特征越重要(也称为 Gini importance)。

(3) max_features_：max_features 的推断值。

(4) n_classes_：给出了分类的数量。

(5) n_features_：执行 fit()方法之后，特征的数量。

(6) n_outputs_：执行 fit()方法之后，输出的数量。

(7) tree_：一个 Tree 对象，即底层的决策树。

3) 方法

(1) fit(X,y[,sample_weight,check_input,…])：训练模型。

(2) predict(X[,check_input])：用模型进行预测，返回预测值。

(3) predict_log_proba(X)：返回一个数组，数组的元素依次是 X 预测为各个类别的概率的对数值。

(4) predict_proba(X)：返回一个数组，数组的元素依次是 X 预测为各个类别的概率值。

(5) score(X,y[,sample_weight]):返回在(X,y)上预测的准确率。

首先导入包:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn import model_selection
from sklearn.tree import DecisionTreeClassifier
```

定义 DecisionTreeClassifier() 函数:

```
def test_DecisionTreeClassifier( * data):
  '''
测试 DecisionTreeClassifier()函数的用法

  :param data: 可变参数.它是一个元组,这里要求其元素依次为训练样本集、测试样本集、训练样本
的标记、测试样本的标记
  :return: None
  '''
  X_train, X_test, y_train, y_test = data
  clf = DecisionTreeClassifier()
  clf.fit(X_train, y_train)
  print("Training score: %f" % (clf.score(X_train, y_train)))
  print("Testing score: %f" % (clf.score(X_test, y_test)))
```

调入数据后,使用 DecisionTreeClassifier() 函数进行分类:

```
# 产生用于分类问题的数据集
X_train, X_test, y_train, y_test = load_data()

# 调用 test_DecisionTreeClassifier()函数
test_DecisionTreeClassifier(X_train, X_test, y_train, y_test)
```

结果如下:

```
Training score:1.000000
Testing score:0.974359
```

可以看出,实现了对训练数据集的完全拟合,对测试数据集的拟合精度高达97.435 9%。下面考察评价切分质量的评价准则 criterion 对分类性能的影响,给出如下函数:

```
def test_DecisionTreeClassifier_criterion( * data):
  '''
    测试 DecisionTreeClassifier()函数的预测性能随 criterion 参数的影响

  :param data: 可变参数.它是一个元组,这里要求其元素依次为训练样本集、测试样本集、训练样本
的标记、测试样本的标记
  :return: None
  '''
  X_train, X_test, y_train, y_test = data
```

```
    criterions = ['gini', 'entropy']
    for criterion in criterions:
        clf = DecisionTreeClassifier(criterion = criterion)
        clf.fit(X_train, y_train)
        print("criterion: %s" % criterion)
        print("Training score: %f" % (clf.score(X_train, y_train)))
        print("Testing score: %f" % (clf.score(X_test, y_test)))
```

使用该函数进行测试：

```
# 产生用于分类问题的数据集
X_train, X_test, y_train, y_test = load_data()
test_DecisionTreeClassifier_criterion(X_train,X_test,y_train,y_test)
```

结果如下：

```
criterion:gini
Training score:1.000000
Testing score:0.974359
criterion:entropy
Training score: 1.000000
Testing score:0.948718
```

可以看出，对于本问题，二者对训练集的拟合都非常完美(100%)，对于测试集的预测准确率都较高，但是稍有不同；使用基尼系数的策略预测性能较高。

接下来检验随机划分与最优划分的影响，给出如下函数：

```
def test_DecisionTreeClassifier_splitter( *data):
    '''
测试 DecisionTreeClassifier()函数的预测性能随划分类型的影响

    :param data: 可变参数.它是一个元组,这里要求其元素依次为训练样本集、测试样本集、训练样本的标记、测试样本的标记
    :return: None
    '''
    X_train, X_test, y_train, y_test = data
    splitters = ['best', 'random']
    for splitter in splitters:
        clf = DecisionTreeClassifier(splitter = splitter)
        clf.fit(X_train, y_train)
        print("splitter: %s" % splitter)
        print("Training score: %f" % (clf.score(X_train, y_train)))
        print("Testing score: %f" % (clf.score(X_test, y_test)))
```

使用该函数：

```
# 产生用于分类问题的数据集
X_train, X_test, y_train, y_test = load_data()
test_DecisionTreeClassifier_splitter(X_train,X_test,y_train,y_test)
```

结果如下：

```
splitter:best
Training score:1.000000
Testing score:0.974359
splitter:random
Training score:1.008000
Testing score:0.948718
```

可以看出,对于本问题,二者对训练集的拟合都非常完美(100%),对于测试集的预测准确率都较高,但是稍有不同;使用最优划分的性能要高于随机划分。

最后考察决策树深度的影响。决策树的深度对应着树的复杂度。决策树越深,则模型越复杂,给出如下函数:

```
def test_DecisionTreeClassifier_depth( * data, maxdepth):
  '''
  测试 DecisionTreeClassifier()函数的预测性能随 maxdepth 参数的影响

  :param data: 可变参数.它是一个元组,这里要求其元素依次为训练样本集、测试样本集、训练样本
的标记、测试样本的标记
  :param maxdepth: 一个整数,用于指定 DecisionTreeClassifier()函数的 max_depth 参数
  :return: None
  '''
  X_train, X_test, y_train, y_test = data
  depths = np.arange(1, maxdepth)
  training_scores = []
  testing_scores = []
  for depth in depths:
     clf = DecisionTreeClassifier(max_depth = depth)
     clf.fit(X_train, y_train)
     training_scores.append(clf.score(X_train, y_train))
     testing_scores.append(clf.score(X_test, y_test))
## 绘图
  fig = plt.figure()
  ax = fig.add_subplot(1, 1, 1)
  ax.plot(depths, training_scores, label = "traing score", marker = 'o')
  ax.plot(depths, testing_scores, label = "testing score", marker = ' * ')
  ax.set_xlabel("maxdepth")
  ax.set_ylabel("score")
  ax.set_title("Decision Tree Classification")
  ax.legend(framealpha = 0.5, loc = 'best')
plt.show()
```

使用该函数:

```
# 产生用于分类问题的数据集
X_train, X_test, y_train, y_test = load_data()
test_DecisionTreeClassifier_depth(X_train,X_test,y_train,y_test,maxdepth = 100)
```

运行结果如图 3.13 所示。可以看出,随着树深度的增加(对应着模型复杂度的提高),模型对训练集和预测集的拟合度都在提高。这里训练数据集大小仅为 150,不考虑任何条件,只需要一棵深度为$\log_2 150 \leqslant 8$ 的二叉树就能够完全拟合数据,使得每个叶节点最多只有一个样本。考虑到决策树算法中的提前终止条件(如叶节点中所有样本都是同一类则不再划分,此时叶节点中有超过一个样本),则树的深度小于 8。

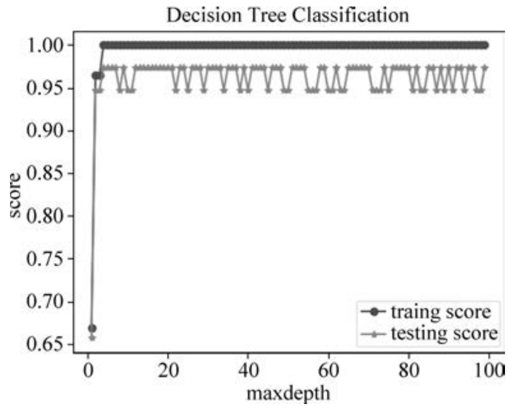


图 3.13　调用 DecisionTreeClassifier_depth()函数的运行结果

3. 算法实现

下面就以一个经典的打球的例子来说明如何构建决策树。是否去打球(play)主要由天气(outlook)、温度(temperature)、湿度(humidity)、是否有风(windy)来确定。样本中共 14 条数据。

表 3.2　打球数据集示例

序号	outlook	temperature	humidity	windy	play
1	sunny	hot	high	FALSE	no
2	sunny	hot	high	TRUE	no
3	overcast	hot	high	FALSE	yes
4	rainy	mild	high	FALSE	yes
5	rainy	cool	normal	FALSE	yes
6	rainy	cool	normal	TRUE	no
7	overcast	cool	normal	TRUE	yes
8	sunny	mild	high	FALSE	no
9	sunny	cool	normal	FALSE	yes
10	rainy	mild	normal	FALSE	yes
11	sunny	mild	normal	TRUE	yes
12	overcast	mild	high	TRUE	yes
13	overcast	hot	normal	FALSE	yes
14	rainy	mild	high	TRUE	no

下面将分别介绍使用 ID3 算法和 C4.5 算法构建决策树的方法。

1) 使用 ID3 算法构建决策树

ID3 算法是使用信息增益来选择特征的。

(1) 计算 play(是否去打球)的经验熵。

在本例中,目标变量 $\boldsymbol{D}$ 就是 play(是否去打球),即 yes(打球)和 no(不打球)。$|\boldsymbol{D}|=14$。K 就是目标变量 play(是否去打球)的分类数,有两类:yes(打球)和 no(不打球)。yes(打球)这个分类下有 9 个样本,而 no(不打球)这个分类下有 5 个样本,所以信息熵 $H(\boldsymbol{D})=H(\text{play})=-((9/14)\log(9/14)+(5/14)\log(5/14))=0.651\ 756\ 561\ 173$。

注意：本书中的 $\log x$ 均按照以 e 为底进行计算的。

(2) 计算 outlook(天气)特征的信息增益。

记 outlook 特征为特征 A，共有 3 个不同的取值{sunny、overcast、rainy}，即 $v=3$，根据特征 A 的取值，将数据集 $\boldsymbol{D}$ 划分为如下 3 个子集。

■ sunny 的子集：共有 5 个样本，2 个打球(play=yes)，3 个不打球(play=no)。

■ overcast 的子集：共有 4 个样本，均为打球(play=yes)。

■ rainy 的子集：共有 5 个样本，3 个打球(play=yes)，2 个不打球(play=no)。

每个子集可以分别计算熵，具体公式如下：

$H(\boldsymbol{D}|A)=H(\text{play}|\text{outlook})=(5/14)\text{sunny}$ 熵 $+(4/14)\text{overcast}$ 熵 $+(5/14)\text{rainy}$ 熵 $=(5/14)(-(2/5)\log(2/5)-(3/5)\log(3/5))+(4/14)(-(4/4)\log(4/4))+(5/14)(-(3/5)\log(3/5)-(2/5)\log(2/5))=0.480\ 722\ 619\ 292$

所以 outlook 特征的信息增益为 $g(\boldsymbol{D},A)=g(\boldsymbol{D},\text{outlook})=H(\boldsymbol{D})-H(\boldsymbol{D}|A)=0.171\ 033\ 941\ 88$。

(3) 计算 temperature(温度)特征的信息增益。

记 temperature 特征为特征 B，共有 3 个不同的取值{hot，mild，cool}，则 $v=3$。根据特征的取值，将数据集 $\boldsymbol{D}$ 划分为如下 3 个子集。

■ hot 的子集：共有 4 个样本，2 个打球(play=yes)，2 个不打球(play=no)。

■ mild 的子集：共有 6 个样本，4 个打球(play=yes)，2 个不打球(play=no)。

■ cool 的子集：共有 4 个样本，3 个打球(play=yes)，1 个不打球(play=no)。

每个子集可以分别计算熵，具体公式如下：

$H(\boldsymbol{D}|B)=H(\text{play}|\text{temperature})=(4/14)\text{hot}$ 熵 $+(6/14)\text{mild}$ 熵 $+(4/14)\text{cool}$ 熵 $=(4/14)(-(2/4)\log(2/4)-(2/4)\log(2/4))+(6/14)(-(4/6)\log(4/6)-(2/6)\log(2/6))+(4/14)(-(3/4)\log(3/4)-(1/4)\log(1/4))=0.631\ 501\ 022\ 177$

所以 temperature 特征的信息增益为 $g(\boldsymbol{D},B)=g(\boldsymbol{D},\text{temperature})=H(\boldsymbol{D})-H(\boldsymbol{D}|B)=0.020\ 255\ 538\ 995\ 2$。

(4) 计算 humidity(湿度)特征的信息增益。

记 humidity 特征为特征 C，共有两个不同的取值{high，normal}，$v=2$。根据特征的取值将数据集 $\boldsymbol{D}$ 划分为如下两个子集：

high 的子集：共有 7 个样本，3 个打球(play=yes)，4 个不打球(play=no)。

normal 的子集：共有 7 个样本，6 个打球(play=yes)，1 个不打球(play=no)。

每个子集可以分别计算熵，具体公式如下：

$H(\boldsymbol{D}|C)=H(\text{play}|\text{humidity})=(7/14)\text{high}$ 熵 $+(7/14)\text{normal}$ 熵 $=(7/14)(-(3/7)\cdot\log(3/7)-(4/7)\log(4/7))+(7/14)(-(6/7)\log(6/7)-(1/7)\log(1/7))=0.546\ 512\ 211\ 494$

所以 humidity 特征的信息增益为 $g(\boldsymbol{D},C)=g(\boldsymbol{D},\text{humidity})=H(\boldsymbol{D})-H(\boldsymbol{D}|C)=0.105\ 244\ 349\ 678$。

(5) 计算 windy(是否有风)特征的信息增益。

记 windy 特征为特征 E，共有两个不同的取值{TRUE，FALSE}，$v=2$。根据特征的取值，将数据集 $\boldsymbol{D}$ 划分为如下两个子集：

TRUE 的子集：共有 6 个样本，2 个打球(play=yes)，4 个不打球(play=no)。

FALSE 的子集：共有 8 个样本，6 个打球(play=yes)，2 个不打球(play=no)。

每个子集可以分别计算熵，具体公式如下：

$H(\boldsymbol{D}|E)=H(\text{play}|\text{windy})=(6/14)$TRUE 熵$+(8/14)$FALSE 熵$=(6/14)(-(2/6)\cdot\log(2/6)-(4/6)\log(4/6))+(8/14)(-(6/8)\log(6/8)-(2/8)\log(2/8))=0.594\ 126\ 154\ 766$

所以 windy 特征的信息增益为 $g(\boldsymbol{D},E)=g(\boldsymbol{D},\text{windy})=H(\boldsymbol{D})-H(\boldsymbol{D}|E)=0.057\ 630\ 406\ 407$。

(6) 确定 root 节点。

对比上面四个特征的信息增益如下：

$$g(\boldsymbol{D},A)=g(\boldsymbol{D},\text{outlook})=0.171\ 033\ 941\ 88$$
$$g(\boldsymbol{D},B)=g(\boldsymbol{D},\text{temperature})=0.020\ 255\ 538\ 995\ 2$$
$$g(\boldsymbol{D},C)=g(\boldsymbol{D},\text{humidity})=0.105\ 244\ 349\ 678$$
$$g(\boldsymbol{D},E)=g(\boldsymbol{D},\text{windy})=0.057\ 630\ 406\ 407$$

可以看出 outlook 特征的信息增益最大，所以选择 outlook 特征作为决策树的根节点。

特征 A(天气)有三个不同的取值{sunny、overcast、rainy}，即 $v=3$，根据特征 A(天气)的取值，将数据集 $\boldsymbol{D}$ 划分为 3 个子集，其中：

■ sunny 的子集中有 5 个样本：2 个打球(play=yes)，3 个不打球(play=no)。

■ overcast 的子集中有 4 个样本：都为打球(play=yes)。

■ rainy 的子集中有 5 个样本：3 个打球(play=yes)，2 个不打球(play=no)。

对每个子集分别计算熵如下：

$$\text{sunny 熵}=-(2/5)\log(2/5)-(3/5)\log(3/5)>0$$
$$\text{overcast 熵}=-(4/4)\log(4/4)=0$$
$$\text{rainy 熵}=-(3/5)\log(3/5)-(2/5)\log(2/5)>0$$

上面的 overcast 熵=0，如图 3.14 所示。也就是这部分已经分好类了，都为打球，所以直接就可以作为叶节点了，不需要再进行分类。而 sunny 熵、rainy 熵都大于 0，还需要按照上面根节点的选择方式继续选择特征。

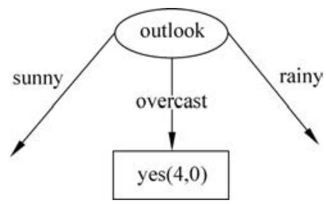


图 3.14 overcast 熵=0

(7) 计算 outlook 特征为 sunny 的数据集，该数据集如表 3.3 所示。

表 3.3 outlook 特征为 sunny 的数据集

序号	outlook	temperature	humidity	windy	play
1	sunny	hot	high	FALSE	no
2	sunny	hot	high	TRUE	no
8	sunny	mild	high	FALSE	no
9	sunny	cool	normal	FALSE	yes
11	sunny	mild	normal	TRUE	yes

① 计算 outlook 这个分支样本的信息熵。

yes(打球)这个分类下有 2 个样本，而 no(不打球)这个分类下有 3 个样本，所以信息熵 $H(\boldsymbol{D})=H(\text{play})=-((2/5)\log(2/5)+(3/5)\log(3/5))=0.673\ 011\ 667\ 009$。

② 计算 temperature 特征的信息增益。

记 temperature 特征为特征 A，其共有 3 个不同的取值{hot，mild，cool}，则 $v=3$。根据特征的取值，将数据集 $\boldsymbol{D}$ 划分为 3 个子集。其中：

■ hot 的子集中有 2 个样本，2 个不打球(play＝no)。

■ mild 的子集中有个 2 样本，1 个打球(play＝yes)，1 个不打球(play＝no)。

■ cool 的子集中有 1 个样本，1 个打球(play＝yes)。

每个子集可以分别计算熵，具体公式如下：

$$
\begin{aligned}
H(\boldsymbol{D}|A) &= H(\text{play}|\text{temperature}) = (2/5)\text{hot 熵} + (2/5)\text{mild 熵} + (1/5)\text{cool 熵} \\
&= (2/5)(-(2/2)\log(2/2)) + (2/5)(-(1/2)\log(1/2) - \\
&\quad (1/2)\log(1/2)) + (1/5)(-(1/1)\log(1/1)) = -(2/5)\log(1/2) \\
&= 0.277\ 258\ 872\ 224
\end{aligned}
$$

所以 temperature 特征对应的信息增益为

$g(\boldsymbol{D},A)=g(\boldsymbol{D},\text{temperature})=H(\boldsymbol{D})-H(\boldsymbol{D}\mid A)=0.395\ 752\ 794\ 785$。

③ 计算 humidity 特征的信息增益。

记 humidity 特征为特征 B，共有两个不同的取值{high，normal}，$v=2$。根据特征的取值将数据集 $\boldsymbol{D}$ 划分为两个子集，其中：

■ high 的子集有 3 个样本，3 个不打球(play＝no)。

■ normal 的子集有 2 个样本，2 个打球(play＝yes)。

每个子集可以分别计算熵，具体公式如下：

$$
\begin{aligned}
H(\boldsymbol{D}|B) &= H(\text{play}|\text{humidity}) = (3/5)\text{high 熵} + (2/5)\text{normal 熵} \\
&= (3/5)(-(3/3)\log(3/3)) + (2/5)(-(2/2)\log(2/2)) = 0
\end{aligned}
$$

所以 humidity 特征的信息增益为

$$g(\boldsymbol{D},B)=g(\boldsymbol{D},\text{humidity})=H(\boldsymbol{D})-H(\boldsymbol{D}\mid B)=0.673\ 011\ 667\ 009$$

humidity 特征划分已经将信息熵降为 0，所以就不用继续计算了，直接把湿度作为分裂的特征即可，如图 3.15 所示。

(8) 计算 outlook 特征为 rainy 的数据。

用相同的方法计算此部分的数据后，最终得出的决策树如图 3.16 所示。

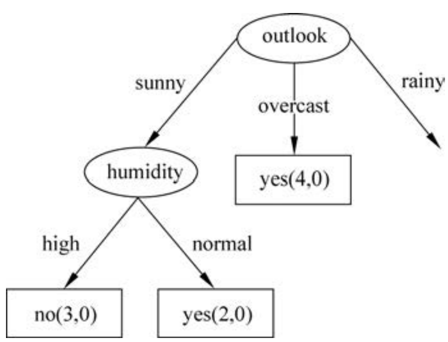


图 3.15　humidity 作为分裂的特征

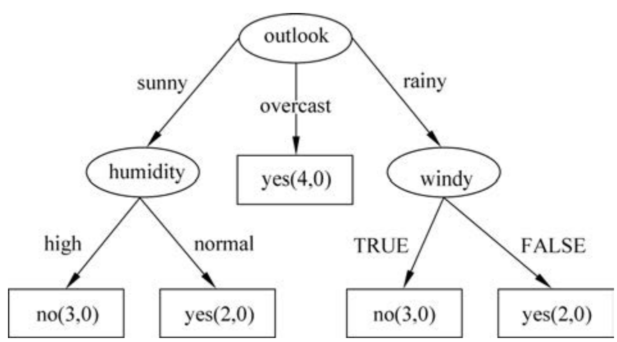


图 3.16　windy 作为分裂的特征

下面的代码是基于 ID3 算法的信息增益来实现的。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

```
# @Time  : 2022/5/12 20:04
# @Author :
# @File  : id3.py
# @Description :

from math import log
import operator
import treePlotter


def createDataSet():
  dataset = [[1, 1, 'yes'],
          [1, 1, 'yes'],
          [1, 0, 'no'],
          [0, 1, 'no'],
          [0, 1, 'no']]
  labels = ['no surfacing', 'flippers']
  return dataset, labels


def calcshannonent(dataset):
  """
  计算信息熵

  @param dataset:
  @return:
  """
  numEntries = len(dataset) # 样本数
  labelCounts = {} # 创建一个数据字典:key是最后一列的数值(即标签,也就是目标分类的类
# 别),value是属于该类别的样本个数
  for featVec in dataset: # 遍历整个数据集,每次取一行
    currentLabel = featVec[-1] # 取该行最后一列的值
    if currentLabel not in labelCounts.keys(): labelCounts[currentLabel] = 0
    labelCounts[currentLabel] += 1
  shannonEnt = 0.0 # 初始化信息熵
  for key in labelCounts:
    prob = float(labelCounts[key]) / numEntries
    shannonEnt -= prob * log(prob, 2) # log base 2 计算信息熵
  return shannonEnt


def splitDataSet(dataset, axis, value):
  """
  按给定的特征划分数据

  @param dataset:
  @param axis: 是dataset数据集下要进行特征划分的列号,如outlook是0列
  @param value: 是该列下某个特征值,如0列中的sunny
  @return:
  """
  retDataSet = []
  for featVec in dataset: # 遍历数据集,并抽取按axis的当前value特征进行划分的数据集(不
包括axis列的值)
```

```
        if featVec[axis] == value: # 判断 featVec 数组的第 axis 个是否等于 value
            reducedFeatVec = featVec[:axis] # 从 0 到 axis 进行分片
            reducedFeatVec.extend(featVec[axis + 1:])
            retDataSet.append(reducedFeatVec)
            # print axis,value,reducedFeatVec
    # print retDataSet
    return retDataSet


def chooseBestFeatureToSplit(dataset):
    """
    选取当前数据集下,用于划分数据集的最优特征

    @param dataset:
    @return:
    """
    numFeatures = len(dataset[0]) - 1      # 获取当前数据集的特征个数,最后一列是分类标签
    baseEntropy = calcshannonent(dataset)  # 计算当前数据集的信息熵
    bestInfoGain = 0.0;
    bestFeature = -1                       # 初始化最优信息增益和最优的特征
    for i in range(numFeatures):           # 遍历每个特征 iterate over all the features
        featList = [example[i] for example in dataset]   # 获取数据集中当前特征下的所有值
        uniqueVals = set(featList) # 获取当前特征值,如 outlook 下有 sunny、overcast、rainy
        newEntropy = 0.0
        for value in uniqueVals:           # 计算每种划分方式的信息熵
            subDataSet = splitDataSet(dataset, i, value)
            prob = len(subDataSet) / float(len(dataset))
            newEntropy += prob * calcshannonent(subDataSet)
        infoGain = baseEntropy - newEntropy # 计算信息增益
        if (infoGain > bestInfoGain):      # 比较每个特征的信息增益,只要最好的信息增益
            bestInfoGain = infoGain        # 如果此值比当前最优值要好,则将最优值更换为此值
            bestFeature = i
    return bestFeature                     # 返回一个整数


  def majorityCnt(classList):
    """
    该函数使用分类名称的列表,然后创建键值为 classList 中唯一值的数据字典.
    字典对象存储了 classList 中每个类标签出现的频率.
    最后利用 operator 操作键值排序字典,
    并返回出现次数最多的分类名称

    @param classList:
    @return:
    """
    classcount = {}
    for vote in classList:
        if vote not in classcount.keys(): classcount[vote] = 0
        classcount[vote] += 1
    sortedclasscount = sorted(classcount.iteritems(), key=operator.itemgetter(1), reverse=
True)
    return sortedclasscount[0][0]
```

```
  def createTree(dataset, labels):
  """
  生成决策树的主方法

  @param dataset:
  @param labels:
  @return:
  """
  classList = [example[-1] for example in dataset] # 返回当前数据集下标签列的所有值
  if classList.count(classList[0]) == len(classList):
     return classList[0]   # 当类别完全相同时则停止继续划分,直接返回该类的标签
  if len(dataset[0]) == 1:# 遍历完所有的特征时,仍然不能将数据集划分成仅包含唯一类别的
#分组 dataset
     return majorityCnt(classList) # 由于无法简单地返回唯一的类标签,这里就将返回出现次数
#最多的类别作为返回值
  bestFeat = chooseBestFeatureToSplit(dataset) # 获取最好的分类特征索引
  bestFeatLabel = labels[bestFeat] # 获取该特征的名字

  # 这里直接使用字典变量来存储树信息,这对于绘制树形图很重要
  myTree = {bestFeatLabel:{}} # 当前数据集选取最好的特征存储在 bestFeat 中
  del (labels[bestFeat])       # 删除已经在选取的特征
  featValues = [example[bestFeat] for example in dataset]
  uniqueVals = set(featValues)
  for value in uniqueVals:
     subLabels = labels[:] # 复制所有标签,这样生成树就不会弄乱存在的对应的标签
     myTree[bestFeatLabel][value] = createTree(splitDataSet(dataset, bestFeat, value),
subLabels)
  return myTree


def classify(inputTree, featLabels, testVec):
  firstStr = inputTree.keys()[0]
  secondDict = inputTree[firstStr]
  featIndex = featLabels.index(firstStr)
  key = testVec[featIndex]
  valueOfFeat = secondDict[key]
  if isinstance(valueOfFeat, dict):
     classLabel = classify(valueOfFeat, featLabels, testVec)
  else:
     classLabel = valueOfFeat
  return classLabel


def storeTree(inputTree, filename):
  import pickle
  fw = open(filename, 'w')
  pickle.dump(inputTree, fw)
  fw.close()


def grabTree(filename):
  import pickle
```

```
    fr = open(filename)
    return pickle.load(fr)


if __name__ == '__main__':
    fr = open('play.txt')
    lenses = [inst.strip().split(' ') for inst in fr.readlines()]
    print(lenses)
    lensesLabels = ['outlook', 'temperature', 'huminidy', 'windy']
    lensesTree = createTree(lenses, lensesLabels)
    treePlotter.createPlot(lensesTree)

#!/usr/bin/env python
# -*- coding: utf-8 -*-
# @Time   : 2022/5/12 20:04
# @Author :
# @File   : treePlotter.py
# @Description :

import matplotlib.pyplot as plt

decisionNode = dict(boxstyle="sawtooth", fc="0.8")  # 定义文本框与箭头的格式
leafNode = dict(boxstyle="round4", fc="0.8")
arrow_args = dict(arrowstyle="<-")


def getNumLeafs(myTree):
    """
    获取树叶节点的数目
    @param myTree:
    @return:
    """
    numLeafs = 0
    firstStr = list(myTree.keys())[0]
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
# 测试节点的数据类型是不是字典,如果是则需要递归地调用getNumLeafs()函数
        if type(secondDict[key]).__name__ == 'dict':
            numLeafs += getNumLeafs(secondDict[key])
        else:
            numLeafs += 1
    return numLeafs


def getTreeDepth(myTree):
    """
    获取树的深度

    @param myTree:
    @return:
    """
    maxDepth = 0
    firstStr = list(myTree.keys())[0]
```

```
    secondDict = myTree[firstStr]
    for key in secondDict.keys():
        if type(secondDict[
                key]).__name__ == 'dict': # 如果不是叶节点则测试该节点是否是字典类型
            thisDepth = 1 + getTreeDepth(secondDict[key])
        else:
            thisDepth = 1
        if thisDepth > maxDepth: maxDepth = thisDepth
    return maxDepth


# 绘制带箭头的注释
def plotNode(nodeTxt, centerPt, parentPt, nodeType):
    createPlot.ax1.annotate(nodeTxt, xy = parentPt, xycoords = 'axes fraction',
                            xytext = centerPt, textcoords = 'axes fraction',
                            va = "center", ha = "center", bbox = nodeType, arrowprops = arrow_args)

# 计算父节点和子节点的中间位置,在父节点间填充文本的信息
def plotMidText(cntrPt, parentPt, txtString):
    xMid = (parentPt[0] - cntrPt[0]) / 2.0 + cntrPt[0]
    yMid = (parentPt[1] - cntrPt[1]) / 2.0 + cntrPt[1]
    createPlot.ax1.text(xMid, yMid, txtString, va = "center", ha = "center", rotation = 30)

# 画决策树的准备方法
def plotTree(myTree, parentPt, nodeTxt): # 从第一个关键节点开始拆分
    numLeafs = getNumLeafs(myTree)       # 计算树的宽度
    depth = getTreeDepth(myTree)         # 计算树的深度
    firstStr = list(myTree.keys())[0]
    cntrPt = (plotTree.xOff + (1.0 + float(numLeafs)) / 2.0 / plotTree.totalW, plotTree.
yOff)
    plotMidText(cntrPt, parentPt, nodeTxt)
    plotNode(firstStr, cntrPt, parentPt, decisionNode)
    secondDict = myTree[firstStr]
    plotTree.yOff = plotTree.yOff - 1.0 / plotTree.totalD
    for key in secondDict.keys():
        if type(secondDict[
                key]).__name__ == 'dict':
            plotTree(secondDict[key], cntrPt, str(key)) # 递归
        else: # 如果是叶节点则打印
            plotTree.xOff = plotTree.xOff + 1.0 / plotTree.totalW
            plotNode(secondDict[key], (plotTree.xOff, plotTree.yOff), cntrPt, leafNode)
            plotMidText((plotTree.xOff, plotTree.yOff), cntrPt, str(key))
    plotTree.yOff = plotTree.yOff + 1.0 / plotTree.totalD

        # 画决策树的主方法
def createPlot(inTree):
    fig = plt.figure(1, facecolor = 'white')
    fig.clf()
    axprops = dict(xticks = [], yticks = [])
    createPlot.ax1 = plt.subplot(111, frameon = False, **axprops)
    # createPlot.ax1 = plt.subplot(111, frameon = False) # 用于演示目的的记号
    plotTree.totalW = float(getNumLeafs(inTree))
```

```
    plotTree.totalD = float(getTreeDepth(inTree))
    plotTree.xOff = -0.5 / plotTree.totalW;
    plotTree.yOff = 1.0;
    plotTree(inTree, (0.5, 1.0), '')
    plt.show()

  # def createPlot():
  #   fig = plt.figure(1, facecolor='white')
  #   fig.clf()
  #   createPlot.ax1 = plt.subplot(111, frameon=False) # 用于演示目的的记号
  #   plotNode('a decision node', (0.5, 0.1), (0.1, 0.5), decisionNode)
  #   plotNode('a leaf node', (0.8, 0.1), (0.3, 0.8), leafNode)
  #   plt.show()

  def retrieveTree(i):
    listOfTrees = [{'no surfacing': {0: 'no', 1: {'flippers': {0: 'no', 1: 'yes'}}}},
               {'no surfacing': {0: 'no', 1: {'flippers': {0: {'head': {0: 'no', 1: 'yes'}}, 1: 'no'}}}}
               ]
    return listOfTrees[i]
```

执行上述代码后画出的决策树如图 3.17 所示。

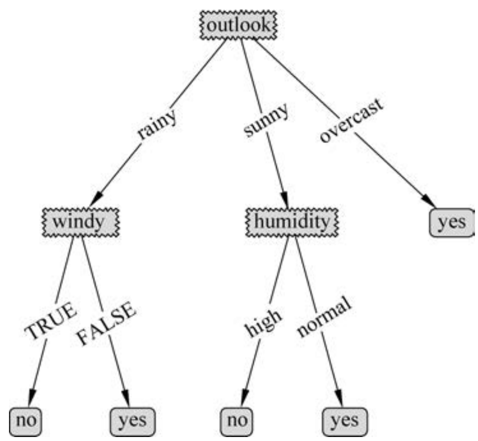


图 3.17 运行程序后画出的决策树

2）使用 C4.5 算法构建决策树

C4.5 算法使用信息增益率来进行特征选择。由于前面 ID3 算法使用信息增益选择分裂属性的方式会倾向于选择具有大量值的特征，如对于 no，每条数据都对应一个 play 值，即按此特征划分，每个划分都是纯的（即完全的划分，只有属于一个类别），no 的信息增益为最大值 1，但这种按该特征的每个值进行分类的方式是没有任何意义的。为了解决这一弊端，有人提出了采用信息增益率（GainRate）来选择分裂特征。计算方式如下：

$$\mathrm{gr}(\boldsymbol{D},A)=g(\boldsymbol{D},A)/H(A)$$

其中，$g(\boldsymbol{D},A)$就是 ID3 算法中的新增增益。

计算各特征的信息增益率如下。

（1）outlook 特征的信息增益率。

$H(A)=H(\text{outlook})=-(5/14)\log(5/14)-(5/14)\log(5/14)-(4/14)\log(4/14)=1.093\ 374\ 717\ 56$

而上面已经计算了

$$g(\boldsymbol{D},A)=g(\boldsymbol{D},\text{outlook})=0.171\ 033\ 941\ 88$$

$$\mathrm{gr}(\boldsymbol{D},A)=\mathrm{gr}(\text{play},\text{outlook})=g(\boldsymbol{D},A)/H(A)=0.156\ 427\ 562\ 421$$

（2）temperature 特征的信息增益率。

$H(B)=H(\text{temperature})=-(4/14)\log(4/14)-(6/14)\log(6/14)-(4/14)\log(4/14)=$

1.078 992 207 88

而上面已经计算了

$$g(\boldsymbol{D},B)=g(\boldsymbol{D},\text{temperature})=0.020\,255\,538\,995\,2$$

$$\text{gr}(\boldsymbol{D},B)=\text{gr}(\text{play},\text{temperature})=g(\boldsymbol{D},B)/H(B)=0.018\,772\,646\,222\,4$$

(3) humidity 特征的信息增益率。

$$H(C)=H(\text{humidity})=-(7/14)\log(7/14)-(7/14)\log(7/14)=0.693\,147\,180\,56$$

而上面已经计算了

$$g(\boldsymbol{D},C)=g(\boldsymbol{D},\text{humidity})=0.105\,244\,349\,678$$

$$\text{gr}(\boldsymbol{D},C)=\text{gr}(\text{play},\text{humidity})=g(\boldsymbol{D},C)/H(C)=0.151\,835\,501\,362$$

(4) windy 特征的信息增益率。

$$H(E)=H(\text{windy})=-(8/14)\log(8/14)-(6/14)\log(6/14)=0.682\,908\,104\,7$$

而上面已经计算了

$$g(\boldsymbol{D},E)=g(\boldsymbol{D},\text{windy})=0.057\,630\,406\,407$$

$$\text{gr}(\boldsymbol{D},E)=\text{gr}(\text{play},\text{windy})=g(\boldsymbol{D},E)/H(E)=0.084\,389\,694\,616\,8$$

对比上面 4 个特征的信息增益率，outlook 特征的信息增益率最大，所以 outlook 作为 root 节点。其他计算方法类似。

```
#!/usr/bin/python
#encoding:utf-8

# 对原始数据分类为训练数据和测试数据
import numpy as np
from sklearn import tree
from sklearn.model_selection import train_test_split
import pydotplus

def outlook_type(s):
  it = {b'sunny':1, b'overcast':2, b'rainy':3}
  return it[s]
def temperature(s):
  it = {b'hot':1, b'mild':2, b'cool':3}
  return it[s]
def humidity(s):
  it = {b'high':1, b'normal':0}
  return it[s]
def windy(s):
  it = {b'TRUE':1, b'FALSE':0}
  return it[s]

def play_type(s):
  it = {b'yes': 1, b'no': 0}
  return it[s]

play_feature_E = 'outlook', 'temperature', 'humidity', 'windy'
```

```
play_class = 'yes', 'no'

# 1.读入数据,并将原始数据中的数据转换为数字形式
data = np.loadtxt("play.txt", delimiter = " ", dtype = str, converters = {0:outlook_type,
1:temperature, 2:humidity, 3:windy,4:play_type})
x, y = np.split(data,(4,),axis = 1)

# 2.为了进行交叉验证,拆分训练数据与测试数据
# x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3,random_state = 2)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3)

# 3.使用信息熵作为划分标准,对决策树进行训练
clf = tree.DecisionTreeClassifier(criterion = 'entropy')
print(clf)
clf.fit(x_train, y_train)

# 4.把决策树结构写入文件
dot_data = tree.export_graphviz(clf, out_file = None, feature_names = play_feature_E, class_
names = play_class,filled = True, rounded = True, special_characters = True)
graph = pydotplus.graph_from_dot_data(dot_data)
graph.write_pdf('play1.pdf')

# 系数反映每个特征的影响力。系数越大表示该特征在分类中起到的作用越大
print(clf.feature_importances_)

# 5.使用训练数据预测,预测结果完全正确
answer = clf.predict(x_train)
y_train = y_train.reshape(-1)
print(answer)
print(y_train)
print(np.mean(answer == y_train))

# 6.对测试数据进行预测,准确率较低,说明过拟合
answer = clf.predict(x_test)
y_test = y_test.reshape(-1)
print(answer)
print(y_test)
print(np.mean(answer == y_test))
#输出
DecisionTreeClassifier(criterion = 'entropy')
[0.18045935 0.         0.59531747 0.22422318]
['1' '0' '0' '0' '1' '1' '0' '0' '1']
['1' '0' '0' '0' '1' '1' '0' '0' '1']
1.0
['0' '0' '1' '1' '0']
['1' '1' '1' '1' '1']
0.4
```

运行结果如图 3.18 所示。

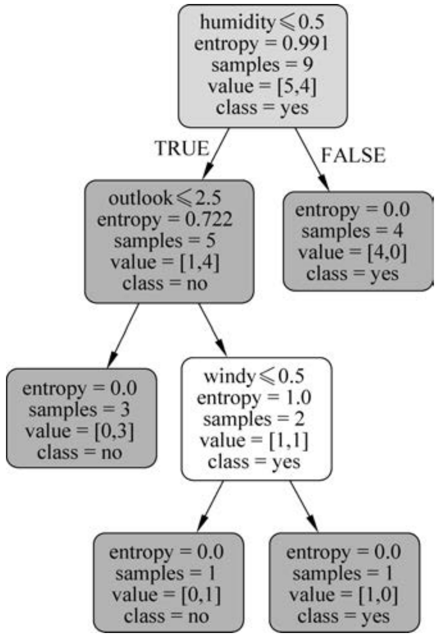


图 3.18 决策树结构图

3.3.4 实验

1. 实验目的

学会决策树的基本原理和基本的构建方法。了解分类问题以及训练集、测试集的构造以及决策树的基本定义，决策树的用法，构建决策树的方法流程。并拓展到在属性选择时用什么指标来度量，了解 ID3 信息熵以及 CART 使用基尼系数进行度量的两种重要方法。

2. 实验数据

选用经典的打球数据集，其数据示例见表 3.2。

3. 实验要求

分别使用 scikit-learn 的相关包、Python 语言编程来构建 ID3、C4.5 决策树，最后将训练出的决策树以图表形式显示。

3.4 支持向量机算法

3.4.1 原理简介

支持向量机(support vector machine，SVM)是一种二分类模型，它的基本模型是定义在特征空间上的间隔最大的线性分类器，间隔最大使它有别于感知机；SVM 还包括核技巧，这使它成为实质上的非线性分类器。SVM 的学习策略就是间隔最大化，可形式化为一个求解凸二次规划的问题，也等价于正则化的合页损失函数的最小化问题。SVM 的学习算法就是求解凸二次规划的最优化算法。

1. 线性 SVM 算法原理

SVM 的基本思想是求解能够正确划分训练数据集并且使几何间隔最大的分离超平面。如图 3.19 所示，其中，实心圆和空心圆代表两类样本；虚线为分类线，它们之间的距离叫作分类间隔(margin)；虚线上的点$(\boldsymbol{x}_i, y_i)$称为支持向量；$\boldsymbol{w}\cdot\boldsymbol{x}+b=0$ 即为分离超平面，对于线性可分的数据集来说，这样的超平面有无穷多个(即感知机)，但是几何间隔最大的分离超平面却是唯一的。

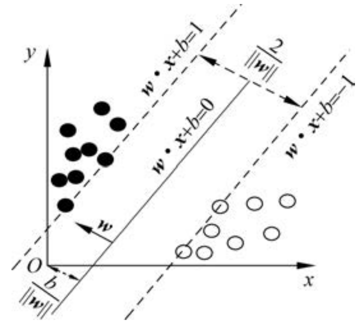


图 3.19　SVM 算法原理

假设给定一个在特征空间上线性可分的训练数据集为

$$\boldsymbol{T}=\{(\boldsymbol{x}_1,y_1),(\boldsymbol{x}_2,y_2),\cdots,(\boldsymbol{x}_N,y_N)\} \tag{3.49}$$

其中，$\boldsymbol{x}_i\in\mathbb{R}^n$，$y_i\in\{+1,-1\}$，$i=1,2,\cdots,N$；$\boldsymbol{x}_i$ 为第 i 个特征向量；y_i 为类标记，其为$+1$时代表正例，其为-1时代表负例。

对于给定的数据集 $\boldsymbol{T}$ 和超平面 $\boldsymbol{w}\cdot\boldsymbol{x}+b=0$，则超平面关于样本点$(\boldsymbol{x}_i, y_i)$的几何间隔为

$$\gamma_i=y_i\left(\frac{\boldsymbol{w}}{\|\boldsymbol{w}\|}\cdot\boldsymbol{x}_i+\frac{b}{\|\boldsymbol{w}\|}\right) \tag{3.50}$$

超平面关于所有样本点的几何间隔的最小值为

$$\gamma=\min_{i=1,2,\cdots,N}\gamma_i$$

实际上，这个最小的几何间隔就是支持向量到超平面的距离。

根据以上定义，SVM 模型求解最大分割超平面的问题可以表示为以下约束最优化问题：

$$\text{s.t.}\quad y_i\left(\frac{\boldsymbol{w}}{\|\boldsymbol{w}\|}\cdot\boldsymbol{x}_i+\frac{b}{\|\boldsymbol{w}\|}\right)\geqslant\gamma,\quad i=1,2,\cdots,N \tag{3.51}$$

将约束条件两边同时除以 γ，得到

$$y_i\left(\frac{\boldsymbol{w}}{\|\boldsymbol{w}\|\gamma}\cdot\boldsymbol{x}_i+\frac{b}{\|\boldsymbol{w}\|\gamma}\right)\geqslant 1 \tag{3.52}$$

因为$\|\boldsymbol{w}\|$、γ 都是标量，简化为

$$\boldsymbol{w}=\frac{\boldsymbol{w}}{\|\boldsymbol{w}\|\gamma}$$

$$b=\frac{b}{\|\boldsymbol{w}\|\gamma}$$

得到

$$y_i(\boldsymbol{w}\cdot\boldsymbol{x}_i+b)\geqslant 1,\quad i=1,2,\cdots,N \tag{3.53}$$

因为最大化 γ 等价于最大化$\frac{1}{\|\boldsymbol{w}\|}$，也等价于最小化$\frac{1}{2}\|\boldsymbol{w}\|^2$($\frac{1}{2}$是为了使后面的求导形式简洁，不影响结果)，故 SVM 模型求解最大分割超平面的问题又可以表示为以下约束最优化问题：

$$\min_{w,b} \frac{1}{2} \| \boldsymbol{w} \|^2 \tag{3.54}$$

约束条件为

$$\text{s.t.}\quad y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \geqslant 1, \quad i = 1,2,\cdots,N$$

这是一个含有不等式约束的凸二次规划问题，可以对其使用拉格朗日乘子法得到其对偶问题(dual problem)，此处不再进行公式推导。

至此都是基于训练集数据线性可分的假设下进行的，但是实际情况下几乎不存在完全线性可分的数据，为了解决这个问题，引入了“软间隔”的概念，即允许某些点不满足约束：

$$y_i(\boldsymbol{w} \cdot \boldsymbol{x}_j + b) \geqslant 1 \tag{3.55}$$

采用 hinge 损失函数，将原优化问题改写为

$$\min_{w,b,\varepsilon_i} \frac{1}{2} \| \boldsymbol{w} \|^2 + C \sum_{i=1}^{m} \varepsilon_i \tag{3.56}$$

约束条件为

$$\text{s.t.}\quad y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \geqslant 1 - \varepsilon_i, \varepsilon_i \geqslant 0, \quad i = 1,2,\cdots,N$$

其中，ε_i 为松弛变量，$\varepsilon_i = \max(0, 1 - y_i(w \cdot x_i + b))$，即一个 hinge 损失函数。每个样本都有一个对应的松弛变量，表征该样本不满足约束的程度。$C>0$ 称为惩罚参数，C 值越大，对分类的惩罚越大。与线性完全可分求解的思路一样，这里需先用拉格朗日乘子法得到拉格朗日函数，再求其对偶问题。

2. 非线性 SVM 算法原理

在非线性的情况下，对于参数 ε 无法精准地估计，会对结果造成较大的影响及误差，在这里可以引入松弛因子 ξ_i、ξ_i^*，则二次凸优化问题可以变形为

$$\min \frac{1}{2} \| \boldsymbol{w} \|^2 + C \sum_{i=1}^{N} (\xi_i + \xi_i^*) \tag{3.57}$$

而相应的约束条件也发生变化：

$$\begin{cases} y_i - \boldsymbol{w} \cdot \boldsymbol{x}_i - b \leqslant \varepsilon + \xi_i \\ \boldsymbol{w} \cdot \boldsymbol{x}_i + b - y_i \leqslant \varepsilon + \xi_i^* \end{cases} \tag{3.58}$$

根据实际情况可知，ξ_i、ξ_i^* 为正数，ε 则需满足不敏感损失函数，其表达式为

$$e(f(\boldsymbol{x}) - y) = \max(0, | f(\boldsymbol{x}) - y | - \varepsilon) \tag{3.59}$$

运用拉格朗日函数及对偶变量，则有

$$\begin{aligned} L = {} & \frac{1}{2} \| \boldsymbol{w} \|^2 + C \sum_{i=1}^{N} (\xi_i + \xi_i^*) - \sum_{i=1}^{N} \alpha_i (\xi_i + \varepsilon - y_i + \boldsymbol{w} \cdot \boldsymbol{x}_i + b) - \\ & \sum_{i=1}^{N} \alpha_i^* (\xi_i^* + \varepsilon - y_i - \boldsymbol{w} \cdot \boldsymbol{x}_i - b) - \sum_{i=1}^{N} (\eta_i \xi_i + \eta_i^* \xi_i^*) \end{aligned} \tag{3.60}$$

其中，η_i、η_i^*、α_i、α_i^*、C 均大于 0。

再通过 KKT(Karush-Kuhn-Tucker)条件的运算得出

$$\begin{aligned} W(\alpha_i, \alpha_i^*) = {} & -\frac{1}{2} \sum_{i,j=1}^{N} (\alpha_i - \alpha_i^*)(\alpha_j - \alpha_j^*) k(\boldsymbol{x}_i, \boldsymbol{x}_j) + \\ & \sum_{i=1}^{N} (\alpha_i - \alpha_i^*) y_i - \sum_{i=1}^{N} (\alpha_i + \alpha_i^*) \varepsilon \end{aligned} \tag{3.61}$$

并且有

$$\omega=\sum_{i=1}^{N}(\alpha_i-\alpha_i^*)\phi(\boldsymbol{x}_i)$$

对于输入空间中的非线性分类问题,可以通过非线性变换将其转换为不同维特征空间中的线性分类问题,在高维特征空间中去学习一个线性支持向量机。由于在线性 SVM 学习的对偶问题里,目标函数和分类决策函数都只涉及实例和实例之间的内积,不需要显式地指定非线性变换,所以在非线性 SVM 学习的对偶问题里,可用核函数替换当中的内积,其中,核函数表示通过一个非线性转换后的两个实例间的内积。具体地,$K(\boldsymbol{x}_i,\boldsymbol{x}_j)$是一个函数,意味着存在一个从输入空间到特征空间的映射 $\phi(\boldsymbol{x})$,对任意输入空间中的 $\boldsymbol{x}_i$、$\boldsymbol{x}_j$,有

$$K(\boldsymbol{x}_i,\boldsymbol{x}_j)=\phi(\boldsymbol{x}_i)\phi(\boldsymbol{x}_j) \tag{3.62}$$

在实际应用中,通常人们会从一些常用的核函数里选择(根据样本数据的不同,选择不同的参数,实际上就得到了不同的核函数),下面给出常用的核函数:

- 线性函数:$K(\boldsymbol{x}_i,\boldsymbol{x}_j)=\boldsymbol{x}_i\cdot\boldsymbol{x}_j$。
- 多项式核函数:$K(\boldsymbol{x}_i,\boldsymbol{x}_j)=(\boldsymbol{x}_i\cdot\boldsymbol{x}_j+1)^d$。
- 径向基核函数:$K(\boldsymbol{x}_i,\boldsymbol{x}_j)=\exp(-\|\boldsymbol{x}_i-\boldsymbol{x}_j\|^2/(2\sigma^2))$。
- 拉普拉斯核函数:$K(\boldsymbol{x}_i,\boldsymbol{x}_j)=\exp(-\|\boldsymbol{x}_i-\boldsymbol{x}_j\|/\sigma^2)$。
- Sigmoid 核函数:$(\beta>0,\theta>0)K(x_i,x_j)=\tanh(\beta\boldsymbol{x}_i^{\mathrm{T}}\cdot\boldsymbol{x}_j+\theta)$。

综上所述,用核函数替代线性 SVM 学习的对偶问题中的内积,求解得到非线性 SVM:

$$\begin{aligned}f(\boldsymbol{x})&=\sum_{i=1}^{N}(\alpha_i-\alpha_i^*)(\phi(\boldsymbol{x}_i)\cdot\phi(\boldsymbol{x}))+b\\&=\sum_{i=1}^{N}(\alpha_i-\alpha_i^*)K(\boldsymbol{x}_i,\boldsymbol{x})+b\end{aligned} \tag{3.63}$$

3.4.2 算法步骤

1) SVM 的选用

当训练数据线性可分时,通过硬间隔最大化,学习一个线性分类器,即线性可分支持向量机。

当训练数据近似线性可分时,通过软间隔最大化,学习一个线性分类器,即线性支持向量机。

当训练数据线性不可分时,通过使用核函数,将低维度的非线性问题转换为高维度下的线性问题,学习得到非线性支持向量机。

2) 线性 SVM 算法

(1) 输入:训练数据集 $\boldsymbol{T}=\{(\boldsymbol{x}_1,y_1),(\boldsymbol{x}_2,y_2),\cdots,(\boldsymbol{x}_N,y_N)\}$,其中,$\boldsymbol{x}_i\in\mathbb{R}^n$,$y_i\in\{+1,-1\}$,$i=1,2,\cdots,N$。

(2) 输出:分离超平面和分类决策函数。

(3) 算法步骤。

① 选择惩罚参数 $C>0$,构造并求解凸二次规划问题。

$$\min_{\alpha}\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j(\boldsymbol{x}_i\cdot\boldsymbol{x}_j)-\sum_{i=1}^{N}\alpha_i \tag{3.64}$$

$$\text{s.t.}\quad \sum_{i=1}^{N}\alpha_i y_i = 0$$

$$0 \leqslant \alpha_i \leqslant C,\quad i=1,2,\cdots,N$$

得到最优解$\boldsymbol{\alpha}^*=[\alpha_1^*,\alpha_2^*,\cdots,\alpha_N^*]^{\mathrm{T}}$。

② 计算。

$$\boldsymbol{w}^* = \sum_{i=1}^{N}\alpha_i^* y_i \boldsymbol{x}_i \tag{3.65}$$

选择$\boldsymbol{\alpha}^*$的一个分量 α_j^* 满足条件 $0\leqslant\alpha_j^*\leqslant C$,计算:

$$b^* = y_j - \sum_{i=1}^{N}\alpha_j^* y_i(\boldsymbol{x}_i \cdot \boldsymbol{x}_j) \tag{3.66}$$

③ 求分离超平面。

$$\boldsymbol{w}^* \cdot \boldsymbol{x} + b^* = 0 \tag{3.67}$$

分类决策函数:

$$f(\boldsymbol{x}) = \mathrm{sign}(\boldsymbol{w}^* \cdot \boldsymbol{x} + b^*) \tag{3.68}$$

3) 非线性 SVM 算法

(1) 输入:训练数据集 $\boldsymbol{T}=\{(\boldsymbol{x}_1,y_1),(\boldsymbol{x}_2,y_2),\cdots,(\boldsymbol{x}_N,y_N)\}$,其中,$\boldsymbol{x}_i\in\mathbb{R}^n$,$y_i\in\{+1,-1\}$,$i=1,2,\cdots,N$。

(2) 输出:分离超平面和分类决策函数。

(3) 算法步骤。

① 选择适当的核函数 $K(\boldsymbol{x},y)$和惩罚参数 $C>0$,构造并求解凸二次规划问题。

$$\min_{\alpha}\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j K(\boldsymbol{x}_i \cdot \boldsymbol{x}_j) - \sum_{i=1}^{N}\alpha_i \tag{3.69}$$

$$\text{s.t.}\quad \sum_{i=1}^{N}\alpha_i y_i = 0$$

$$0 \leqslant \alpha_i \leqslant C,\quad i=1,2,\cdots,N$$

得到最优解$\boldsymbol{\alpha}^*=[\alpha_1^*,\alpha_2^*,\cdots,\alpha_N^*]^{\mathrm{T}}$。

② 计算。

$$\boldsymbol{w}^* = \sum_{i=1}^{N}\alpha_i^* y_i \boldsymbol{x}_i \tag{3.70}$$

选择$\boldsymbol{\alpha}^*$的一个分量 α_j^* 满足条件 $0\leqslant\alpha_j^*\leqslant C$,计算:

$$b^* = y_j - \sum_{i=1}^{N}\alpha_j^* y_i K(\boldsymbol{x}_i \cdot \boldsymbol{x}_j) \tag{3.71}$$

③ 分类决策函数。

$$f(\boldsymbol{x}) = \mathrm{sign}\left(\sum_{i=1}^{N}\alpha_j^* y_i K(\boldsymbol{x} \cdot \boldsymbol{x}_i) + b^*\right) \tag{3.72}$$

4) 自编 SMO(序列最小优化)算法的步骤

(1) 启发式方法选择 α_1 和 α_2。

(2) 计算上界 H 和下界 L。

(3) 计算误差项 E_i。

(4) 更新 α_2。

(5) 更新 α_1。

(6) 更新 b。

3.4.3 实战

1. 数据集

本实战选用经典的数据集：鸢尾花数据集，该数据集的详细介绍见 2.1.4 节。

2. Sklearn 实现

1) 线性 SVM

导入相关模块：

```
# 导入模块
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import svm, datasets
from sklearn.preprocessing import StandardScaler
```

导入鸢尾花数据：

```
# 鸢尾花数据
iris = datasets.load_iris()
x = iris.data
y = iris.target
X = x[y<2,:2]  # 为便于绘图仅选择两个特征
y = y[y<2]     # 为提供线性可分数据集,只选取了前两类鸢尾花数据集
```

数据预处理。选择归一化对数据进行无量纲化处理：

```
# 数据预处理：数据归一化
scaler = StandardScaler()
scaler.fit(X)
X = scaler.transform(X)
```

模型引入及训练：

```
#引入模型：线性 SVM
model = svm.SVC(C=1e9, kernel='linear')
# 模型训练
model.fit(X, y)
```

绘制分类区域的网格：

```
# 测试样本(绘制分类区域)：以样本的特征向量生成多维的网格采样点矩阵
axis=[-3,3,-3,3]
xlist1 = np.linspace(axis[0],axis[1],int((axis[1]-axis[0])*100))
xlist2 = np.linspace(axis[2],axis[3],int((axis[3]-axis[2])*100))
XGrid1, XGrid2 = np.meshgrid(xlist1, xlist2)
```

计算正例分类线、负例分类线：

```
w = model.coef_[0]
b = model.intercept_[0]
plot_x = np.linspace(axis[0],axis[1],200)
up_y = -w[0]/w[1]*plot_x - b/w[1] + 1/w[1]
down_y = -w[0]/w[1]*plot_x - b/w[1] - 1/w[1]
up_index = (up_y>=axis[2]) & (up_y<=axis[3])
down_index = (down_y>=axis[2]) & (down_y<=axis[3])
```

结果可视化：

```
# 预测并绘制结果
Z = model.predict(np.vstack([XGrid1.ravel(), XGrid2.ravel()]).T)
Z = Z.reshape(XGrid1.shape)
# 设置标题、横坐标、纵坐标
plt.figure('SVM Linear', facecolor='lightgray')
plt.title('SVM Linear', fontsize=14)
plt.xlabel('x', fontsize=12)
plt.ylabel('y', fontsize=12)
plt.tick_params(labelsize=10)
# 绘制分类线
# y=1, 正例分类线
plt.plot(plot_x[up_index],up_y[up_index],c='black',linewidth=2,linestyle=':')
# y=-1, 负例分类线
plt.plot(plot_x[down_index],down_y[down_index],c='black',linewidth=2,linestyle=':')
# 绘制超平面
# 填充分类轮廓
plt.contourf(XGrid1, XGrid2, Z, cmap=ListedColormap(['#EF9A9A','#FFF59D','#90CAF9']))
# 绘制轮廓线、超平面
plt.contour(XGrid1, XGrid2, Z, colors=('k',))
# 绘制数据点
# 绘制类别 1
plt.scatter(X[y==0,0],X[y==0,1],s=60, marker='o', alpha=0.7, label='class 1', c='r')
# 绘制类别 2
plt.scatter(X[y==1,0],X[y==1,1],s=60, marker='o', alpha=0.7, label='class 2', c='b')
# 绘制支持向量
plt.scatter(model.support_vectors_[:,0],model.support_vectors_[:,1],s=100, c="none",
edgecolor='k', marker="o",label="support_vector")
plt.legend()
plt.show()
```

线性 SVM 的效果展示如图 3.20 所示。

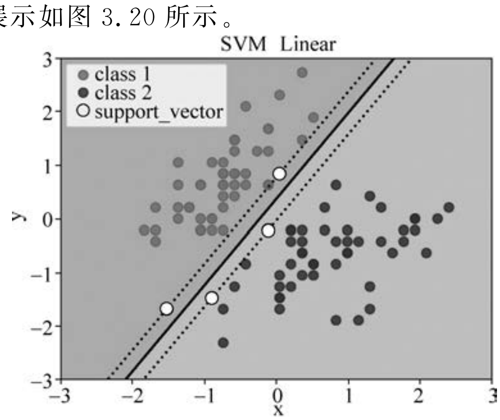


图 3.20 线性 SVM 的效果展示

2）非线性 SVM

导入相关模块：

```
# 导入模块
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn import svm, datasets
from sklearn.preprocessing import StandardScaler
```

导入鸢尾花数据集：

```
# 鸢尾花数据
iris = datasets.load_iris()
X = iris.data[:, :2]   # 为便于绘图仅选择两个特征
y = iris.target        # 对三类鸢尾花进行分类
```

数据预处理。选择归一化对数据进行无量纲化处理：

```
# 数据预处理：数据归一化
scaler = StandardScaler()
scaler.fit(X)
X = scaler.transform(X)
```

模型引入及训练：

```
# 非线性 SVM:
'''''
  常用核函数：kernel,多项式核函数:poly，径向基核函数:rbf，线性核函数:linear。
  核函数的超参数 gamma：0.5,
  SVC 的惩罚参数 C：1,
  SMO 迭代精度：1e-5，即停止训练的误差值大小
  内存 cache_size：1000MB，即核函数 cache 缓存大小
'''
model = svm.SVC(kernel='rbf', C=1, gamma=0.5, tol=1e-5, cache_size=1000).fit(X, y)
```

绘制分类区域的网格：

```
# 测试样本(绘制分类区域)
xlist1 = np.linspace(X[:, 0].min(), X[:, 0].max(), 200)
xlist2 = np.linspace(X[:, 1].min(), X[:, 1].max(), 200)
XGrid1, XGrid2 = np.meshgrid(xlist1, xlist2) # 以样本的特征向量生成多维的网格采样点矩阵
```

结果可视化：

```
# 预测并绘制结果
Z = model.predict(np.vstack([XGrid1.ravel(), XGrid2.ravel()]).T)
Z = Z.reshape(XGrid1.shape)
# 设置标题、横坐标、纵坐标
plt.figure('SVM RBF', facecolor='lightgray')
plt.title('SVM RBF', fontsize=14)
plt.xlabel('x', fontsize=12)
```

```
plt.ylabel('y', fontsize = 12)
plt.tick_params(labelsize = 10)
# 绘制分类区域
# 填充分类轮廓
plt.contourf(XGrid1, XGrid2, Z, cmap = ListedColormap(['#EF9A9A','#FFF59D','#90CAF9']))
plt.contour(XGrid1, XGrid2, Z, colors = ('k',)) # 绘制分类轮廓线
# 绘制样本点
plt.scatter(X[:, 0], X[:, 1], s = 60, marker = 'o', alpha = 0.7, label = 'Sample Points', c = y,
cmap = 'copper')
plt.legend()
plt.show()
```

非线性 SVM 的效果展示如图 3.21 所示。

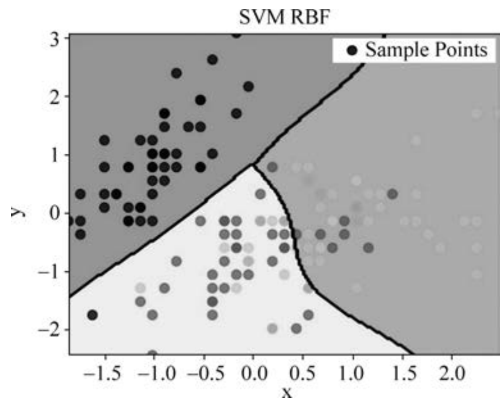


图 3.21　非线性 SVM 的效果展示

3. 自编代码实现

导入相应模块：

```
from numpy import *
import numpy as np
import random
import matplotlib.pyplot as plt
import pandas as pd
from sklearn import svm, datasets
from sklearn.preprocessing import StandardScaler
```

生成对应 alpha 的下标：

```
# 从 0～m 中产生一个不为 i 的整数
# i 是第一个 alpha 的下标,m 是所有 alpha 的数目
def selectJrand(i,m):
  j = i
  while(j == i):
     j = int(random.uniform(0,m))
  return j
```

过滤条件：

```
# 使得 aj 在边界值[L,H]以内
def clipAlpha(aj,H,L):
  if aj > H:
    aj = H
  if L > aj:
    aj = L
  return aj
```

简化 SMO 实现：

```
#SMO(序列最小优化)
#dataMatIn = 数据集,classLabels = 类别标签,常数 C,toler = 容错率 和 maxIter = 退出前的最大
#循环次数
def smoSimple(dataMatIn,classLabels,C,toler,maxIter):
  dataMat = mat(dataMatIn) #(100,2)
  labelMat = mat(classLabels).transpose() #(100,1)
  b = 0
  m,n = shape(dataMat) # m 为行数 ,m = 100, n 为列数, n = 2
  alphas = mat(zeros((m,1)))
  # iter 存储在没有任何 alpha 改变的情况下遍历数据集的次数,当该变量达到输入值 maxIter
# 时,函数结束运行并退出
  iter = 0
  # alphaPairsChanged 表示用来更新的次数
  # 当遍历连续无更新 maxIter 轮,则认为收敛,迭代结束
  while iter < maxIter:
    alphaPairsChanged = 0
    for i in range(m):
        # KKT 条件计算出
        # fXi 是预测的类别
        fXi = float(multiply(alphas,labelMat).T * (dataMat * dataMat[i,:].T)) + b
        # 误差,即输入 xi 的预测值和真实输出值 yi 之差
        Ei = fXi - float(labelMat[i])
        # toler:容忍错误的程度
        # labelMat[i] * Ei < - toler 则需要 alphas[i]增大,但是不能≥C
        # labelMat[i] * Ei > toler 则需要 alphas[i]减小,但是不能≤0
        if ((labelMat[i] * Ei < - toler) and (alphas[i] < C)) or ((labelMat[i] * Ei > toler)
and (alphas[i] > 0)):
            # 从 0～m 产生一个不为 i 的整数
            j = selectJrand(i,m)
            fXj = float(multiply(alphas,labelMat).T * (dataMat * dataMat[j,:].T) + b)
            Ej = fXj - float(labelMat[j])
            alphaIold = alphas[i].copy()
            alphaJold = alphas[j].copy()

            if labelMat[i] != labelMat[j]:
              L = max(0,alphas[j] - alphas[i])
              H = min(C,C + alphas[j] - alphas[i])
            else:
              L = max(0,alphas[j] + alphas[i] - C)
              H = min(C,alphas[j] + alphas[i])

            if L == H:
              continue
```

```
            eta = -2.0 * dataMat[i,:] * dataMat[j,:].T + dataMat[i,:] * \
                dataMat[i,:].T + dataMat[j,:] * dataMat[j,:].T
            # eta 是用于做分母的,不能为 0
            if eta <= 0:
                continue
            # 更新 alphas[j] 数值
            alphas[j] += labelMat[j] * (Ei - Ej)/eta
            alphas[j] = clipAlpha(alphas[j],H,L)
            # 更新 alphas[i] 数值
            alphas[i] += labelMat[j] * labelMat[i] * (alphaJold - alphas[j])
            b1 = b - Ei - labelMat[i] * (alphas[i] - alphaIold) * dataMat[i,:] * dataMat[i,:].T\
                 - labelMat[j] * (alphas[j] - alphaJold) * dataMat[i,:] * dataMat[j,:].T
             b2 = b - Ej - labelMat[i] * (alphas[i] - alphaIold) * dataMat[i, :] *
                   dataMat[j, :].T \ - labelMat[j] * (alphas[j] - alphaJold) * dataMat[j, :] *
                   dataMat[j, :].T

            if (0 < alphas[i]) and (C > alphas[i]):
                b = b1
            elif (0 < alphas[j]) and (C > alphas[j]):
                b = b2
            else:
                b = (b1 + b2)/2.0

            alphaPairsChanged += 1

        if alphaPairsChanged == 0:
            iter += 1
        else:
            iter = 0

    return b,alphas
```

引入鸢尾花数据集：

```
# 鸢尾花数据
iris = datasets.load_iris()
x = iris.data
y = iris.target
X = x[y<2,:2]   # 为便于绘图,仅选择两个特征
y = y[y<2]      # 为提供线性可分数据集,只选取了前两类鸢尾花数据集
```

重置标签。将两种类别分别记为−1 和 1：

```
# 重置标签
for i in range(int(len(y))):
  if y[i] == 0:
     y[i] = -1
```

归一化：

```
# 数据预处理: 数据归一化
scaler = StandardScaler()
```

```
scaler.fit(X)
X = scaler.transform(X)
```

转变类型：

```
# 转为 list 类型
dataMat, labelMat = X.tolist(), y.tolist()
```

调用 SMO 算法：

```
# 调用 SMO 算法:计算 b 和 alphas
b, alphas = smoSimple(dataMat, labelMat,1,0.001,40)
```

计算 w：

```
# 计算最佳的 w
w_best = np.dot(np.multiply(alphas, labelMat).T, dataMat)
```

计算超平面、正例分类线、负例分类线：

```
# 按照数据集所在范围,选取 200 个范围内的随机点,分别计算超平面、正例分类线、负例分类线
plot_x = np.linspace(-3,3,200)   # 横坐标
lin_y = (float(b) + w_best[0,0] * plot_x) / w_best[0,1] # 超平面上的随机点的纵坐标
# 正例分类线上随机点的纵坐标
up_y = w_best[0,0]/w_best[0,1] * plot_x + b/ w_best[0,1] + 1/ w_best[0,1]
# 负例分类线上的随机点的纵坐标
down_y = w_best[0,0]/w_best[0,1] * plot_x + b/w_best[0,1] - 1/w_best[0,1]
# 过滤条件:防止 lin_y、up_y 和 down_y 的结果超过了 plot_x 中 y 坐标的范围
line_index = (lin_y>=-3) & (lin_y<=3)
up_index = (up_y>=-3) & (up_y<=3)
down_index = (down_y>=-3) & (down_y<=3)
# 将 matrix 转换为 array,并展平
up_y = up_y.A
up_y = up_y.flatten()
down_y = down_y.A
down_y = down_y.flatten()
# 同上
up_index = up_index.A
up_index = up_index.flatten()
down_index = down_index.A
down_index = down_index.flatten()
```

结果可视化：

```
# 设置标题、横坐标、纵坐标
plt.figure('SMO', facecolor='lightgray')
plt.title('SMO', fontsize=14)
plt.xlabel('x', fontsize=12)
plt.ylabel('y', fontsize=12)
plt.tick_params(labelsize=10)
# 绘制分类线
# y=1, 正例分类线
```

```
plt.plot(plot_x[up_index],up_y[up_index],c = 'black',linewidth = 2,linestyle = ':')
# y = -1, 负例分类线
plt.plot(plot_x[down_index],down_y[down_index],c = 'black',linewidth = 2,linestyle = ':')
plt.plot(plot_x[line_index],lin_y[line_index],c = 'black',linewidth = 2,linestyle = '-')
# 画出支持向量
plt.scatter(X[alphas.A.flatten()> 0, 0], X[alphas.A.flatten()> 0, 1], s = 100, c = 'none',
edgecolor = 'k', label = "support_vector")
plt.scatter(X[y == -1,0],X[y == -1,1],s = 60, marker = 'o', label = 'class 1', c = 'r') # 绘制类别 1
plt.scatter(X[y == 1,0],X[y == 1,1],s = 60, marker = 'o', label = 'class 2', c = 'b')  # 绘制类别 2
plt.legend()
plt.show()
```

调用 SMO 的效果展示如图 3.22 所示。

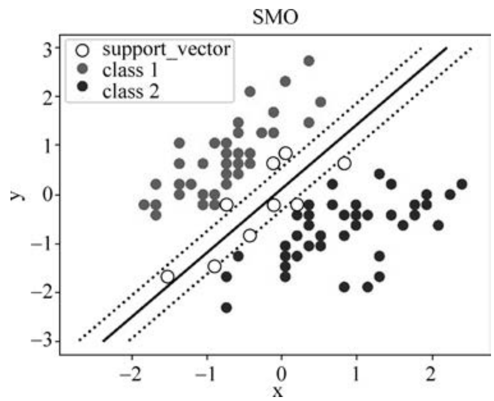


图 3.22 调用 SMO 的效果展示

3.4.4 实验

1. 实验目的

(1) 掌握 SVM 引入核函数的动机和核函数思想,会选用合适的核函数,对数据集进行分类。

(2) 理解软间隔和硬间隔,尤其是 KKT 条件的不同,并针对正则化系数 C 会进行参数调整和模型选择。

(3) 熟练使用 Python 及其 NumPy、Sklearn、Matplotlib 等第三方库。

2. 实验数据

定义五个函数,分别为 linear()、nolinear()、gauss_linear()、gauss_nolinear()、circle(),并按要求生成不同分布的数据:

(1) 生成两组线性均匀分布的数据(完全线性可分)。

(2) 生成两组线性均匀分布的数据(线性不可分)。

(3) 生成两组高斯分布的数据(完全线性可分)。

(4) 生成两组高斯分布的数据(线性不可分)。

(5) 生成环状数据。

3. 实验要求

使用线性 SVM 对实验数据(1)中生成的数据进行分类,并画出分类界面。

使用线性 SVM 对实验数据(2)中生成的数据进行分类,并画出分类界面。

分别使用 Linear 核、rbf 核、degree=2 的 poly 核和 degree=3 的 poly 核的 SVM 对实验数据(3)中生成的数据进行分类,并画出分类界面。

分别使用 Linear 核、rbf 核、degree=2 的 poly 核和 degree=3 的 poly 核的 SVM 对实验数据(4)中生成的数据进行分类,并画出分类界面。

分别使用 rbf 核、degree=2,3,4 的 poly 核的 SVM 对实验数据(5)中生成的数据进行分类,并画出分类界面。

3.5 EM 算法

3.5.1 原理简介

EM 算法是一种求含隐变量概率模型最优参数的迭代算法。该算法通过 E(expectation)步求期望,通过 M(maximization)步求期望最大化,不断迭代,使得似然函数的值不断变大。然而它不是机器学习模型,不学习输入(自变量)与输出(因变量)之间的模型关系。但是,若模型关系是一个含参数的概率分布,并且含有隐变量,则可以用 EM 算法估计出参数的取值。这就是 EM 算法有广泛应用的原因。在具体应用中,首先应根据具体问题给出具体的概率模型,然后,利用 EM 算法求解模型。常见的含隐变量的机器学习概率模型有高斯混合模型(Gaussian mixture model,GMM)和隐马尔可夫模型(hidden Markov model,HMM)。本节先介绍极大似然估计求解概率模型的参数,由隐变量导致完全数据与不完全数据,再阐述 EM 算法的原理,最后给出基于高斯混合模型的 EM 算法的应用。

1. 极大似然估计

概率模型是指:假定产生一组观测数据$\{x_i\}_{i=1}^n$的概率分布为$p(\boldsymbol{x}|\boldsymbol{\theta})$的形式已知,$\boldsymbol{x}$是随机变量,$\boldsymbol{\theta}$是未知参数。例如,图 3.23 中有 300 个样本点,这些样本点均来自于二元高斯模型:

$$p(\boldsymbol{x} \mid \boldsymbol{\theta})=\frac{1}{\sqrt{(2\pi)^m}}|\boldsymbol{\Sigma}|^{-1/2}\exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right) \tag{3.73}$$

其中,$\boldsymbol{\theta}=(\boldsymbol{\mu},\boldsymbol{\Sigma})$是参数,此时取$m=2$,且$\boldsymbol{\mu}$是二维均值向量,$\boldsymbol{\Sigma}$是二阶协方差矩阵。如何根据已知样本数据求取参数?

基于样本独立同分布的原理,由已知分布得到,这一组样本的$\{x_i\}_{i=1}^n$产生的似然函数为

$$l(\boldsymbol{\theta})=\prod_{i=1}^{n}p(x_i \mid \boldsymbol{\theta}) \tag{3.74}$$

最优参数是使似然函数取得最大值的一组参数,即

$$\hat{\theta}=\underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, l(\boldsymbol{\theta})=\underset{\boldsymbol{\theta}}{\operatorname{argmax}}\prod_{i=1}^{n}p(x_i \mid \boldsymbol{\theta}) \tag{3.75}$$

式(3.75)的解就是极大似然估计的解。顾名思义,估计是指求未知参数,方式是对似然函数求极大值点。通俗地解释为,求得的参数是使这一组样本值出现概率最大的一组参数值。但是由于概率小于1,再连乘后的值会越来越小并接近0,导致数值下溢。同时求解优化问题的梯度法在这里仍然适用,但是连乘的形似不利于求导。因此将似然函数取对数,得到对数似然函数,记为 $L(\boldsymbol{\theta})$,即

$$L(\boldsymbol{\theta})=\ln(l(\boldsymbol{\theta}))=\sum_{i=1}^{n}\ln(p(x_i \mid \boldsymbol{\theta})) \tag{3.76}$$

对数化以后会使得函数的值域从原本的(0,1)区间,变成了(0,+∞),有效地防止了数值下溢。同时,由于对数函数是单调递增的,导致对数似然函数的最优值点与原似然函数的最优值点是相同的,即下列优化问题

$$\hat{\theta}=\underset{\boldsymbol{\theta}}{\operatorname{argmax}} L(\boldsymbol{\theta})=\underset{\boldsymbol{\theta}}{\operatorname{argmax}}\sum_{i=1}^{n}\ln(p(x_i \mid \boldsymbol{\theta})) \tag{3.77}$$

与式(3.75)的解是一致的。例如,若 $p(\boldsymbol{x}|\boldsymbol{\theta})$ 为二元高斯分布,由式(3.77)求出的最优解为样本均值与样本协方差矩阵(具体的推导可参阅统计学教材)。

2. EM 算法的引入

设概率模型中含有隐变量,例如,图3.24中1000个样本点分别来自3个二元高斯分布。此时每个样本点属于某个高斯分布就是隐变量 z。

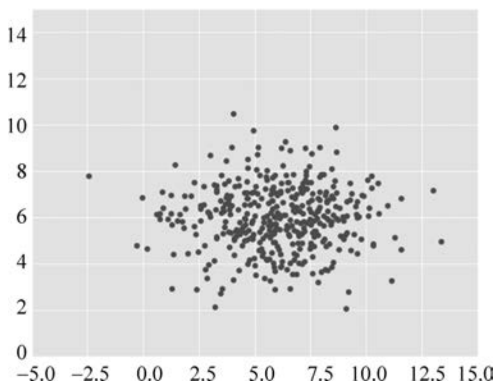


图3.23 一个二元高斯分布产生的样本点

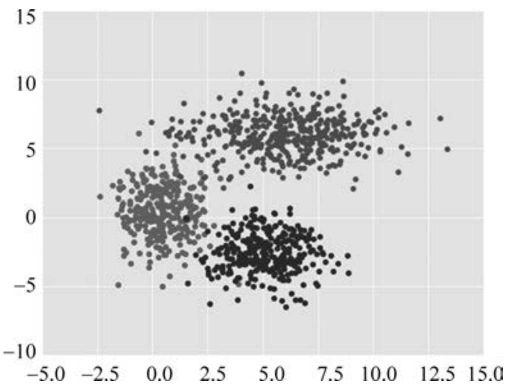


图3.24 3个二元高斯分布产生的样本点

此时,可观察到的数据仅仅为样本点的横纵坐标,则数据 $\{x_i\}_{i=1}^{n}$ 被称为不完全数据,若不仅知道样本点的横、纵坐标,还知道每个样本点来自哪个高斯分布 $\{x_i, z_i\}_{i=1}^{n}$,则称这样的数据为完全数据。在完全数据下估计参数是相对容易的,然而现实中若只能得到不完全数据,则需要在式(3.77)中引入隐变量 z,令隐变量 z 的概率分布为

$$\begin{cases} p(z=j)=\pi_j \geqslant 0 \\ \sum_{j=1}^{k}\pi_j=1 \end{cases} \tag{3.78}$$

在处理实际问题中,除了对概率模型的参数做出估计,往往也关心每个样本由 $z=j$ 这个类产生的概率值大小,这样就可以根据概率最大值来认定这个样本来自第几个类。例如,图3.24中每个样本点是由哪类高斯分布生成的,就可以根据概率最大值所属的类进行判定。因此,令 $p(z=j|x_i,\boldsymbol{\theta})$ 表示第 i 个样本来自 $z=j$ 这个类的概率,简记为 $q_{ij}(\boldsymbol{\theta})$,由条

件概率公式与全概率公式,有

$$q_{ij}(\boldsymbol{\theta})=p(z=j \mid x_i,\boldsymbol{\theta})=\frac{p(z=j \mid x_i,\boldsymbol{\theta})}{p(x_i,\boldsymbol{\theta})}=\frac{p(z=j \mid x_i,\boldsymbol{\theta})}{\sum_{j=1}^{k}p(x_i,z=j \mid \boldsymbol{\theta})} \tag{3.79}$$

式(3.79)中第一个等式是条件概率公式,第二个等式是对分母使用全概率公式。

将 $q_{ij}(\boldsymbol{\theta})$对第二个指标求和,有

$$\sum_{i=1}^{k}q_{ij}(\boldsymbol{\theta})=\frac{\sum_{j=1}^{k}p(z=j \mid x_i,\boldsymbol{\theta})}{\sum_{j=1}^{k}p(x_i,z=j \mid \boldsymbol{\theta})}=1$$

表明对固定的 i、$\boldsymbol{\theta}$,$\{q_{ij}(\boldsymbol{\theta})\}_{j=1}^{k}$ 构成随机事件的一个概率分布。

考虑任意固定值 Θ,分布$\{q_{ij}(\Theta)\}_{j=1}^{k}$ 是与参数$\boldsymbol{\theta}$ 无关的概率分布,式(3.76)中的对数似然函数可写为

$$\begin{aligned}L(\boldsymbol{\theta})&=\sum_{i=1}^{n}\ln(p(x_i \mid \boldsymbol{\theta}))=\sum_{i=1}^{n}\ln\Big(\sum_{j=1}^{k}p(x_i,z=j \mid \boldsymbol{\theta}\Big)\\&=\sum_{i=1}^{n}\ln\left(\sum_{j=1}^{k}q_{ij}(\Theta)\frac{p(z=j,x_i,\boldsymbol{\theta})}{q_{ij}(\Theta)}\right)\end{aligned} \tag{3.80}$$

含有隐变量的极大似然估计的优化问题,为求式(3.80)的最值,但其中的 $L(\boldsymbol{\theta})$含有求和的对数,这在求导后会导致出现分式,且分母是求和式,不利于参数求解。这里利用 Jensen 不等式,将求和的对数优化转换为对数的和的形式,从而降低优化求解的难度。

Jensen 不等式是指当 X 是随机变量、$f(X)$为凸函数时,下面的关系成立:

$$E[f(X)]\geqslant f(E[X])$$

即对凸函数求期望值,大于或等于函数在期望处的取值。特别地,若 $f(X)$是对数函数,为凹函数,则 Jensen 不等式反号,即

$$E[\ln(X)]\leqslant \ln(E[X]) \tag{3.81}$$

在式(3.80)中,由于 $\sum_{j=1}^{k}q_{ij}(\Theta)=1$,因此 $\sum_{j=1}^{k}q_{ij}(\Theta)X=E[X]$。取 $X=\frac{p(z=j,x_i,\boldsymbol{\theta})}{q_{ij}(\Theta)}$,有$\sum_{j=1}^{k}q_{ij}(\Theta)\frac{p(z=j,x_i,\boldsymbol{\theta})}{q_{ij}(\Theta)}=E[X]$。代入式(3.81),有

$$\ln\left(\sum_{j=1}^{k}q_{ij}(\Theta)\frac{p(z=j,x_i,\boldsymbol{\theta})}{q_{ij}(\Theta)}\right)\geqslant\sum_{j=1}^{k}q_{ij}(\Theta)\ln\left(\frac{p(z=j,x_i,\boldsymbol{\theta})}{q_{ij}(\Theta)}\right) \tag{3.82}$$

将式(3.82)代入式(3.80),得到

$$\begin{aligned}L(\boldsymbol{\theta})&=\sum_{i=1}^{n}\ln\left(\sum_{j=1}^{k}q_{ij}(\Theta)\frac{p(z=j,x_i,\boldsymbol{\theta})}{q_{ij}(\Theta)}\right)\\&\geqslant\sum_{i=1}^{n}\sum_{j=1}^{k}q_{ij}(\Theta)\ln\left(\frac{p(z=j,x_i,\boldsymbol{\theta})}{q_{ij}(\Theta)}\right)\end{aligned} \tag{3.83}$$

式(3.83)右侧的表达式记为 $Q(\boldsymbol{\theta},\Theta)$,称为期望函数,即

$$Q(\boldsymbol{\theta},\Theta)=\sum_{i=1}^{n}\sum_{j=1}^{k}q_{ij}(\Theta)\ln\left(\frac{p(z=j,x_i,\boldsymbol{\theta})}{q_{ij}(\Theta)}\right) \tag{3.84}$$

由以上推导可知，对任意的θ、Θ，均有

$$L(\theta) \geqslant Q(\theta, \Theta) \tag{3.85}$$

将期望函数$Q(\theta,\Theta)$在$\theta=\Theta$处取值，利用式(3.79)，有

$$\begin{aligned}
Q(\Theta,\Theta) &= \sum_{i=1}^{n}\sum_{j=1}^{k} q_{ij}(\Theta)\ln\left(\frac{p(z=j,x_i,\Theta)}{q_{ij}(\Theta)}\right) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{k} q_{ij}(\Theta)\ln\left(\frac{p(z=j,x_i,\Theta)}{p(z=j \mid x_i,\Theta)}\right) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{k} q_{ij}(\Theta)\ln\left(\frac{p(z=j,x_i,\Theta)}{\dfrac{p(z=j,x_i,\Theta)}{p(x_i,\Theta)}}\right) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{k} q_{ij}(\Theta)\ln(p(x_i,\Theta)) \\
&= \sum_{i=1}^{n}\ln(p(x_i,\Theta))\sum_{j=1}^{k} q_{ij}(\Theta) \\
&= \sum_{i=1}^{n}\ln(p(x_i,\Theta)) = L(\Theta)
\end{aligned}$$

即$Q(\Theta,\Theta)=L(\Theta)$。

现在设置一个取值$\Theta=\theta^{(t)}$，对期望函数$Q(\theta,\theta^{(t)})$关于θ求极大值点，令其为$\theta^{(t+1)}$，即

$$\theta^{(t+1)} = \underset{\theta}{\operatorname{argmax}} Q(\theta, \theta^{(t)}) \tag{3.86}$$

由式(3.85)有

$$L(\theta^{(t+1)}) \geqslant Q(\theta^{(t+1)}, \theta^{(t)}) \geqslant Q(\theta^{(t)}, \theta^{(t)}) = L(\theta^{(t)}) \tag{3.87}$$

将式(3.87)形成的迭代序列$\{\theta^{(t)}\}$代入对数似然函数$L(\theta)$，得到单调递增的数列$\{L(\theta^{(t)})\}$。由对数函数的单调性，可知数列$\{l(\theta^{(t)})\}$也是单调递增的。同时由似然函数$l(\theta)$的定义（见式(3.74)），有$l(\theta)\leqslant 1$，即数列$\{l(\theta^{(t)})\}$有上界。而单调有界数列必有极限，即数列$\{l(\theta^{(t)})\}$收敛。因此数列$\{L(\theta^{(t)})\}$也收敛。事实上，式(3.86)就是EM算法的M步骤：求期望函数的极大值点。如此不断地迭代，算法必然收敛。但是如图3.25所示，EM算法可能导致数列$\{L(\theta^{(t)})\}$收敛于期望函数的局部极大值，而不能保证$L(\theta)$全局最大。

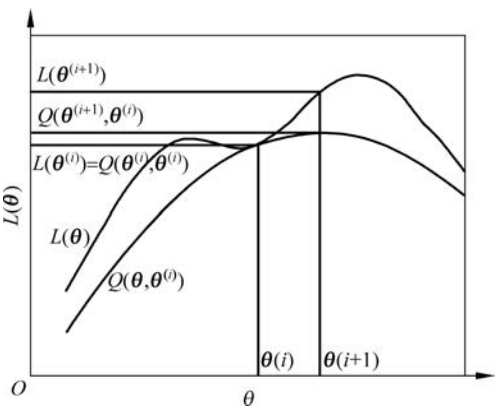


图3.25 EM算法原理示意图

EM算法的使用方法非常简单，通过E步和M步的交替计算即可实现。在M步中，对期望函数$Q(\theta,\theta^{(t)})$取极值点的计算使迭代不断朝着对数似然函数$L(\theta)$增大的方向迭代，这避免了直接对$L(\theta)$求导的复杂计算，使得计算变得可行，同时保证算法的收敛性。但是带来的后果是，迭代收敛的值不能保证是全局最优值。同时，EM算法在输入时，需要指定联合分布$p(z,\boldsymbol{x}|\theta)$的形式与条件分布$q_{ij}(\theta)$的初值。联合分布形式的确定需要视具体

问题而定，初始值的选择也会影响最后的结果。

3. 基于 EM 算法的混合高斯模型

混合高斯模型的应用相当广泛，理论上任何连续分布的概率模型都可以用混合高斯模型来拟合。在混合高斯模型中，产生样本 $\boldsymbol{x}_i$ 的方式是，先从 k 个高斯分布中以概率 $p(z=j)$ 选中一个高斯分布，再以概率$\mathcal{N}(\boldsymbol{x}_i,\boldsymbol{\mu}_j,\boldsymbol{\Sigma}_j)$生成该样本，具体公式如下：

$$p(\boldsymbol{x}_i)=\sum_{j=1}^{k}\pi_j\ \mathcal{N}(\boldsymbol{x}_i,\boldsymbol{\mu}_j,\boldsymbol{\Sigma}_j) \tag{3.88}$$

其中，$\sum_{j=1}^{k}p(z=j)=\sum_{j=1}^{k}\pi_j=1$，$p(\boldsymbol{x}_i\mid z=j,\boldsymbol{\theta})=\mathcal{N}(\boldsymbol{x}_i,\boldsymbol{\mu}_j,\boldsymbol{\Sigma}_j)$ 是多元高斯分布，具体形式如式(3.73)所示。高斯分布的选择成为隐变量，因此在已知样本$\{\boldsymbol{x}_i\}_{i=1}^{n}$ 的情况下，待估计的参数$\boldsymbol{\theta}=(\pi_j,\boldsymbol{\mu}_j,\boldsymbol{\Sigma}_j\mid j=1,\cdots,k)$可由 EM 算法给出。下面给出具体的过程。

E 步：

计算期望函数 $Q(\boldsymbol{\theta},\boldsymbol{\theta}^{(t)})$，将式(3.88)代入式(3.84)，得到

$$\begin{aligned}Q(\boldsymbol{\theta},\boldsymbol{\theta}^{(t)})&=\sum_{i=1}^{n}\sum_{j=1}^{k}q_{ij}(\boldsymbol{\theta}^{(t)})\ln\left(\frac{\pi_j\ \mathcal{N}(\boldsymbol{x}_i,\boldsymbol{\mu}_j,\boldsymbol{\Sigma}_j)}{q_{ij}(\boldsymbol{\theta}^{(t)})}\right)\\&=\sum_{i=1}^{n}\sum_{j=1}^{k}q_{ij}(\boldsymbol{\theta}^{(t)})\left[\ln(\pi_j)+\ln(\mathcal{N}(\boldsymbol{x}_i,\boldsymbol{\mu}_j,\boldsymbol{\Sigma}_j))-\ln(q_{ij}(\boldsymbol{\theta}^{(t)}))\right]\end{aligned} \tag{3.89}$$

其中，$q_{ij}(\boldsymbol{\theta}^{(t)})$是与$\boldsymbol{\theta}$ 无关的量，将式(3.88)代入式(3.79)，得到 $q_{ij}(\boldsymbol{\theta}^{(t)})$的形式为

$$q_{ij}(\boldsymbol{\theta}^{(t)})=\frac{\pi_j\ \mathcal{N}(\boldsymbol{x}_i,\boldsymbol{\mu}_j^{(t)},\boldsymbol{\Sigma}_j^{(t)})}{\sum_{j=1}^{k}\pi_j\ \mathcal{N}(\boldsymbol{x}_i,\boldsymbol{\mu}_j^{(t)},\boldsymbol{\Sigma}_j^{(t)})} \tag{3.90}$$

M 步：

求期望函数 $Q(\boldsymbol{\theta},\boldsymbol{\theta}^{(t)})$的极值点，由式(3.89)可以直接对向量$\boldsymbol{\mu}_j$和矩阵$\boldsymbol{\Sigma}_j$分别求导，并令导数为 0，求解得到

$$\boldsymbol{\mu}_j=\frac{\sum_{i=1}^{n}q_{ij}(\boldsymbol{\theta}^{(t)})\boldsymbol{x}_i}{\sum_{i=1}^{n}q_{ij}(\boldsymbol{\theta}^{(t)})} \tag{3.91}$$

$$\boldsymbol{\Sigma}_j=\frac{\sum_{i=1}^{n}q_{ij}(\boldsymbol{\theta}^{(t)})(\boldsymbol{x}_i-\boldsymbol{\mu}_j)^{\mathrm{T}}(\boldsymbol{x}_i-\boldsymbol{\mu}_j)}{\sum_{i=1}^{n}q_{ij}(\boldsymbol{\theta}^{(t)})} \tag{3.92}$$

由于参数 π_j 带有约束条件 $\sum_{j=1}^{k}\pi_j=1$，因此添加 Lagrange 乘子 λ，构造 Lagrange 函数$\mathcal{L}$如下：

$$\mathcal{L}=Q(\boldsymbol{\theta},\boldsymbol{\theta}^{(t)})-\lambda\left(\sum_{j=1}^{k}\pi_j-1\right) \tag{3.93}$$

将式(3.89)代入式(3.93)，计算$\frac{\partial\mathcal{L}}{\partial\pi_j}$并令其为 0，有

$$\frac{\partial \mathcal{L}}{\partial \pi_j}=\frac{\sum_{i=1}^{n} q_{ij}(\boldsymbol{\theta}^{(t)})}{\pi_j}-\lambda=0$$

解得

$$\pi_j=\frac{\sum_{i=1}^{n} q_{ij}(\boldsymbol{\theta}^{(t)})}{\lambda} \tag{3.94}$$

又由于 $\sum_{j=1}^{k}\pi_j=1$，对式(3.94)两边关于 j 求和，并利用 $\sum_{j=1}^{k} q_{ij}(\boldsymbol{\theta})=1$，有

$$1=\frac{\sum_{j=1}^{k}\sum_{i=1}^{n} q_{ij}(\boldsymbol{\theta}^{(t)})}{\lambda}=\frac{\sum_{i=1}^{n}\left(\sum_{j=1}^{k} q_{ij}(\boldsymbol{\theta}^{(t)})\right)}{\lambda}=\frac{n}{\lambda}$$

得到 $\lambda=n$，代入式(3.94)得到

$$\pi_j=\frac{\sum_{i=1}^{n} q_{ij}(\boldsymbol{\theta}^{(t)})}{n}$$

由此得到参数 $\boldsymbol{\theta}=(\pi_j,\boldsymbol{\mu}_j,\boldsymbol{\Sigma}_j \mid j=1,\cdots,k)$ 的全部更新公式。

3.5.2 算法步骤

EM 算法流程如下。

(1) 输入：观测数据 $\{\boldsymbol{x}_i\}_{i=1}^{n}$，隐变量 z，联合分布 $p(z=j,\boldsymbol{x}_i,\boldsymbol{\theta})$ 与条件分布 $q_{ij}(\boldsymbol{\theta})=p(z=j\mid \boldsymbol{x}_i,\Theta)$，正常数 ε。

(2) 输出：模型参数 $\boldsymbol{\theta}$。

① 设置初始值 $\boldsymbol{\theta}^{(0)}$。

② E 步：将第 t 次迭代参数 $\boldsymbol{\theta}^{(t)}$ 的值代入期望函数 $Q(\boldsymbol{\theta},\boldsymbol{\theta}^{(t)})$ 的形式，即

$$Q(\boldsymbol{\theta},\boldsymbol{\theta}^{(t)})=\sum_{i=1}^{n}\sum_{j=1}^{k} q_{ij}(\boldsymbol{\theta}^{(t)})\ln\left(\frac{p(z=j,\boldsymbol{x}_i,\boldsymbol{\theta})}{q_{ij}(\boldsymbol{\theta}^{(t)})}\right)$$

③ M 步：求期望函数 $Q(\boldsymbol{\theta},\boldsymbol{\theta}^{(t)})$ 的最值点，以此确定第 $t+1$ 次迭代参数 $\boldsymbol{\theta}^{(t+1)}$ 的值

$$\boldsymbol{\theta}^{(t+1)}=\underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, Q(\boldsymbol{\theta},\boldsymbol{\theta}^{(t)})$$

④ 若 $\|\boldsymbol{\theta}^{(t+1)}-\boldsymbol{\theta}^{(t)}\|<\varepsilon$ 或 $\|Q(\boldsymbol{\theta}^{(t+1)})-Q(\boldsymbol{\theta}^{(t)})\|<\varepsilon$，则停止迭代；否则，重复第②步和第③步。

基于 EM 算法的高斯混合模型参数估计流程如下。

(1) 输入：观测数据 $\{\boldsymbol{x}_i\}_{i=1}^{n}$，正整数 k。

(2) 输出：模型参数 $\boldsymbol{\theta}$。

① 设置初始值 $\boldsymbol{\theta}^{(0)}=(\pi_j^{(0)},\boldsymbol{\mu}_j^{(0)},\boldsymbol{\Sigma}_j^{(0)} \mid j=1,\cdots,k)$；

② E 步：利用第 t 次迭代参数 $\boldsymbol{\theta}^{(t)}$ 的值根据式(3.90)计算 $q_{ij}(\boldsymbol{\theta}^{(t)})$，即

$$q_{ij}(\boldsymbol{\theta}^{(t)})=\frac{\pi_j\,\mathcal{N}(\boldsymbol{x}_i,\boldsymbol{\mu}_j^{(t)},\boldsymbol{\Sigma}_j^{(t)})}{\sum_{j=1}^{k}\pi_j\,\mathcal{N}(\boldsymbol{x}_i,\boldsymbol{\mu}_j^{(t)},\boldsymbol{\Sigma}_j^{(t)})}$$

③ M 步：由 $q_{ij}(\boldsymbol{\theta}^{(t)})$ 的值依次更新参数，即

$$\boldsymbol{\mu}_j^{(t+1)}=\frac{\sum_{i=1}^{n} q_{ij}(\boldsymbol{\theta}^{(t)})\boldsymbol{x}_i}{\sum_{i=1}^{n} q_{ij}(\boldsymbol{\theta}^{(t)})}$$

$$\boldsymbol{\Sigma}_j^{(t+1)}=\frac{\sum_{i=1}^{n} q_{ij}(\boldsymbol{\theta}^{(t)})(\boldsymbol{x}_i-\boldsymbol{\mu}_j)^{\mathrm{T}}(\boldsymbol{x}_i-\boldsymbol{\mu}_j)}{\sum_{i=1}^{n} q_{ij}(\boldsymbol{\theta}^{(t)})}$$

$$\pi_j^{(t+1)}=\frac{\sum_{i=1}^{n} q_{ij}(\boldsymbol{\theta}^{(t)})}{n}$$

④ 若 $\|\boldsymbol{\theta}^{(t+1)}-\boldsymbol{\theta}^{(t)}\|<\varepsilon$ 或 $\|Q(\boldsymbol{\theta}^{(t+1)})-Q(\boldsymbol{\theta}^{(t)})\|<\varepsilon$，则停止迭代；否则，重复第②步和第③步。

以上算法不仅能求出 GMM 的各个参数，还记录下了每个样本属于各个高斯分布的概率值 $q_{ij}(\boldsymbol{\theta}^{(t)})$，可以根据 $\underset{j}{\arg\max}\, q_{ij}(\boldsymbol{\theta}^{(t)})$，认定样本来自哪个高斯分布，由此可以应用 GMM 于数据聚类。

3.5.3　实战

1. 数据集

使用表 3.4 中的西瓜数据 4.0 实现高斯混合模型(GMM)。

表 3.4　数据集

序号	x_1	x_2	序号	x_1	x_2	序号	x_1	x_2	序号	x_1	x_2
1	0.697	0.46	9	0.666	0.091	17	0.719	0.103	25	0.525	0.369
2	0.744	0.376	10	0.243	0.267	18	0.359	0.188	26	0.751	0.489
3	0.634	0.264	11	0.245	0.057	19	0.339	0.241	27	0.532	0.472
4	0.608	0.318	12	0.343	0.099	20	0.282	0.257	28	0.473	0.376
5	0.556	0.215	13	0.639	0.161	21	0.748	0.232	29	0.725	0.445
6	0.403	0.237	14	0.657	0.198	22	0.714	0.346	30		
7	0.481	0.149	15	0.36	0.37	23	0.483	0.312			
8	0.437	0.211	16	0.593	0.042	24	0.478	0.437			

2. Sklearn 实现

```
# coding:utf8
# 高斯混合模型,使用 EM 算法
# 数据集:《机器学习》-- 西瓜数据 4.0,文件 watermelon4.txt
import numpy as np
import matplotlib.pyplot as plt
```

```
# 预处理数据
def loadData(filename):
  dataset = []
  fr = open(filename)
  for line in fr.readlines():
     curLine = line.strip().split('\t ')
     fltLine = [float(item) for item in curLine]
     dataset.append(fltLine)
  return dataset
# 高斯分布的概率密度函数
def prob(x, mu, sigma):
  n = np.shape(x)[1]
  expOn = float(-0.5 * (x - mu) * (sigma.I) * ((x - mu).T))
    # np.linalg.det 计算矩阵的行列式
  divBy = pow(2 * np.pi, n / 2) * pow(np.linalg.det(sigma), 0.5)
  return pow(np.e, expOn) / divBy
# EM 算法
def EM(dataMat, maxIter = 50):
  m, n = np.shape(dataMat)
    # 1.初始化各高斯混合成分参数
  alpha = [1 / 3, 1 / 3, 1 / 3] # 1.初始化 alpha1 = alpha2 = alpha3 = 1/3
   # 2.初始化 mu1 = x6,mu2 = x22,mu3 = x27
  mu = [dataMat[5, :], dataMat[21, :], dataMat[26, :]]
  sigma = [np.mat([[0.1, 0], [0, 0.1]]) for x in range(3)]  # 3.初始化协方差矩阵
  gamma = np.mat(np.zeros((m, 3)))
  for i in range(maxIter):
     for j in range(m):
         sumAlphaMulP = 0
         for k in range(3):
             gamma[j, k] = alpha[k] * prob(dataMat[j, :], mu[k], sigma[k]) # 4.计算混合成
                                                                   #分生成的后验概率,即 gamma
             sumAlphaMulP += gamma[j, k]
         for k in range(3):
             gamma[j, k] /= sumAlphaMulP
     sumGamma = np.sum(gamma, axis = 0)
     for k in range(3):
         mu[k] = np.mat(np.zeros((1, n)))
         sigma[k] = np.mat(np.zeros((n, n)))
         for j in range(m):
             mu[k] += gamma[j, k] * dataMat[j, :]
         mu[k] /= sumGamma[0, k] # 5.计算新均值向量
         for j in range(m):
             sigma[k] += gamma[j, k] * (dataMat[j, :] - mu[k]).T * (dataMat[j, :] - mu[k])
         sigma[k] /= sumGamma[0, k] # 6. 计算新的协方差矩阵
         alpha[k] = sumGamma[0, k] / m # 7. 计算新混合系数
          # print(mu)
  return gamma
# 随机选取中心点
def initCentroids(dataMat, k):
  numSamples, dim = dataMat.shape
  centroids = np.zeros((k, dim))
  for i in range(k):
     index = int(np.random.uniform(0, numSamples))
```

```
            centroids[i, :] = dataMat[index, :]
        return centroids
    def gaussianCluster(dataMat):
        m, n = np.shape(dataMat)
        centroids = initCentroids(dataMat, m)
        clusterAssign = np.mat(np.zeros((m, 2)))
        gamma = EM(dataMat)
        for i in range(m):
            # amx 返回矩阵最大值,argmax 返回矩阵最大值所在下标
            # 9.确定 x 的簇标记 lambda
            clusterAssign[i, :] = np.argmax(gamma[i, :]), np.amax(gamma[i, :])
        for j in range(m):
            pointsInCluster = dataMat[np.nonzero(clusterAssign[:, 0].A == j)[0]]
            centroids[j, :] = np.mean(pointsInCluster, axis=0) # 计算出均值向量
        return centroids, clusterAssign
    def showCluster(dataMat, k, centroids, clusterAssment):
        numSamples, dim = dataMat.shape
        if dim != 2:
            print("Sorry! I can not draw because the dimension of your data is not 2!")
            return 1
        mark = ['or', 'ob', 'og', 'ok', '^r', '+r', 'sr', 'dr', '<r', 'pr']
        if k > len(mark):
            print("Sorry! Your k is too large!")
            return 1
            # draw all samples
        for i in range(numSamples):
            markIndex = int(clusterAssment[i, 0])
            plt.plot(dataMat[i, 0], dataMat[i, 1], mark[markIndex])
        mark = ['Dr', 'Db', 'Dg', 'Dk', '^b', '+b', 'sb', 'db', '<b', 'pb']
        # 绘制聚类中心
        for i in range(k):
            plt.plot(centroids[i, 0], centroids[i, 1], mark[i], markersize=12)
        plt.show()
    if __name__ == "__main__":
        dataMat = np.mat(loadData('.txt'))
        centroids, clusterAssign = gaussianCluster(dataMat)
        print(clusterAssign)
        showCluster(dataMat, 3, centroids, clusterAssign)
```

EM 算法的实现效果如图 3.26 所示。

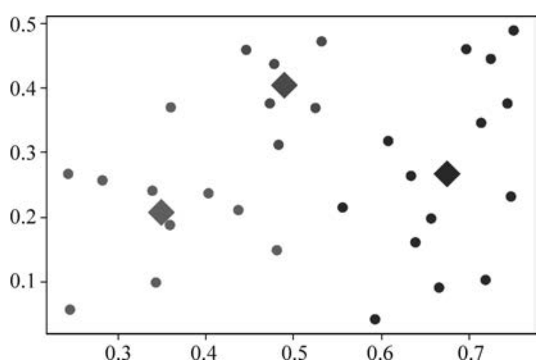


图 3.26　EM 算法的实现效果

3. 自编代码实现

```
# -*- coding: utf-8 -*-
# 使用 EM 算法解算 GGM ,EM 算法采用 scikit-learn 包提供的 api
# 数据集:《机器学习》-- 西瓜数据 4.0,文件 watermelon4.txt
from sklearn import mixture
import matplotlib.pyplot as plt
import numpy as np
# 预处理数据
def loadData(filename):
  dataset = []
  fr = open(filename)
  for line in fr.readlines():
     curLine = line.strip().split('\t ')
     fltLine = [float(item) for item in curLine]
     dataset.append(fltLine)
  return dataset

def test_GMM(dataMat, components=3,iter = 100,cov_type="full"):
  clst = mixture.GaussianMixture(n_components=n_components,max_iter=iter,covariance_
type=cov_type)
  clst.fit(dataMat)
  predicted_labels =clst.predict(dataMat)
  return clst.means_,predicted_labels     # clst.means_返回均值
def showCluster(dataMat, k, centroids, clusterAssment):
  numSamples, dim = dataMat.shape
  if dim != 2:
     print("Sorry! I can not draw because the dimension of your data is not 2!")
     return 1
  mark = ['or', 'ob', 'og', 'ok', '^r', '+r', 'sr', 'dr', '<r', 'pr']
  if k > len(mark):
     print("Sorry! Your k is too large!")
     return 1
     # draw all samples
  for i in range(numSamples):
     markIndex = int(clusterAssment[i])
     plt.plot(dataMat[i, 0], dataMat[i, 1], mark[markIndex])
  mark = ['Dr', 'Db', 'Dg', 'Dk', '^b', '+b', 'sb', 'db', '<b', 'pb']
  # draw the centroids
  for i in range(k):
     plt.plot(centroids[i, 0], centroids[i, 1], mark[i], markersize=12)
  plt.show()
if __name__=="__main__":
  dataMat = np.mat(loadData('.txt'))
  n_components = 3
  iter=100
  cov_types = ['spherical', 'tied', 'diag', 'full']
  centroids,labels = test_GMM(dataMat,n_components,iter,cov_types[3])
  showCluster(dataMat, n_components, centroids, labels) # 这里 labels 的维度改变了,注意修
# 改 showCluster()方法
```

自编代码的实现效果如图 3.27 所示。

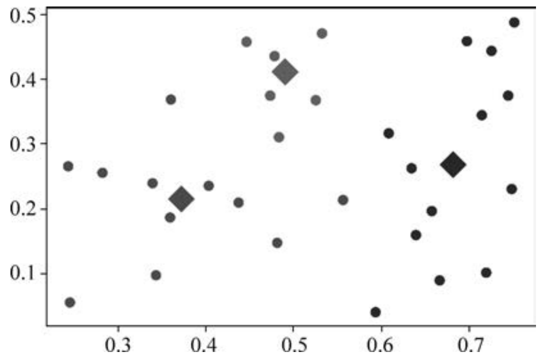


图 3.27 自编代码的实现效果

3.5.4 实验

1. 实验目的

判断 EM 算法与最大似然估计的关系。

2. 实验数据

考虑一个投掷硬币的实验：有两枚特制的硬币 A 和 B，这两枚硬币和普通的硬币不一样。A 和 B 投掷出正面的概率分别记为 θ_A 和 θ_B。现在独立地做 6 次试验：随机从这两枚硬币中抽取 1 枚，投掷 10 次，统计出现正面的次数。那么就得到了如表 3.5 所示的实验结果。

表 3.5 实验结果

试验代号	正面次数	硬币
1	3	B
2	9	A
3	8	A
4	4	B
5	7	A

在这个实验中，记录两组随机变量 $X=(3,9,8,4,7)$，$Z=(Z^1,Z^2,Z^3,Z^4,Z^5)$，其中，Z^i 辨识第 i 次实验选择的硬币，$Z^i=(1,2,\cdots,5)$，分别表示选择的是 A 还是 B。

3. 实验要求

本实验的目标是通过这个实验来估计 $\theta=(\theta_A;\theta_B)$ 的数值。实验中的参数估计就是有完整数据的参数估计，这是因为我们不仅仅知道每次实验中投掷出正面的次数，还知道每次实验中投掷的是硬币 A 还是硬币 B。

一个很简单也很直接的估计 θ 的方法如下(其实是最大似然估计的结果)：

$$\hat{\theta}_A=\frac{\text{用硬币 A 投掷出正面的次数}}{\text{用硬币 A 投掷的次数}}=\frac{24}{30}$$

$$\hat{\theta}_B=\frac{\text{用硬币 B 投掷出正面的次数}}{\text{用硬币 B 投掷的次数}}=\frac{10}{30}$$

问：如果不知道表 3.3 中的第三列，即标量 Z 为未知变量，如何估计 θ_A 和 θ_B？

3.6 BP 神经网络的分类和回归算法

3.6.1 原理简介

2012 年 AlexNet 的提出，大大降低了 ImageNet 数据集上图像识别的错误率，神经网络的研究又一次被推上了热潮。由于网络的结构越来越复杂、层次越来越深，Hinton 大师提出了一个新的名词——深度学习(deep learning)——来区别传统的神经网络。从此，深度学习引来了爆发性发展，在语音、图像、视频等各类复杂任务中均获得了前所未有的准确率。深度学习的基础是 BP 神经网络，BP 为 Back Propagation 的简写，最早是由 Rumelhart、McCelland 等科学家于 1986 年提出的，是一种按照误差逆向传播算法训练的多层前馈神经网络，经过不断的改进、更新，已成为应用最为广泛的神经网络模型。为了保持算法的经典性和可读性，本章在理论部分沿用了 UFDL(用户文件说明程序库)中的基本术语和符号表示。

1. 神经元模型

神经元是神经网络构成的基本单元，为更好地描述神经网络，先描述神经元结构。神经元本质上是仅有一个神经元的最简单的神经网络，如图 3.28 所示。

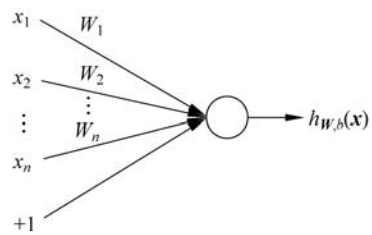


图 3.28 神经元结构

神经元是一个以多个变量 $x_1,x_2,\cdots,x_n$ 及偏置项 b(图 3.28 中的+1 符号)为输入的运算单元，其输出为

$$h_{\boldsymbol{W},b}(\boldsymbol{x})=f(\boldsymbol{W}^{\mathrm{T}}\boldsymbol{x})=f\left(\sum_{i=1}^{n}W_i x_i+b\right) \tag{3.95}$$

其中，n 为输入变量的个数，函数 $f:\mathbb{R}\mapsto\mathbb{R}$ 称为激活函数(activation function)，它负责将神经元的输入映射到输出端，其作用是在神经网络中引入非线性，强化网络的学习能力。常用的激活函数有 Sigmoid、双曲正切(tanh)及 ReLU 等。

Sigmoid 函数如下：

$$f(z)=\frac{1}{1+\exp(-z)} \tag{3.96}$$

Sigmoid 函数的优点是值域为(0,1)，可以放到模型最后一层，作为模型的概率输出，例如，单神经元构成的输入－输出映射关系本质上是一个逻辑回归(logistic regression)，目标检测模型 YoLov3 同时做多目标分类时，也是使用 Sigmoid 作为最后一层。缺点是容易产生梯度消失，涉及指数运算，计算量大。

双曲正切(tanh)函数：

$$f(z)=\tanh(z)=\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \tag{3.97}$$

$\tanh(z)$函数是 Sigmoid 函数的一种变体，它的取值范围为[－1,1]，仍然存在梯度消失问题；涉及指数运算，缺点是复杂度高一些，优点是函数均值为 0，可作为图像生成的最后一层激活函数。

ReLU 函数如下：

$$f(x)=\max(0,x) \tag{3.98}$$

ReLU 函数的优点是梯度和计算量小,可以得到稀疏激活的神经网络。经常用于神经网络的中间层激活函数。缺点是函数均值不为 0,当 $x<0$ 时梯度为 0,不利于反向传播。

2. BP 神经网络模型及参数表示

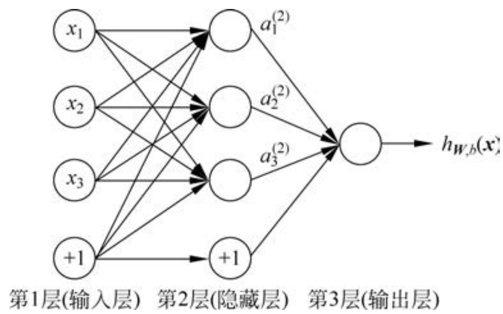


图 3.29 具有三层结构的神经网络

神经网络是将多个神经元连接在一起的网络结构,一个神经元的输出可以作为其他神经元的输入。BP 神经网络指的是神经网络中应用 BP 算法进行权重修正的网络,它构成的连接图没有闭环或回路。图 3.29 给出了一个具有输入层、隐藏层和输出层的简单三层结构神经网络,其中的圆圈表示神经元,标上"+1"的圆圈被称为偏置节点,也就是截距项。一般地,将神经网络最左边的一层定义为输入层,它作为特殊的神经元处理。最右的一层定义为输出层,中间所有节点组成的各层均是隐藏层,隐藏层可以有多个。图 3.29 中所示的网络,输入层由 3 个输入单元组成,隐藏层由 3 个隐藏单元组成,输出层由一个输出单元组成。

为方便描述和计算,对一个含有多个隐藏层的网络结构,用 n_l 来表示网络的层数,第 l 层记为 L_l,一般 L_1 是输入层,L_{n_l} 是输出层。用 s_l 表示第 l 层的节点数。$a_i^{(l)}$ 表示第 l 层第 i 单元的激活值(输出值)。网络中需要训练的参数包括权重 $\boldsymbol{W}$ 和偏置项 b,其中 $W_{ij}^{(l)}$ 是第 l 层第 j 单元与第 $l+1$ 层第 i 单元之间连接的权重,$b_i^{(l)}$ 是第 $l+1$ 层第 i 单元的偏置项。图 3.29 中的网络层数 $n_l=3$,需要训练的参数 $(\boldsymbol{W},b)=(\boldsymbol{W}^{(1)},b^{(1)},\boldsymbol{W}^{(2)},b^{(2)})$,其中 $\boldsymbol{W}^{(1)}\in\mathbb{R}^{3\times3}$,$\boldsymbol{W}^{(2)}\in\mathbb{R}^{1\times3}$。当 $l=1$ 时,$a_i^{(1)}=x_i$,即第 i 个输入值(输入值的第 i 个特征)。

3. 前向传播

无论是回归还是分类任务,最终是要通过迭代方法求出模型中的参数集合 $\boldsymbol{W}$、b,迭代求参数就是不断重复前向传播(forward propagation)和反向传播这两个过程,直到神经网络中的参数趋于稳定。给定参数 $\boldsymbol{W}$、b,前向传播就可以按照每个神经元函数 $h_{\boldsymbol{W},b}(\boldsymbol{x})$ 从输入层到输出层依次计算输出结果。以图 3.29 中三层结构的神经网络为例,其前向传播计算如式(3.99)至(3.102)所示。

$$a_1^{(2)}=f(W_{11}^{(1)}x_1+W_{12}^{(1)}x_2+W_{13}^{(1)}x_3+b_1^{(1)}) \tag{3.99}$$

$$a_2^{(2)}=f(W_{21}^{(1)}x_1+W_{22}^{(1)}x_2+W_{23}^{(1)}x_3+b_2^{(1)}) \tag{3.100}$$

$$a_3^{(2)}=f(W_{31}^{(1)}x_1+W_{32}^{(1)}x_2+W_{33}^{(1)}x_3+b_3^{(1)}) \tag{3.101}$$

$$h_{\boldsymbol{W},b}(\boldsymbol{x})=a_1^{(3)}=f(W_{11}^{(2)}a_1^{(2)}+W_{12}^{(2)}a_2^{(2)}+W_{13}^{(2)}a_3^{(2)}+b_1^{(2)}) \tag{3.102}$$

如果用 $z_i^{(l)}$ 表示第 l 层第 i 单元的输入加权和(包括偏置单元),例如,$z_i^{(2)}=\sum_{j=1}^{n}W_{ij}^{(1)}x_j+b_i^{(1)}$,则 $a_i^{(l)}=f(z_i^{(l)})$。

如果将激活函数 $f(\cdot)$ 扩展为用向量(分量的形式)表示,即 $f([\boldsymbol{z}_1,\boldsymbol{z}_2,\boldsymbol{z}_3])=[f(\boldsymbol{z}_1),f(\boldsymbol{z}_2),f(\boldsymbol{z}_3)]$,则上面的等式可以更简洁地表示为

$$\boldsymbol{z}^{(2)}=\boldsymbol{W}^{(1)}\boldsymbol{x}+b^{(1)} \tag{3.103}$$

$$a^{(2)}=f(z^{(2)}) \tag{3.104}$$

$$z^{(3)}=W^{(2)}a^{(2)}+b^{(2)} \tag{3.105}$$

$$h_{W,b}(x)=a^{(3)}=f(z^{(3)}) \tag{3.106}$$

上面的计算步骤叫作前向传播。令 $a^{(1)}=x$ 表示输入层的激活值，则前向传播可统一表示为

$$z^{(l+1)}=W^{(l)}a^{(l)}+b^{(l)} \tag{3.107}$$

$$a^{(l+1)}=f(z^{(l+1)}) \tag{3.108}$$

其中，$a^{(l)}$、$a^{(l+1)}$ 分别为第 l 层、第 $l+1$ 层的激活值。

前文以三层网络结构直观讲述了前向传播的过程，对于具有更多隐藏层的 BP 神经网络，其前向传播的过程方法完全一样：从输入层 L_1 开始，逐一计算第 L_2 层的所有激活值，然后是第 L_3 层的激活值，以此类推，直到第 L_{n_l} 层，即得到了神经网络的输出。

4. 反向传播

BP 算法通过前向传播和反向传播（back propagation）不断迭代，每轮迭代根据神经网络的输出与真值的差来调整网络参数 W、b，直到网络趋于稳定。因此 BP 算法属于监督学习，要求每个样本数据具有真值标签。假定数据集的第 k 个样本及真值标签用 $(x^{(k)}, y^{(k)})$ 表示，其中 $k=1,2,\cdots,m$，m 表示样本的数量，$x^{(k)}=[x_1^{(k)},x_2^{(k)},\cdots,x_n^{(k)}]$，$n$ 为输入数据的维度，$y^{(k)}=[y_1^{(k)},y_2^{(k)},\cdots,y_c^{(k)}]$，$c$ 为输出数据的维度。则求权重 W 和偏置项 b 的过程如下：①初始化参数 W 和 b 为很小的、接近零的随机值；②给定某个样本输入，使用前向传播得到网络输出；③计算神经网络的输出结果和给定真值标签的误差；④根据误差进行反向传播，更新权重 W 和偏置项参数 b。

步骤①②在前向传播环节实现，步骤③需要定义模型的代价函数，计算网络输出和真值标签之间的误差。以回归任务为例，只有一个输出单元的神经网络，如波士顿房价预测案例，经典的代价函数（cost function）如式(3.109)所示，该代价函数由均方误差和权重衰减项两部分组成，均方误差的目标是通过误差最小化使神经网络的输出尽量和真值标签靠近，而权重衰减项属于 L_2 正则化，目标是使权值收敛到较小的绝对值，而惩罚大的权值，防止系统出现过拟合现象，提升网络的泛化性能，参数 λ 为权重衰减系数，控制权重衰减项的重要性。

$$\begin{aligned}J(W,b)&=\left[\frac{1}{m}\sum_{k=1}^{m}J(W,b;x^{(k)},y^{(k)})\right]+\frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ji}^{(l)})^2\\&=\left[\frac{1}{m}\sum_{k=1}^{m}\left(\frac{1}{2}\|h_{W,b}(x^{(k)})-y^{(k)}\|^2\right)\right]+\frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ji}^{(l)})^2\end{aligned} \tag{3.109}$$

其中

$$J(W,b;x^{(k)},y^{(k)})=\frac{1}{2}\|h_{W,b}(x^{(k)})-y^{(k)}\|^2 \tag{3.110}$$

表示单个样本的代价函数。

反向传播的方法是：采用梯度下降算法，从输出层到输入层反向依次计算代价函数对参数的偏导数值，根据偏导数值对各参数进行修改，网络结构中的每个参数的更新如式(3.111)和式(3.112)所示，公式中的参数 α 为神经网络的学习速率。

$$W_{ij}^{(l)}=W_{ij}^{(l)}-\alpha\frac{\partial}{\partial W_{ij}^{(l)}}J(\boldsymbol{W},b) \tag{3.111}$$

$$b_i^{(l)}=b_i^{(l)}-\alpha\frac{\partial}{\partial b_i^{(l)}}J(\boldsymbol{W},b) \tag{3.112}$$

5. 权重更新公式推导

BP算法推导的关键是定义残差变量和偏导逐级反向传播。先计算出最后一层的残差，通过利用相邻两层神经元间的数据关系（$z_i^{(l)}$ 与 $z_j^{(l+1)}$）建立偏导和残差之间的关系以及相邻两层残差之间的关系，从而计算出各隐藏层代价函数对权重和偏置项参数的偏导（即式(3.111)和式(3.112)中的$\frac{\partial}{\partial W_{ij}^{(l)}}J(\boldsymbol{W},b)$和$\frac{\partial}{\partial b_i^{(l)}}J(\boldsymbol{W},b)$），以下给出推导过程，在推导过程中会给出残差的定义。

根据式(3.111)和式(3.112)，权重更新的关键是需要计算代价函数对权重和偏置项的偏导数，将式(3.109)分别代入式(3.111)和式(3.112)，可得到式(3.113)和式(3.114)。

$$\frac{\partial}{\partial W_{ij}^{(l)}}J(\boldsymbol{W},b)=\left[\frac{1}{m}\sum_{k=1}^{m}\frac{\partial}{\partial W_{ij}^{(l)}}J(\boldsymbol{W},b;\boldsymbol{x}^{(k)},\boldsymbol{y}^{(k)})\right]+\lambda\boldsymbol{W}_{ij}^{(l)} \tag{3.113}$$

$$\frac{\partial}{\partial b_i^{(l)}}J(\boldsymbol{W},b)=\frac{1}{m}\sum_{k=1}^{m}\frac{\partial}{b_i^{(l)}}J(\boldsymbol{W},b;\boldsymbol{x}^{(k)},\boldsymbol{y}^{(k)}) \tag{3.114}$$

为了描述方便，在下文推导过程中，分别用$\frac{\partial J}{\partial W_{ij}^{(l)}}$、$\frac{\partial J}{\partial b_i^{(l)}}$表示$\frac{\partial}{\partial W_{ij}^{(l)}}J(\boldsymbol{W},b;\boldsymbol{x}^{(k)},\boldsymbol{y}^{(k)})$、$\frac{\partial}{\partial b_i^{(l)}}J(\boldsymbol{W},b;\boldsymbol{x}^{(k)},\boldsymbol{y}^{(k)})$，先求解对权重的偏导$\frac{\partial J}{\partial W_{ij}^{(l)}}$。

如果令 $\delta_j^{(l+1)}=\frac{\partial J}{\partial z_j^{(l+1)}}$，$\delta_j^{(l+1)}$ 定义为第 $l+1$ 层第 j 个神经元的残差。

又因为

$$z_j^{(l+1)}=\sum_{k=1}^{s_l}W_{jk}^{(l)}a_k^{(l)}+b_j^{(l+1)}=\sum_{k=1}^{s_l}W_{jk}^{(l)}f(z_k^{(l)})+b_j^{(l+1)} \tag{3.115}$$

可得

$$\frac{\partial z_j^{(l+1)}}{\partial W_{ij}^{(l)}}=a_j^{(l)}$$

则

$$\frac{\partial J}{\partial W_{ij}^{(l)}}=\frac{\partial J}{\partial z_j^{(l+1)}}\frac{\partial z_j^{(l+1)}}{\partial W_{ij}^{(l)}}=\frac{\partial z_j^{(l+1)}}{\partial W_{ij}^{(l)}}\frac{\partial J}{\partial z_j^{(l+1)}}=a_j^{(l)}\delta_i^{(l+1)} \tag{3.116}$$

同理可得

$$\frac{\partial J}{\partial b_i^{(l)}}=\frac{\partial J}{\partial z_j^{(l+1)}}\frac{\partial z_j^{(l+1)}}{\partial b_i^{(l)}}=\frac{\partial z_j^{(l+1)}}{\partial b_i^{(l)}}\frac{\partial J}{\partial z_j^{(l+1)}}=\delta_i^{(l+1)} \tag{3.117}$$

下面推导根据第 $l+1$ 层的残差计算第 l 层的残差的公式。

由式(3.115)得出$\frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}}=W_{ji}^{(l)}f'(z_i^{(l)})$，则

$$\begin{aligned}\delta_i^{(l)} &= \frac{\partial J}{\partial z_i^{(l)}} = \sum_{j=1}^{s_{l+1}} \frac{\partial J}{\partial z_j^{(l+1)}} \frac{\partial z_j^{(l+1)}}{\partial z_i^{(l)}} = \sum_{j=1}^{s_{l+1}} \delta_j^{(l+1)} W_{ji}^{(l)} f'(z_i^{(l)}) \\ &= \sum_{j=1}^{s_{l+1}} (W_{ji}^{(l)} \delta_j^{(l+1)}) f'(z_i^{(l)})\end{aligned} \tag{3.118}$$

有了残差计算的递推公式(见式(3.118))和代价函数对参数的偏导计算公式(见式(3.116)和式(3.117))仍然不够,还需要计算最后一层的残差,有了最后一层残差的数值,才能根据式(3.116)～式(3.118)计算各层的偏导数,从而对参数进行更新。由于神经网络的最后一层有输出,因此可以根据输出和真值标签数据得到残差：

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} J(\boldsymbol{W}, b; \boldsymbol{x}^{(k)}, \boldsymbol{y}^{(k)}) = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \| \boldsymbol{y}^{(k)} - h_{\boldsymbol{W},b}(\boldsymbol{x}^{(k)}) \|^2 \tag{3.119}$$

对于神经网络输出层,$h_{\boldsymbol{W},b}(x_j^{(k)}) = a_j^{(n_l)} = f(z_j^{(n_l)})$,因此：

$$\text{式(3.119)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \sum_{j=1}^{s_{n_l}} (y_j^{(k)} - f(z_j^{(n_l)}))^2 \tag{3.120}$$

该偏导数为累和求导的问题,只有当 $j=i$ 时的求和项,导数才存在,当 $j \neq i$ 时导数都为 0,因此：

$$\begin{aligned}\text{式(3.120)} &= -(y_i^{(k)} - f(z_i^{(n_l)})) f'(z_i^{(n_l)}) \\ &= -(y_i^{(k)} - a_i^{(n_l)}) f'(z_i^{(n_l)})\end{aligned} \tag{3.121}$$

由于 $y_i^{(k)}$ 为已知真值标签,$f(\cdot)$为激活函数,$a_i^{(n_l)}$、$f'(z_i^{(n_l)})$均可通过前向传播计算,因此输出层某个单元的残差 $\delta_i^{(n_l)}$ 可以计算,结合式(3.111)至式(3.114),式(3.116)至式(3.118)即可进行逐层反向传播,对参数进行更新。

3.6.2 算法步骤

1. 算法步骤

BP 神经网络算法属于监督学习(supervised learning)范畴,即带有标签的学习,包括正向传播和反向传播两个过程。正向传播环节的目标是求损失,在这个环节,根据输入样本,给定初始化权值 $\boldsymbol{W}$ 和偏置项 b,计算最终输出值以及输出值与真实值(标签)之间的损失值。如果损失值未达到设定的条件则进行反向传播并更新参数 $\boldsymbol{W}$、b,否则停止 $\boldsymbol{W}$、b 的更新。而反向传播的任务是进行误差的回传,它将输出以某种形式通过隐藏层向输入层逐层反传,并将误差分摊到各层的所有单元,根据各层单元的误差信号对各单元权值和偏置项参数进行修正。

具体来讲,完成一个 BP 神经网络算法程序主要包含以下几部分内容。

(1) 数据准备。数据对神经网络模型的性能有很大影响。同样一组数据,在不同的数据空间有着不同的分布,因此,提取有差异化的特征、选择合适的数据表达可以有效降低解决问题的难度、提升神经网络模型的性能。

(2) 神经网络模型定义。不同结构的神经网络在不同的问题下得到的效果不同,应该针对具体问题,选择合适的神经网络结构。

（3）神经网络模型训练。使用给定的训练数据集训练神经网络。通过前向传播和反向传播更新模型中的参数，直到模型趋于稳定或达到预先设定的条件。如果模型训练时无法趋于稳定，必要时需要调整神经网络模型结构。

（4）模型预测。使用训练好的神经网络模型对测试集或未知数据进行预测。

具体过程可以由图 3.30 表示。

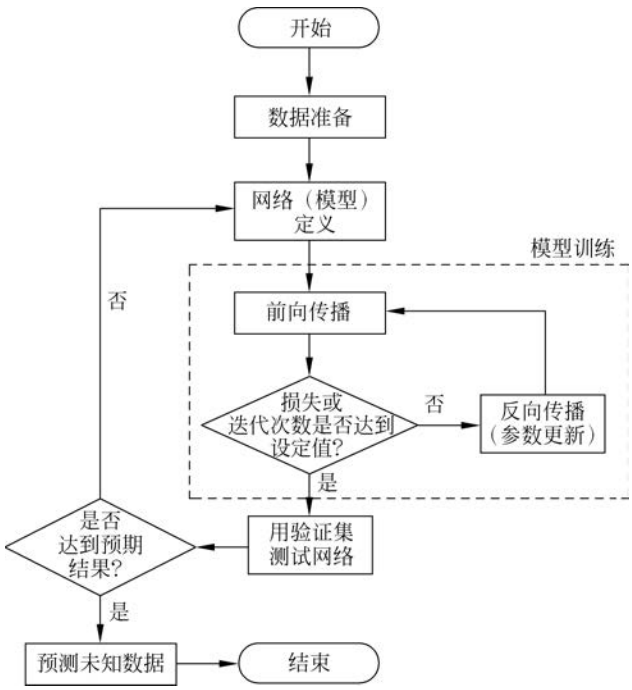


图 3.30　BP 神经网络实施流程图

2. 算法实现

3.6.1 节已经给出了 BP 神经网络的前向传播、反向传播的公式推导，以及 BP 神经网络算法的步骤过程。为了让读者更清晰地理解权重更新过程，3.6.1 节“前向传播”中的大部分公式都是逐层基于每个神经元进行计算的，如式(3.99)至式(3.102)，逐层计算每个神经元的输出，直到最后一层得到整个神经网络的输出，“权重更新公式推导”中也是针对每个权值或偏置项求解。最直观、容易理解的 BP 算法就是在程序中采用多层循环依次进行前向传播和反向传播过程，在前向传播和反向传播过程，逐层计算每个神经元的输出或梯度更新。然而多层循环的计算方式计算效率低，不能充分利用 GPU 资源和并行计算，因而更多的实现是基于矩阵运算的方式进行 BP 算法的实现。具有一个隐藏层的 BP 算法实现鸢尾花数据(参考 3.1 节)分类的代码如下：

```
import sklearn.datasets as ds
import sklearn.preprocessing as prep
import sklearn.model_selection as ms
import numpy as np

# Sigmoid 激活函数
```

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

# Sigmoid 函数求倒数
def sigmoidDerv(x):
  return x * (1 - x)

# 将 BP 神经网络封装为一个类
class BpNeuralNet:
   # 定义只有一个隐藏层,包括输入层、输出层共三层的神经网络,本质上是定义层与层之间的权值
# W 和偏置项 b
  def __init__(self,layers):
     self.W1 = np.random.random((layers[0],layers[1])) * 2 - 1
     self.W2 = np.random.random((layers[1],layers[2])) * 2 - 1
     self.b1 = np.random.random((1,layers[1])) * 2 - 1
     self.b2 = np.random.random((1,layers[2])) * 2 - 1

  def netTrain(self,train,labelTrain,test,labelTest,lr = 0.1,epochs = 1000): #lr 为学习率,
#epochs 为迭代的次数
     # 使用随机梯度下降进行权值训练更新
     for n in range(epochs + 1):
          i = np.random.randint(train.shape[0])#随机选取一行数据(一个样本)进行更新
          x = train[i]
          x = np.atleast_2d(x) # 维度变化

          L1_out = sigmoid(np.dot(x,self.W1) + self.b1)
          L2_out = sigmoid(np.dot(L1_out,self.W2) + self.b2)

          # 根据公式求梯度 delta
          L2_delta = (labelTrain[i] - L2_out) * sigmoidDerv(L2_out)
          L1_delta = L2_delta.dot(self.W2.T) * sigmoidDerv(L1_out)

          # 更新参数
          self.W2 += lr * L1_out.T.dot(L2_delta)
          self.W1 += lr * x.T.dot(L1_delta)
          self.b2 += lr * L2_delta
          self.b1 += lr * L1_delta

          # 每训练 1000 次预测准确率,使用 test 数据验证当前模型效果
          if n % 1000 == 0:
              predictions = []
              for j in range(test.shape[0]):
                 out = self.netPredict(test[j])
                 predictions.append(np.argmax(out))
              accuracy = np.mean(np.equal(predictions,labelTest))
              print('epoch:',n,'accuracy:',accuracy)

  def netPredict(self,x):
     L1 = sigmoid(np.dot(x,self.W1)++self.b1)#隐藏层输出
     L2 = sigmoid(np.dot(L1,self.W2)++self.b2)#输出层输出
     return L2
    # 神经网络封装 end
    # 主程序部分
```

```
    # 使用 Sklearn 载入数据,并进行归一化
    irisData = ds.load_iris() # 载入数据
    data_x = irisData.data # 数据
    target_y = irisData.target # 真值标签

    data_x -= data_x.min()
    data_x /= data_x.max()

    # 划分训练集和测试集
    # 训练集占 80 %,测试集占 20 %
    train_X,test_X,train_Y,test_Y = ms.train_test_split(data_x,target_y,test_size = 0.2)

    # 对输出标签进行 one - hot 编码
    labels_train = prep.LabelBinarizer().fit_transform(train_Y)

    # 创建神经网络并训练,[4,32,3]表示输入层、隐藏层、输出层的单元个数,鸢尾花数据集的每个
#样本有 4 个特征,共有 3 类不同的花
neuralModel = BpNeuralNet([4,32,3])
neuralModel.netTrain(train_X,labels_train,test_X,test_Y,epochs = 20000)
```

3.6.3 实战

本节分别以鸢尾花数据集和波士顿房价数据集来说明 BP 神经网络如何应用于分类和回归任务。无论是分类还是回归,神经网络也可以有多个输出单元。例如,鸢尾花分类任务输出层具有三个输出单元对应这三类不同的鸢尾属植物,波士顿房价回归任务输出层仅有房价一个输出单元。

1. 数据集介绍

(1) 分类数据集选择经典的鸢尾花数据集,其详细介绍参见 2.1.4 节。

(2) 回归数据集采用波士顿房价数据集,可通过 sklearn.datasets.load_boston 加载相关数据。

波士顿房价数据集是一个回归问题。每个类的观察值数量是均等的,共有 506 个观察值,13 个输入变量和 1 个输出变量。

每条数据包含房屋以及房屋周围的详细信息。其中包含城镇犯罪率、一氧化氮浓度、住宅平均房间数、到中心区域的加权距离以及自住房平均房价等。波士顿房价数据集的属性描述如表 3.6 所示。

表 3.6 波士顿房价数据集的属性描述

属性名	解释	类型
CRIM	该镇的人均犯罪率	连续值
ZN	占地面积超过 $2778m^2$ 的住宅用地比例	连续值
INDUS	非零售商业用地比例	连续值
CHAS	是否邻近查尔斯河	离散值,1=邻近;0=不邻近
NOX	一氧化氮浓度	连续值
RM	每栋房屋的平均客房数	连续值
AGE	1940 年之前建成的自用单位比例	连续值

续表

属 性 名	解 释	类 型
DIS	到波士顿5个就业中心的加权距离	连续值
RAD	到径向公路的可达性指数	连续值
TAX	全值财产税率	连续值
PTRATIO	学生与教师的比例	连续值
B	$1000(BK-0.63)^2$,其中BK为黑人占比	连续值
LSTAT	低收入人群占比	连续值
MEDV	同类房屋价格的中位数	连续值

鸢尾花数据集和波士顿房价数据集都是机器学习中的经典数据集,获取它们的途径非常多,正规途径有两种：其一,UCI机器学习网站下载；其二,利用Python中的机器学习包,如scikit-learn、Pandas等直接导入。

2. 应用第三方工具实现分类和回归

实现了BP算法的第三方平台有很多,使用不同的平台需要的代码量也有区别。本节将依次使用Sklearn和Keras来示范如何使用BP神经网络完成分类和回归任务。

1) 使用Sklearn进行分类和回归

Sklearn的全称为scikit-learn,Sklearn是其包名。Sklearn是一个由Python第三方提供的机器学习库,包含了从数据预处理到训练模型的各个方面。使用Sklearn可以极大地节省编写代码的时间并减少代码量,使人们有更多的精力去关注数据内在的本质,分析数据分布,进行模型更新和超参调整。Sklearn中用于分类和回归的BP神经网络模型是MLPClassifier()和MLPRegressor()函数,其中MLPClassifier()函数有22个参数,MLPRegressor()函数有23个参数,两个函数的前22个参数基本一致,涵盖了定义多层神经网络所需要的隐藏层、激活函数、优化求解器、学习率等,完全可以覆盖对不同需求多层神经网络的定义,在定义神经网络时,对于未设置的参数,系统将自动用默认值进行填充,详细参数说明请参考官方网站。

(1) MLPClassifier()函数用于鸢尾花分类。

MLPClassifier()函数是sklearn.neural_network中的函数,它是利用反向传播误差进行计算的多层感知器算法。完整代码如下,主要包括数据准备、网络定义、网络训练、网络预测和结果评估几个环节。

```
import numpy as np
from functools import reduce
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier

def main():
  #数据准备:使用Sklearn获取数据,data和target数据组合在一起便于随机打乱
  IrisDataSet = datasets.load_iris()
  dataAndTarget = np.zeros(shape=(150,5))
  dataAndTarget[:,:4] = np.array(IrisDataSet['data'])
  dataAndTarget[:,4] = np.array(IrisDataSet['target'])
```

```
    #随机打乱数据,前100个数据作为训练,后50个数据作为测试
    np.random.shuffle(dataAndTarget)
    x_train = dataAndTarget[:100,:4]
    y_train = dataAndTarget[:100,4]
    x_test = dataAndTarget[100:150,:4]
    y_test = dataAndTarget[100:150,4]

    #分别对训练测试数据进行标准化
    scaler = StandardScaler() # 标准化转换
    scaler.fit(x_test) # 训练标准化对象
    x_test_Standard = scaler.transform(x_test)     scaler.fit(x_train) # 训练标准化对象
    x_train_Standard = scaler.transform(x_train)

    #网络(模型)定义:使用MLPClassifier()函数建立神经网络分类模型
    bpNetwork = MLPClassifier(hidden_layer_sizes = (500, ),activation = 'relu', solver = 'lbfgs',
alpha = 0.0001,batch_size = 'auto', learning_rate = 'constant')

    #模型训练:调用fit()函数使用训练集训练网络
    bpNetwork.fit(x_train_Standard,y_train.astype('int'))

    #模型预测:使用训练好的模型对测试数据进行预测
    y_predict = bpNetwork.predict(x_test_Standard)

    y_test = y_test.tolist()
    y_test = list(map(int, y_test))
    y_predict = list(y_predict)

    #结果评估:计算预测错误的数量
    comResult = list(map(lambda x: x[0] - x[1], zip(y_test, y_predict)))
    falsePredict = reduce(lambda x,y:abs(x) + abs(y),comResult)

    print("Ground Truth:\t",y_test)
    print("BP Predicted:\t",y_predict)
    print("Accuracy:",(1 - falsePredict/50) * 100,"%")

      if __name__ == "__main__":
        main()
```

（2）MLPRegressor()函数用于波士顿房价预测。

```
import numpy as np
from math import sqrt
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from sklearn.neural_network import MLPRegressor

def main():
    ##数据准备:使用Sklearn获取数据,data和target数据组合
    BostonDataSet = datasets.load_boston()
    dataAndTarget = np.zeros(shape = (506,140))
    dataAndTarget[:,:13] = np.array(BostonDataSet['data'])
```

```
    dataAndTarget[:,13] = np.array(BostonDataSet['target'])

    #随机打乱,前 400 个数据作为训练,后 106 个数据作为测试
    np.random.shuffle(dataAndTarget)
    x_train = dataAndTarget[:400,:13]
    y_train = dataAndTarget[:400,13]
    x_test = dataAndTarget[400:506,:13]
    y_test = dataAndTarget[400:506,13]


    #分别对训练测试数据进行标准化
    scaler = StandardScaler()
    scaler.fit(x_test)
    x_test_Standard = scaler.transform(x_test)
    scaler.fit(x_train)
    x_train_Standard = scaler.transform(x_train)

    ##网络(模型)定义:使用 MLPRegressor()函数建立神经网络分类模型
    regNetwork = MLPRegressor(hidden_layer_sizes = (500, ), activation = 'relu', solver = 'lbfgs',
alpha = 0.0001, batch_size = 'auto', learning_rate = 'constant')

    #模型训练:调用 fit()函数使用训练集训练网络
    regNetwork.fit(x_train_Standard,y_train)

    #模型预测:使用训练好的模型对测试数据进行预测
    y_predict = regNetwork.predict(x_test_Standard)

    #结果评估:计算预测和真值的均方根误差
    y_test = y_test.tolist()
    y_predict = list(y_predict)
    rms = sqrt(mean_squared_error(y_test, y_predict))

    print("Ground Truth:\t",y_test)
    print("BP Predicted:\t",y_predict)
    print("Root Mean Square Error: \t",rms)

if __name__ == "__main__":
    main()
```

2）使用 Keras 进行分类和回归

Keras 是一个用 Python 编写的高级神经网络 API(应用程序编程接口),它能够以 TensorFlow、CNTK,或者 Theano 作为后端运行。Keras 的优点是用户友好,高度模块化,可扩展,开发效率高,支持当前主流的网络结构,如卷积神经网络和循环神经网络,可以在 CPU 和 GPU 上无缝运行。同样的,代码仍然包括数据准备、网络定义、网络训练、网络预测和结果评估几个环节,与 Sklearn 的差别在于网络定义和网络训练环节更灵活,所定义网络的复杂性可以更高。

(1) Keras 用于鸢尾花分类。

```
import numpy as np
from tensorflow.keras import models
```

```
from tensorflow.keras import layers
from sklearn import datasets
from tensorflow import keras


#数据准备:通过 Sklearn 获取数据
IrisDataSet = datasets.load_iris()
dataAndTarget = np.zeros(shape = (150,5))
dataAndTarget[:,:4] = np.array(IrisDataSet['data'])
dataAndTarget[:,4] = np.array(IrisDataSet['target'])

#随机打乱数据,前 100 个数据作为训练,后 50 个数据作为测试
np.random.shuffle(dataAndTarget)
x_train = dataAndTarget[:100,:4]
y_train = dataAndTarget[:100,4]
x_test = dataAndTarget[100:150,:4]
y_test = dataAndTarget[100:150,4]

#softmax 要求对输出的分类标签进行 one-hot 编码
one_hot_train = keras.utils.to_categorical(y_train)
one_hot_test = keras.utils.to_categorical(y_test)


# 对特征数据进行标准化,train 和 test 数据使用同一个标准
mean = x_train.mean(axis = 0)
x_train -= mean
std = x_train.std(axis = 0)
x_train /= std
x_test -= mean
x_test /= std

# 定义函数,该函数的作用是创建神经网络模型
def newBpClsModel():
  model = models.Sequential()
  model.add(layers.Dense(8, activation = 'relu',input_shape = (x_train.shape[1],)))
  model.add(layers.Dense(3, activation = 'softmax'))

  model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])
  return model

#网络(模型)定义
clsModel = newBpClsModel()

#网络训练
history = clsModel.fit(x_train, one_hot_train, epochs = 200, batch_size = 16, verbose = 0,
validation_data = (x_test, one_hot_test))

#网络预测及结果评估
test_loss_score, test_acc_score = clsModel.evaluate(x_test, one_hot_test)
test_out_hot = clsModel.predict(x_test)
test_out = np.argmax(test_out_hot)

print('test_acc_score: ', test_acc_score) # Mean Squared Error
```

(2) Keras 用于波士顿房价预测。

```
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras.datasets import boston_housing

#数据准备:使用 Keras 直接得到数据
(train_data, train_targets), (test_data, test_targets) = boston_housing.load_data()

# 对特征数据进行标准化,train 和 test 数据使用同一个标准
mean = train_data.mean(axis = 0)
train_data -= mean
std = train_data.std(axis = 0)
train_data /= std
test_data -= mean
test_data /= std

# 定义函数用于新建网络模型
def newBpRegModel():
  model = models.Sequential()
  model.add(layers.Dense(32, activation = 'relu',input_shape = (train_data.shape[1],)))
  model.add(layers.Dense(64, activation = 'relu'))
  model.add(layers.Dense(64, activation = 'relu'))
  # 最后一层不要加激活函数,输出任意房价数值
  model.add(layers.Dense(1))
  # MSE (mean squared error,均方误差),MAE(平均绝对误差) 有绝对值
  model.compile(optimizer = 'rmsprop', loss = 'mse', metrics = ['mae'])
  return model

#网络(模型)定义
regModel = newBpRegModel()

#网络训练
history = regModel.fit(train_data, train_targets, epochs = 200, batch_size = 16, verbose = 0,
validation_data = (test_data, test_targets))
mae_history = history.history['val_mae']

#网络预测及结果评估
test_mse_score, test_mae_score = regModel.evaluate(test_data, test_targets)
test_out = regModel.predict(test_data)
print('test_mae_score: ', test_mae_score) # Mean Squared Error
```

3.6.4 实验

1. 实验目的

(1) 掌握 BP 神经网络的原理及梯度更新推导方法;

(2) 能应用 BP 算法解决实际问题。

2. 实验数据

MINST 数据集。它是机器学习领域中非常经典的一个数据集,由 60 000 个训练样本和 10 000 个测试样本组成,每个样本都是一张 28×28 像素的灰度手写数字图片。

3. 实验要求

（1）基于现有的主流平台 TensorFlow、PyTorch、Keras 等。

（2）构建一个四层神经网络模型，隐藏层神经单元个数及超参数根据经验调整。

（3）按 MINST 数据集已定义好的训练集、测试集对模型进行训练及测试，使测试集上的准确率不低于 96%。

3.7 卷积神经网络分类算法

3.7.1 原理简介

卷积神经网络（convolutional neural network，CNN）是一类包含卷积计算且具有深度结构的前馈神经网络（feedforward neural network），是受生物自然视觉认知机制启发而来的。现在，卷积神经网络已经成为众多科学领域的研究热点之一，特别是在模式分类领域，该网络避免了对图像进行复杂的前期预处理，可以直接输入原始图像，因而得到了更为广泛的应用。可应用于图像分类、目标识别、目标检测、语义分割等。本节介绍可用于图像分类的卷积神经网络的基本结构。

卷积神经网络由三部分构成。第一部分是输入层；第二部分由 N 个卷积层和池化层的组合组成；第三部分由一个全连接的多层感知机分类器构成。

图 3.31 是一个简单的卷积神经网络结构图，第一层输入图片进行卷积（convolution）操作。得到第二层深度为 3 的特征图（feature map）。对第二层的特征图进行池化（pooling）操作，得到第三层深度为 3 的特征图。重复上述操作得到第五层深度为 5 的特征图，最后将 5 个特征图按行展开连接成向量，传入全连接（fully connected）层。图中的每个特征图都可以看成是排列成矩阵形式的神经元，下面是卷积和池化的计算过程。

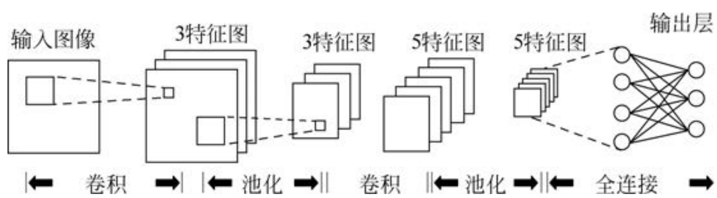


图 3.31 一个简单的卷积神经网络结构图

1. 卷积

对于一张输入图片，将其转换为矩阵，矩阵的元素为对应的像素值。如图 3.32 所示，对于一个 5×5 的图像，使用一个 3×3 的卷积核进行卷积，可得到一个 3×3 大小的特征图，其中卷积核也称为滤波器（filter）。输入图片经过卷积后所得特征图大小的计算公式为

$$w' = \frac{w + 2p - k}{s} + 1$$

其中，w 表示图片大小；p 表示填充；k 表示卷积核大小；s 表示步幅。

详细过程如图 3.33 所示。其中框选出的区域表示卷积核在输入矩阵中从上到下、从左到右滑动，每滑动到一个位置，将对应数字相乘并求和，得到一个特征图矩阵的元素。注意到，动图中卷积核每次滑动了一个单位，也就是步幅（stride）为 1，实际上滑动的幅度可以根

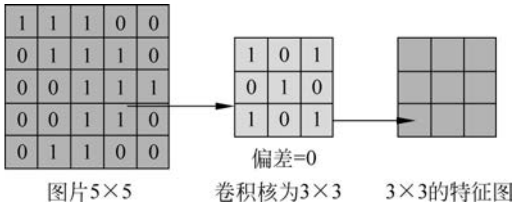


图 3.32 卷积神经网络的卷积过程图

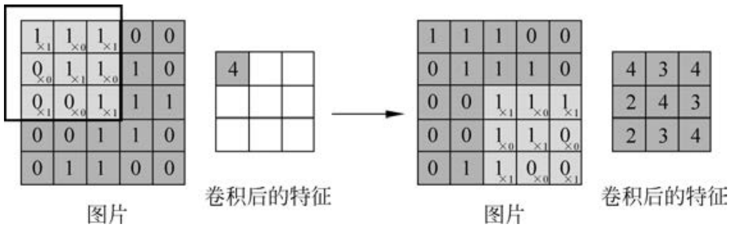


图 3.33 卷积步骤详解图

据需要进行调整。如果滑动步幅大于 1,则卷积核有可能无法恰好滑到边缘,针对这种情况,可在矩阵最外层补零,即用参数填充(padding),补零层(zero padding),是一个可以设置的超参数,但要根据卷积核的大小、步幅,输入矩阵的大小进行调整,以使卷积核恰好滑动到边缘。

2. 池化

池化又叫下采样(down sampling)。图像中的相邻像素倾向于具有相似的值,因此通常卷积层相邻的输出像素也具有相似的值。这意味着卷积层输出中包含的大部分信息都是冗余的。如果使用边缘检测滤波器并在某个位置找到强边缘,那么也可能会在距离这个像素 1 个偏移的位置找到相对较强的边缘。但是它们都一样是边缘,我们并没有找到任何新东西。池化层解决了这个问题。这个网络层所做的就是通过减小输入的大小降低输出值的数量。

卷积得到的特征图一般需要一个池化层以降低数据量。一般不在池化操作中使用 padding 操作,则池化后所得特征图大小的计算公式为

$$\frac{n-f}{s}+1$$

池化的操作步骤如图 3.34 所示。

池化层的滑动可被称为滑动窗口,图 3.34 中滑动窗口的大小为 2×2,步幅为 2,每滑动到一个区域,则取最大值作为输出,这样的操作称为最大池化(max pooling),还可以采用输出均值的方式,被称为平均池化(mean pooling)。

3. 全连接

经过若干层的卷积、池化操作后,将得到的特征图依次按行展开,连接成向量,输入全连接网络,如图 3.35 所示。

卷积神经网络与普通神经网络的区别在于,卷积神经网络包含了一个由卷积层和子采样层构成的特征抽取器。在卷积神经网络的卷积层中,一个神经元只与部分邻层神经元连接。在卷积神经网络的一个卷积层中,通常包含若干特征平面(feature map),每个特征平

面由一些矩形排列的神经元组成，同一特征平面的神经元共享权值，这里共享的权值就是卷积核。卷积核一般以随机小数矩阵的形式初始化，在网络的训练过程中卷积核将学习得到合理的权值。共享权值(卷积核)带来的直接好处是减少网络各层之间的连接，同时又降低了过拟合的风险。子采样也叫作池化(pooling)，通常有均值子采样和最大值子采样两种形式。子采样可以看作一种特殊的卷积过程。卷积和子采样大大简化了模型的复杂度，减少了模型的参数。

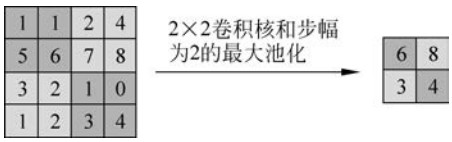


图 3.34　池化步骤详解图

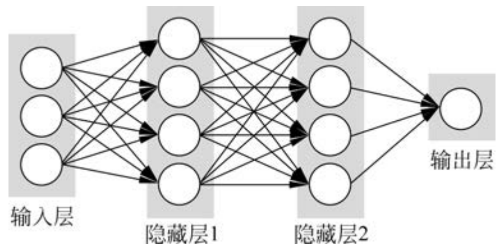


图 3.35　全连接步骤示意图

3.7.2　算法步骤

卷积神经网络的算法流程主要包括数据集的划分和处理、卷积神经网络的训练、模型测试验证、分类四部分，其卷积神经网络的训练包含输入层、卷积层、池化层和全连接层，如图 3.36 所示。

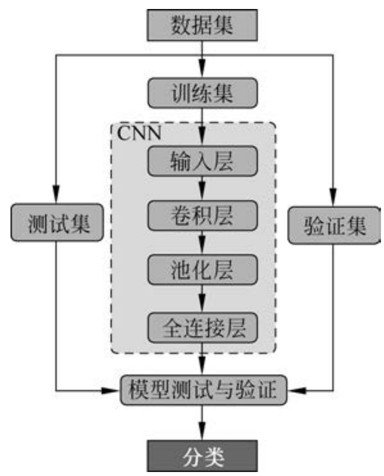


图 3.36　卷积神经网络算法流程图

以分类任务为例，对训练目标数据集，卷积神经网络执行了以下操作：首先对目标数据集进行划分，分别划分为训练集、验证集和测试集。在训练集上训练模型，在验证集上做模型选择，最后为了测试已经训练好的模型的精确度，在测试集上进行模型评估。

在模型训练过程中，先对网络进行权值的初始化，输入数据计算出结果，比较结果计算误差并进行反向传播更新网络中的权值和阈值，直到满足给定正确率后保存网络模型参数。使用训练好的模型计算测试集中的目标数据，得到其数据集的分类标签，完成分类任务。

(1) 输入：目标数据集。

(2) 算法步骤。

① 划分训练集、验证集和测试集。

② 选定训练组，从样本集中分别随机地寻求 N 个样本作为训练组。

③ 卷积网络训练模型。

(3) 训练过程。

① 将各权值、阈值，置成小的接近于 0 的随机值，并初始化精度控制参数和学习率。

② 从训练组中取一个批次数据加到网络，并给出它的目标输出向量。

③ 计算出中间层输出向量，计算出网络的实际输出向量。

④ 将输出向量中的元素与目标向量中的元素进行比较，计算出输出误差；对于中间层

的隐单元也需要计算出误差。

⑤ 依次计算出各权值的调整量和阈值的调整量。

⑥ 调整权值和阈值。

⑦ 当经历 M 次循环后，判断指标是否满足精度要求，如果不满足，则返回③，继续迭代；如果满足就进入下一步。

⑧ 训练结束，将权值和阈值保存在文件中。

（4）分类过程。

① 加载模型权值、阈值文件。

② 从待分类数据集中取一个批次的数据加到网络中。

③ 计算出中间层输出向量，计算出网络的实际输出向量。

④ 将输出向量中通过 softmax 激活函数输出 0、1 整数确定所属类别。

3.7.3 实战

1. MINST 手写数字数据集

如图 3.37 所示，MINST 数据库是由 Yann 提供的手写数字数据库文件，数据集来自美国国家标准与技术研究所（national institute of standards and technology，NIST）。主要包含了 60 000 张的训练图像和 10 000 张的测试图像，训练集（training set）和测试集（test set）均由来自 250 个不同人手写的数字构成，其中 50% 是高中学生，50% 来自人口普查局（the Census Bureau）的工作人员。包含以下四部分。

图 3.37 MINST 数据集展示

（1）Training set images：train-images-idx3-ubyte.gz（9.9MB，解压后为 47MB，包含 60 000 个样本）。

（2）Training set labels：train-labels-idx1-ubyte.gz（29KB，解压后为 60KB，包含 60 000 个标签）。

（3）Test set images：t10k-images-idx3-ubyte.gz（1.6MB，解压后为 7.8MB，包含 10 000 个样本）。

（4）Test set labels：t10k-labels-idx1-ubyte.gz（5KB，解压后为 10KB，包含 10 000 个标签）。

数据库里的图像都是 28×28 大小的灰度图像，每像素对应一个 8 位字节(0～255)。

2. Sklearn 实现

```
#------------------------导入环境运行库--------------------------------
import tensorflow as tf
import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
import sklearn
import os
import sys
import time
from tensorflow import keras
#数据加载
minst = keras.datasets.mnist
img_rows, img_cols = 28, 28 # 图片大小为 28×28
(x_train, y_train), (x_test, y_test) = minst.load_data()
# 数据维度重建,以便网络模型训练
if keras.backend.image_data_format() == 'channels_first':
   x_train = x_train.reshape(x_train.shape[0],1,img_rows, img_cols)
   x_test = x_test.reshape(x_test.shape[0],1,img_rows,img_cols)
   input_shape = (1, img_rows, img_cols)
else:
   x_train = x_train.reshape(x_train.shape[0],img_rows,img_cols, 1)
   x_test = x_test.reshape(x_test.shape[0],img_rows,img_cols, 1)
   input_shape = (img_rows, img_cols, 1)
#数据预处理
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
# 数据归一化,方便计算
x_train = x_train / 255
x_test = x_test / 255
# 样本标签转换为 0-1 矩阵,便于分类
y_train_onehot = tf.keras.utils.to_categorical(y_train)
y_test_onehot = tf.keras.utils.to_categorical(y_test)
# 调用 Keras 中的 Sequential 序贯模型搭建网络
model = tf.keras.Sequential()
# 配置卷积层(Conv2D)并设置卷积核个数、卷积核大小、激活函数等参数
# 首层卷积层需定义输入数据维度大小
model.add(tf.keras.layers.Conv2D(32,kernel_size = (3,3),activation = 'relu', input_shape =
(28, 28, 1)))
# 添加池化层(MaxPooling2D)并设置池化核大小
model.add(tf.keras.layers.MaxPooling2D(pool_size = (2, 2)))
model.add(tf.keras.layers.Conv2D(64,kernel_size = (3,3),activation = 'relu'))
model.add(tf.keras.layers.MaxPooling2D(pool_size = (2, 2)))
# 添加全连接层并设置神经元个数和激活函数
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(128, activation = 'relu'))
model.add(tf.keras.layers.Dropout(0.5))
```

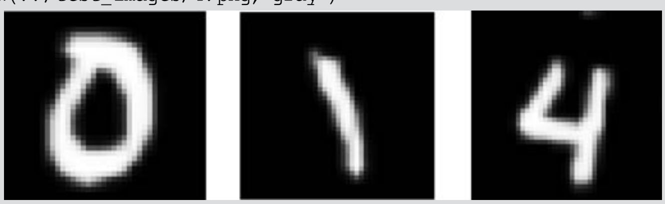
```
model.add(tf.keras.layers.Dense(10, activation='softmax'))
# 打印模型结构
model.summary()
# 设置优化器、损失函数和评价指标
model.compile(optimizer='adam',loss='categorical_crossentropy', metrics=['accuracy'])
# 模型训练,配置训练次数、批次大小、验证集
history=model.fit(x_train, y_train_onehot, batch_size=256, epochs= 10, verbose=1,
validation_data = (x_test, y_test_onehot))
# 模型评估
score = model.evaluate(x_test, y_test_onehot, verbose=0)
# 结果可视化
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

输出结果：

```
Epoch 1/10
235/235 [==============================] - 25s 102ms/step - loss: 0.4057 -
accuracy: 0.8764 - val_loss: 0.0742 - val_accuracy: 0.9768
Epoch 2/10
235/235 [==============================] - 24s 101ms/step - loss: 0.1146 -
accuracy: 0.9665 - val_loss: 0.0458 - val_accuracy: 0.9854
Epoch 3/10
235/235 [==============================] - 24s 102ms/step - loss: 0.0858 -
accuracy: 0.9749 - val_loss: 0.0396 - val_accuracy: 0.9877
Epoch 4/10
235/235 [==============================] - 24s 101ms/step - loss: 0.0696 -
accuracy: 0.9794 - val_loss: 0.0338 - val_accuracy: 0.9882
Epoch 5/10
235/235 [==============================] - 24s 103ms/step - loss: 0.0606 -
accuracy: 0.9812 - val_loss: 0.0321 - val_accuracy: 0.9888
Epoch 6/10
235/235 [==============================] - 25s 105ms/step - loss: 0.0541 -
accuracy: 0.9835 - val_loss: 0.0297 - val_accuracy: 0.9902
Epoch 7/10
235/235 [==============================] - 25s 104ms/step - loss: 0.0479 -
accuracy: 0.9858 - val_loss: 0.0261 - val_accuracy: 0.9904
Epoch 8/10
235/235 [==============================] - 24s 103ms/step - loss: 0.0418 -
accuracy: 0.9871 - val_loss: 0.0243 - val_accuracy: 0.9914
Epoch 9/10
235/235 [==============================] - 24s 103ms/step - loss: 0.0378 -
accuracy: 0.9887 - val_loss: 0.0263 - val_accuracy: 0.9914
Epoch 10/10
235/235 [==============================] - 24s 103ms/step - loss: 0.0355 -
accuracy: 0.9896 - val_loss: 0.0253 - val_accuracy: 0.9920

Test loss: 0.025268295779824257
Test accuracy: 0.991999983787536
    # 定义预测函数
    def predict(image_path):
    # 以黑白方式读取图片
```

```
    img = Image.open(image_path).convert('L')
    img = np.reshape(img, (28, 28, 1)) / 255.
    x = np.array([1 - img])
    y = model.predict(x)
    # np.argmax()取得最大值的下标,即代表的数字
    print(image_path)
    print(y[0])
    print('      -> Predict digit', np.argmax(y[0]))
    # 目标图片分类(输入目标图片路径)
    predict('../test_images/0.png')
    predict('../test_images/1.png')
    predict('../test_images/4.png')
    #待分类照片预览
    plt.imshow(../test_images/0.png,'gray')
    plt.imshow(../test_images/1.png,'gray')
    plt.imshow(../test_images/4.png,'gray')
```



```
    #结果显示
    ../test_images/0.png
    [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
          -> Predict digit 0
    ../test_images/1.png
    [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
          -> Predict digit 1
    ../test_images/4.png
    [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
          -> Predict digit 4
```

3.7.4　实验

1. 实验目的

掌握卷积神经网络算法的基本原理以及执行流程,并熟练运用该算法完成分类任务。

2. 实验数据集

Fashion-MNIST 是由 Zalando 旗下的研究部门提供的图像数据集,涵盖了来自 10 种类别的共 7 万个不同商品的正面图片,包括 t-shirt(T 恤)、trouser(裤子)、pullover(套衫)、dress(裙子)、coat(外套)、sandal(凉鞋)、shirt(衬衫)、sneaker(运动鞋)、bag(包)、ankle boot(短靴)。Fashion-MNIST 的格式和训练集/测试集的划分与原始的 MNIST 完全一致。60000/10000 的训练测试数据划分,28×28 的灰度图片,可直接用来测试机器学习和深度学习算法性能。部分原始图片如图 3.38 所示。

标注编号描述如下。

0：t-shirt/top(T 恤)。

1：trouser(裤子)。

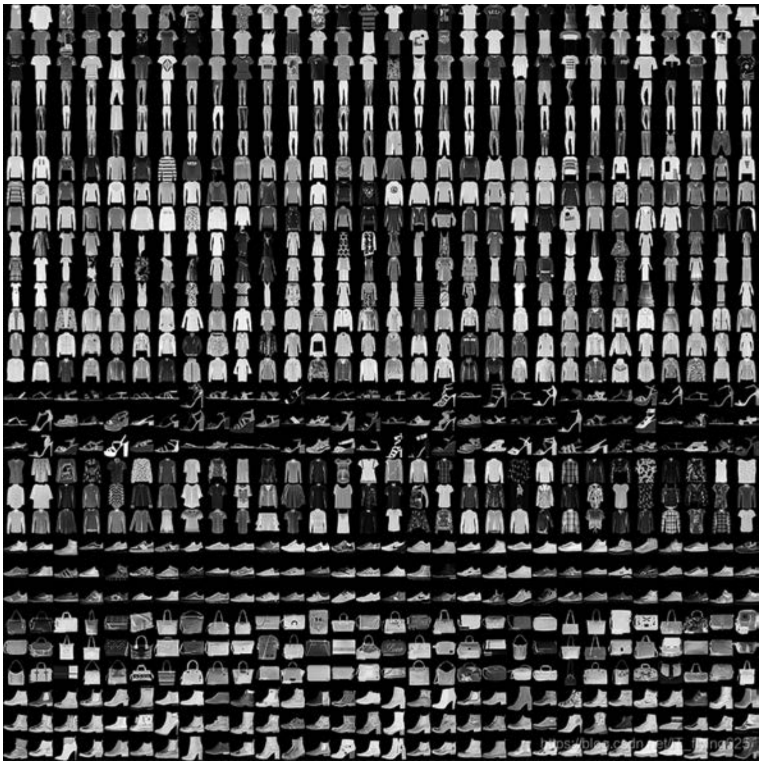


图 3.38 Fashion-MNIST 数据集展示

2：pullover(套衫)。

3：dress(裙子)。

4：coat(外套)。

5：sandal(凉鞋)。

6：shirt(衬衫)。

7：sneaker(运动鞋)。

8：bag(包)。

9：ankle boot(短靴)。

```
#模型编译和训练
model.compile(
  optimizer = 'adam',
  loss = 'sparse_categorical_crossentropy',
  metrics = ['acc']
)
history = model.fit(x_train, x_label, epochs = 100, batch_size = 1000)
```

显示结果：

```
Train on 10000 samples
Epoch 1/100
10000/10000 [ ============================== ] - 14s 1ms/sample - loss: 1.5669 -
acc: 0.4776
```

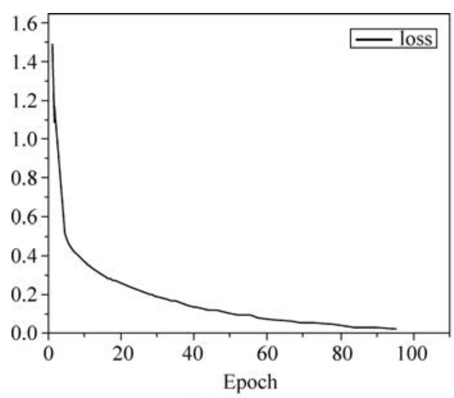
```
Epoch 2/100
10000/10000 [ ============================== ] - 15s 2ms/sample - loss: 0.7624 -
acc: 0.7458
Epoch 3/100
10000/10000 [ ============================== ] - 17s 2ms/sample - loss: 0.6300 -
acc: 0.7808
Epoch 4/100
2000/10000 [ =====>........................] - ETA: 11s - loss: 0.5707 - acc: 0.7960
```

绘制损失函数和准确率图像：

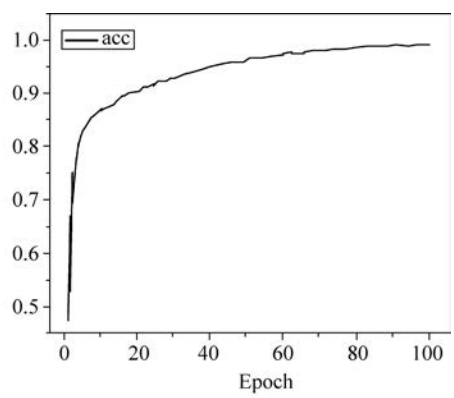
```
#绘制损失函数和准确率图像
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
hist['epoch'] = hist['epoch'] + 1

def plot_history(hist):
  plt.figure(figsize = (10,5))
  plt.subplot(1,2,1)
  plt.xlabel('Epoch')
  plt.plot(hist['epoch'], hist['loss'],
        label = 'loss')
  plt.legend()
  plt.subplot(1,2,2)
  plt.xlabel('Epoch')
  plt.plot(hist['epoch'], hist['acc'],
        label = 'acc',color = 'red')
  plt.legend()
plot_history(hist)
```

运行结果如图 3.39 所示。



(a) 损失函数曲线



(b) 准确率曲线

图 3.39 运行结果

模型评价及结果可视化的代码如下：

```
model.evaluate(test_features, test_labels, verbose = 2)
#模型分类结果及可视化
prediction = model.predict(test_features)
def plot_image(i, predictions_array, true_labels, images):
```

```
  predictions_array, true_label, img = predictions_array[i], true_labels[i], images[i]
  plt.grid(False)
  plt.xticks([])
  plt.yticks([])
  plt.grid(False)
  plt.imshow(images[i], cmap = plt.cm.binary)
  predicted_label = np.argmax(prediction[i])
  true_label = test_labels[i]
  if predicted_label == true_label:
    color = 'black'
  else:
    color = 'red'
plt.xlabel("预测{:2.0f}%是{}(实际{})".
format(100 * np.max(predictions_array),class_names[predicted_label],class_names[true_
label]),color = color)
def plot_value_array(i, predictions_array, true_label):
  predictions_array, true_label = predictions_array[i], true_label[i]
  plt.grid(False)
  plt.xticks(range(10))
  plt.yticks([])
  thisplot = plt.bar(range(10), predictions_array, color = "#777777")
  plt.ylim([0, 1])
  predicted_label = np.argmax(predictions_array)
  thisplot[predicted_label].set_color('red')
  thisplot[true_label].set_color('blue')
num_rows = 5
num_cols = 3
num_images = num_rows * num_cols
plt.figure(figsize = (2 * 2 * num_cols, 2 * num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2 * num_cols, 2 * i + 1)
  plot_image(i, prediction, test_labels, test_features)
  plt.subplot(num_rows, 2 * num_cols, 2 * i + 2)
plot_value_array(i, prediction, test_labels)
#结果显示(图片可视化)
```

模型评价的运行结果如图 3.40 所示。

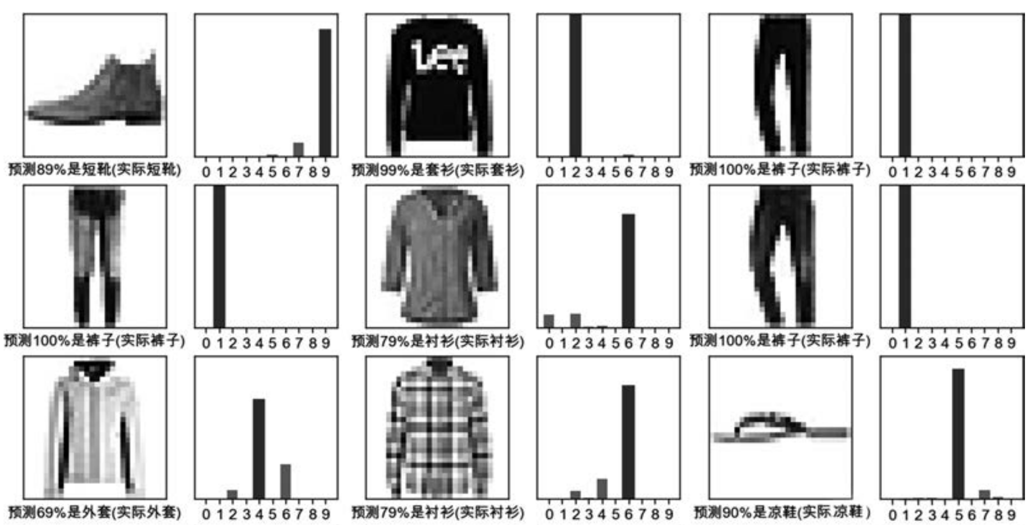


图 3.40　模型评价的运行结果

3.8 多类分类算法

3.8.1 原理简介

1. 基于二分类的多类分类方法

二分类是最简单的分类任务，多分类任务可分解为多个二分类任务，然后通过一定的集成完成多分类。考虑 N 个类别 $C_1, C_2, \cdots, C_n$，多分类学习的基本思路是"拆解法"，即将多分类任务拆为若干二分类任务来求解。具体来说，先对问题进行拆分，然后为拆出的每个二分类任务训练一个分类器。在测试时，对这些分类器的预测结果进行集成以获得最终的多分类结果。这里的关键是如何对多分类任务进行拆分，以及如何对多个分类器进行集成。本节主要介绍拆分策略。

最经典的拆分策略有三种："一对一"(one vs. one，OvO)、"一对其余"(one vs. rest，OvR)和"多对多"(many vs. many，MvM)。

给定数据集 $\boldsymbol{D}=\{(x_1,y_1),(x_2,y_2),\cdots,(x_m,y_m)\}$，$y_i\in\{C_1,C_2,\cdots,C_n\}$。OvO 将这 N 个类别两两配对，从而产生 $N(N-1)/2$ 个二分类任务，如 OvO 将为区分类别 C_i 和 C_j，训练一个分类器，该分类器把 $\boldsymbol{D}$ 中的 C_i 类样例作为正例，C_j 类样例作为反例。在测试阶段，新样本将同时提交给所有分类器，于是将得到 $N(N-1)/2$ 个分类结果，最终结果可通过投票产生，即把被预测得最多的类别作为最终分类结果。图 3.41 给出了一个示意图。

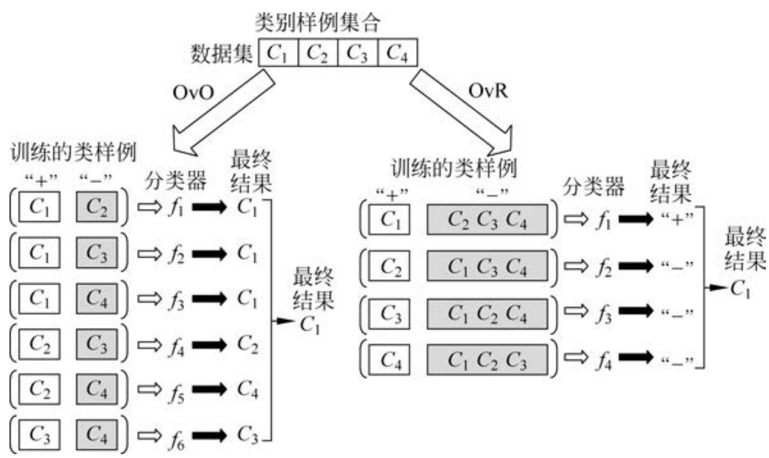


图 3.41 OvO 与 OvR 示意图

OvR 则是每次将一个类的样例作为正例、所有其他类的样例作为反例来训练 N 个分类器。在测试时若仅有一个分类器预测为正类，则对应的类别标记作为最终分类结果，如图 3.41 所示。若有多个分类器预测为正类，则通常考虑各分类器的预测置信度，选择置信度最大的类别标记作为分类结果。

容易看出，OvR 只需训练 N 个分类器，而 OvO 需训练 $N(N-1)/2$ 个分类器，因此，OvO 的存储开销和测试时间开销通常比 OvR 更大。但在训练时，OvR 的每个分类器均使用全部训练样例，而 OvO 的每个分类器仅用到两个类的样例，因此，在类别很多时，OvO 的训练时间开销通常比 OvR 更小。至于预测性能，则取决于具体的数据分布，在多数情形下

两者差不多。

MvM 是每次将若干类作为正类，若干其他类作为反类。显然，OvO 和 OvR 是 MvM 的特例。MvM 的正、反类构造必须有特殊的设计，不能随意选取。这里介绍一种最常用的 MvM 技术：纠错输出码(error correcting output codes，ECOC)。

ECOC 是将编码的思想引入类别拆分，并尽可能在解码过程中具有容错性。ECOC 的工作过程主要分为如下两步。

① 编码：对 N 个类别做 M 次划分，每次划分将一部分类别划为正类，一部分划为反类，从而形成一个二分类训练集。这样共产生 M 个训练集，可训练出 M 个分类器。

② 解码：M 个分类器分别对测试样本进行预测，这些预测标记组成一个编码。将这个预测编码与每个类别各自的编码进行比较，返回其中距离最小的类别作为最终预测结果。

类别划分通过编码矩阵(coding matrix)指定。编码矩阵有多种形式，常见的主要有二元码和三元码。前者将每个类别分别指定为正类和反类，后者在正、反类之外，还可指定"停用类"。图 3.42 给出了一个示意图，在图 3.42 (a)中，分类器 f_2 将 C_1 类和 C_3 类的样例作为正例，C_2 类和 C_4 类的样例作为反例；在图 3.42 (b)中，分类器 f_4 将 C_3 类和 C_4 类的样例作为正例，C_1 类的样例作为反例。在解码阶段，各分类器的预测结果联合起来形成了测试示例的编码，该编码与各类所对应的编码进行比较，将距离最小的编码所对应的类别作为预测结果。例如，在图 3.42 (a)中，若基于欧氏距离，预测结果将是 C_2。

	f_1	f_2	f_3	f_4	f_5	汉明距离	欧氏距离
C_1	−1	+1	+1	−1	+1	1	2
C_2	+1	−1	−1	+1	−1	4	4
C_3	−1	+1	−1	+1	+1	3	$2\sqrt{3}$
C_4	−1	−1	+1	+1	−1	2	$2\sqrt{2}$
示例	−1	−1	+1	−1	+1		

(a) 二元ECOC码

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	汉明距离	欧氏距离
C_1	+1	+1	−1	−1	−1	+1	−1	5	$2\sqrt{5}$
C_2	−1	0	0	0	+1	−1	0	2	2
C_3	−1	−1	+1	+1	−1	+1	+1	4	4
C_4	−1	+1	0	+1	−1	0	+1	3	$\sqrt{10}$
示例	−1	+1	+1	−1	+1	−1	+1		

(b) 三元ECOC码

图 3.42 ECOC 编码示意图

注：其中+1、−1 分别表示学习器将该样本作为正、反例，三元码中 0 表示不使用该类样本

2. 基于 softmax 函数的多类分类

深度学习时代，利用神经网络解决多分类问题时，常常采用 softmax 函数。softmax 一般加在网络的最后，其主要作用是对网络的输出进行对应类别的概率计算。softmax 函数的定义如下：

$$P_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}} \tag{3.122}$$

其中，z_i 为神经网络的输出，N 为网络输出的数量。该公式用于计算该元素的指数除以所有元素的指数和，取指数是为了使差别更大。经过 softmax 函数计算后，神经网络的每个元素被压缩到(0,1)，并且和为 1，即 P_i 可以看成每个输出端对应类别的分类概率。在进行分类时，此概率越大，被分为此类别的可能性就越大。限于篇幅，softmax 函数的分类实例请参考 3.7 节的说明。

3.8.2 算法步骤

1. 一对一

设训练集数据共有 M 个类，一对一方法是在每两个类之间构造一个二分类模型，例如，对第 i 类数据和第 j 类数据训练一个二分类 SVM 模型。

在测试阶段，采用投票策略进行分类。每个二分类 SVM 根据其决策函数对新数据 x_{new} 有一个预测(投票)，以 i 类和 j 类之间二分类 SVM 为例，若对 x_{new} 的预测为 i 类，则 i 类得票加 1；否则 j 类加 1；最终得票最多的类别就是对 x_{new} 的预测；若出现平票的情况，简单地选择索引较小的那个类别作为对 x_{new} 的分类。

2. 一对多

对于每个类，将其作为+1 类，而其余 $M-1$ 个类的所有样本作为−1 类，构造一个二分类模型，如 SVM。

在测试阶段，对于 x_{new}，M 个决策函数共有 M 个输出预测，若有唯一正类即选择其作为 x_{new} 的预测，若有多个正类则需要考虑各分类器的预测置信度，选择置信度最大的类别作为分类结果。

3. ECOC

ECOC-SVM 的算法步骤如图 3.43 所示，其中 M 指对数据所有类进行了 M 次划分，即共有 M 个分类器。

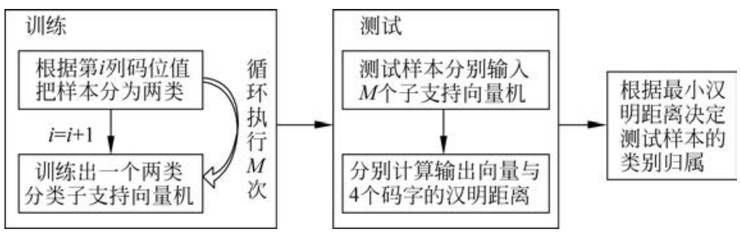


图 3.43 基于汉明距离的 ECOC-SVM 的训练及测试过程

3.8.3 实战

1. 数据集

采用经典的鸢尾花数据集，该数据集的详细介绍见 2.1.4 节。

2. Sklearn 实现

(1) 原理。

主要使用 Sklearn 上的 SVM 进行调包训练，例如，svm.SVC(kernel='rbf',gamma=0.1,C=0.8,decision_function_shape='ovo')，kernel='linear'时，为线性核，C 越大在训练集上的分类准确率越高，但有可能会过拟合(default C=1)。kernel='rbf'时(default)，为高斯核，gamma 值越小，分类界面越连续；gamma 值越大，分类界面越“散”，分类效果越好，但有可能会过拟合。decision_function_shape='ovr'时，为 one vs rest，即一个类别与其他类别进行划分，decision_function_shape='ovo'时，为 one vs one，即将类别两两之间进行划分，用

二分类的方法模拟多分类的结果。具体代码如下。

（2）导入模块。

```
# 导入模块
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib.colors import ListedColormap
from sklearn import svm, datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

（3）数据处理。

```
# 鸢尾花数据处理
iris = datasets.load_iris()
X = iris.data[:, :2] # 为便于绘图仅选择两个特征
y = iris.target # 类别

# 数据预处理：数据标准化
scaler = StandardScaler()
scaler.fit(X)
X = scaler.transform(X)

# 随机划分训练集与测试集，训练集占总数据的 60%(train_size = 0.6),random_state 是随机数
# 种子
x_train, x_test, y_train, y_test = train_test_split(X, y, random_state = 1, train_size = 0.6)
# 搭建模型，训练 SVM 分类器
# 调包实现一对多(OvR)
clf = svm.SVC(C=0.8, kernel='rbf', gamma=20, decision_function_shape='ovr')
# 调包实现一对一(OvO)
# clf = svm.SVC(kernel='rbf',gamma=0.1,C=0.8,decision_function_shape='ovo')

clf.fit(x_train, y_train.ravel()) #调用 ravel()函数将矩阵转变成一维数组
# 输出模型准确率
print("SVM-输出训练集的准确率为：", clf.score(x_train, y_train))
print("SVM-输出测试集的准确率为:",clf.score(x_test,y_test))
# 绘制图像
# 确定坐标轴范围，x、y 轴分别表示两个特征
x1_min, x1_max = X[:, 0].min(), X[:, 0].max()
# 第 0 列的范围 x[:, 0] ":"表示所有行，0 表示第 1 列
x2_min, x2_max = X[:, 1].min(), X[:, 1].max()
# 第 1 列的范围 x[:, 0] ":"表示所有行，1 表示第 2 列
x1, x2 = np.mgrid[x1_min:x1_max:200j, x2_min:x2_max:200j] # 生成网格采样点
grid_test = np.stack((x1.flat, x2.flat), axis=1)
# 先将网格点 x1 和 x2 展平为列向量，然后按列拼接两个列向量，构建测试样本点
grid_hat = clf.predict(grid_test)      # 预测分类值
grid_hat = grid_hat.reshape(x1.shape) # 使之与输入的形状相同
# 指定默认字体
mpl.rcParams['font.sans-serif'] = [u'SimHei']
mpl.rcParams['axes.unicode_minus'] = False
```

```
# 绘制
cm_light = mpl.colors.ListedColormap(['#A0FFA0', '#FFA0A0', '#A0A0FF'])
cm_dark = mpl.colors.ListedColormap(['g', 'r', 'b'])
plt.pcolormesh(x1, x2, grid_hat, cmap=cm_light)
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', s=50, cmap=cm_dark) # 样本
plt.scatter(x_test[:, 0], x_test[:, 1], s=120, facecolors='none', zorder=10) # 圈中测试
# 集样本
plt.xlabel(u'花萼长度', fontsize=13)
plt.ylabel(u'花萼宽度', fontsize=13)
plt.xlim(x1_min, x1_max)
plt.ylim(x2_min, x2_max)
plt.title(u'鸢尾花 SVM 二特征分类', fontsize=15)
plt.rcParams['figure.dpi'] = 2000 #分辨率
# plt.grid()
plt.show()

#由于准确率表现不直观,可以通过其他方式观察结果
#首先将原始结果与训练集预测结果进行对比
y_train_hat=clf.predict(x_train)
y_train_1d=y_train.reshape((-1))
print('鸢尾花训练集准确率:',clf.score(x_train,y_train))

#同样的,可以用训练好的模型对测试集的数据进行预测
#输出训练集的准确率
print('鸢尾花测试集准确率:',clf.score(x_test,y_test))
y_test_hat=clf.predict(x_test)
y_test_1d=y_test.reshape((-1))
comp=zip(y_test_1d,y_test_hat)
print('鸢尾花测试集实际结果与预测结果显示如下:',list(comp))


# 将测试集实际分类与预测分类进行可视化对比
# 还可以通过图像进行可视化
plt.figure()
plt.subplot(121)
plt.scatter(x_test[:,0],x_test[:,1],c=y_test.reshape((-1)),edgecolors='k',s=50)
plt.title(u'鸢尾花测试集实际分类情况', fontsize=15)
plt.subplot(122)
plt.scatter(x_test[:,0],x_test[:,1],c=y_test_hat.reshape((-1)),edgecolors='k',s=50)
plt.title(u'鸢尾花测试集 SVM 分类情况', fontsize=15)
plt.rcParams['figure.dpi'] = 1000 #分辨率
```

(4)运行结果。

一对一模型的准确率如下:

```
SVM-输出训练集的准确率为:0.7888888888888889
SVM-输出测试集的准确率为:0.75
```

一对一模型的鸢尾花分类结果如图3.44所示。

一对一模型中将测试集的实际结果与预测结果相对比,结果如下,作图可以得到如图3.45所示的结果。

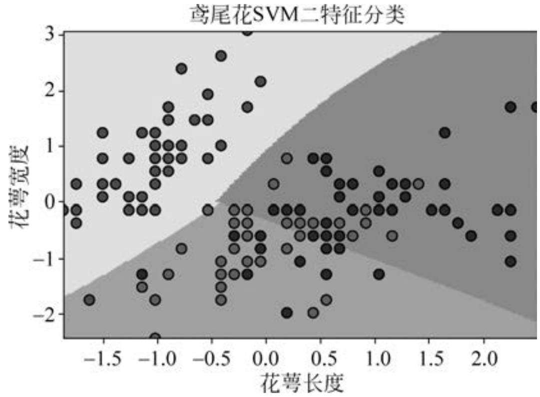


图 3.44　一对一模型的鸢尾花分类结果图

```
鸢尾花训练集准确率: 0.7888888888888889
鸢尾花测试集准确率: 0.75
鸢尾花测试集实际结果与预测结果显示如下: [(0, 0), (1, 1), (1, 2), (0, 0), (2, 2), (1, 2),
(2, 2), (0, 0), (0, 0), (2, 2), (1, 1), (0, 0), (2, 2), (1, 2), (1, 2), (0, 0), (1, 1), (1, 1),
(0, 0), (0, 0), (1, 1), (1, 0), (1, 2), (0, 0), (2, 2), (1, 1), (0, 0), (0, 0), (1, 1), (2, 1),
(1, 2), (2, 2), (1, 1), (2, 2), (2, 1), (0, 0), (1, 1), (0, 0), (1, 2), (2, 1), (2, 2), (0, 0),
(2, 1), (2, 2), (1, 2), (2, 2), (0, 0), (0, 0), (0, 0), (1, 1), (0, 0), (0, 0), (2, 2), (2, 2),
(2, 2), (2, 2), (2, 2), (1, 2), (2, 2), (1, 2)]
```

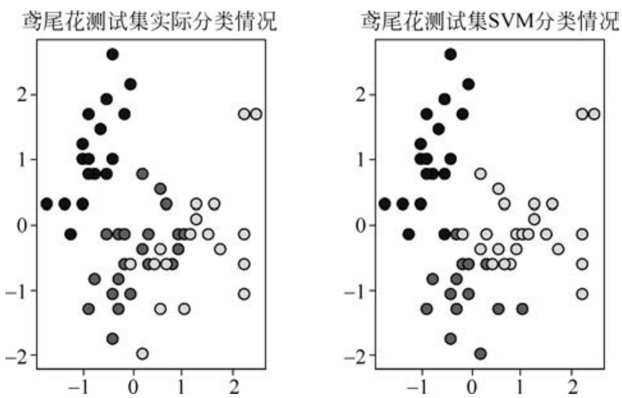


图 3.45　一对一模型鸢尾花测试集结果对比图

一对多模型的准确率如下所示：

```
SVM - 输出训练集的准确率为: 0.9444444444444444
SVM - 输出测试集的准确率为: 0.5333333333333333
```

一对多模型的鸢尾花分类结果如图 3.46 所示。

一对多模型中将测试集的实际结果与预测结果相对比，可以得到如图 3.47 所示的结果。

```
鸢尾花训练集准确率: 0.9444444444444444
鸢尾花测试集准确率: 0.5333333333333333
```

```
鸢尾花测试集实际结果与预测结果显示如下：[(0, 1), (1, 2), (1, 2), (0, 1), (2, 1), (1, 2),
(2, 1), (0, 0), (0, 0), (2, 1), (1, 1), (0, 1), (2, 2), (1, 2), (1, 1), (0, 0), (1, 1), (1, 1),
(0, 0), (0, 0), (1, 1), (1, 1), (1, 2), (0, 0), (2, 1), (1, 1), (0, 1), (0, 0), (1, 1), (2, 1),
(1, 2), (2, 2), (1, 1), (2, 2), (2, 1), (0, 0), (1, 1), (0, 0), (1, 2), (2, 1), (2, 2), (0, 0),
(2, 1), (2, 1), (1, 2), (2, 1), (0, 1), (0, 0), (0, 0), (1, 1), (0, 0), (0, 0), (2, 1), (2, 1),
(2, 2), (2, 2), (2, 1), (1, 2), (2, 1), (1, 1)]
```

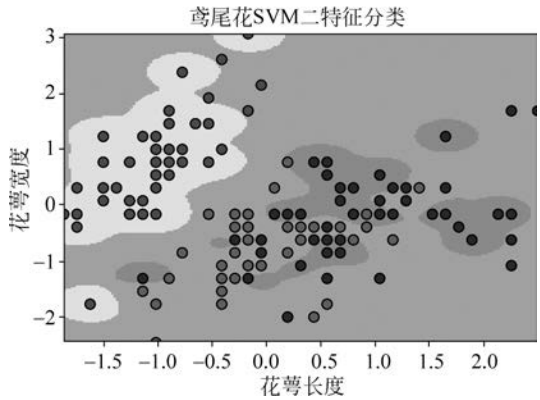


图 3.46　一对多模型的鸢尾花分类结果图

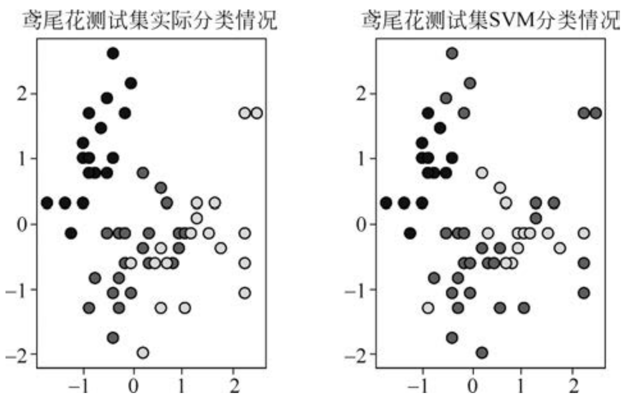


图 3.47　一对多模型测试集结果对比图

3. 自编代码实现

1）一对一自编代码

（1）导入模块。

```
# 导入模块
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib.colors import ListedColormap
from sklearn import svm, datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from numpy import array
from sklearn.utils import shuffle
```

（2）数据导入。

```
# 鸢尾花数据处理
iris = datasets.load_iris()
X = iris.data[:, :2] # 为便于绘图仅选择两个特征
y = iris.target # 类别
# 数据预处理：数据标准化
scaler = StandardScaler()
scaler.fit(X)
X = scaler.transform(X)

# 随机划分训练集与测试集,测试集占训练集的 60 % (train_size = 0.6),random_state 是随机数
# 种子
# x_train, x_test, y_train, y_test = train_test_split(X, y, random_state = 1, train_size =
0.6)
```

（3）将任意两个类别组合并打乱后划分训练集和测试集。

```
# 任意两个类别进行组合,打乱后划分训练集和测试集
x1 = [ex for ex, ey in zip(X, y) if ey in [0, 1]]
y1 = [ey for ey in y if ey in [0, 1]]
x1, y1 = shuffle(x1, y1, random_state = 1)
x1 = array(x1)
y1 = array(y1)
x1_train = x1[:80]
y1_train = y1[:80]
x1_test = x1[80:]
y1_test = y1[80:]
# 任意两个类别进行组合,打乱后划分训练集和测试集
x2 = [ex for ex, ey in zip(X, y) if ey in [0, 2]]
y2 = [ey for ey in y if ey in [0, 2]]
x2, y2 = shuffle(x2, y2, random_state = 1)
x2 = array(x2)
y2 = array(y2)
x2_train = x2[:80]
y2_train = y2[:80]
x2_test = x2[80:]
y2_test = y2[80:]
# 任意两个类别进行组合,打乱后划分训练集和测试集
x3 = [ex for ex, ey in zip(X, y) if ey in [1, 2]]
y3 = [ey for ey in y if ey in [1, 2]]
x3, y3 = shuffle(x3, y3, random_state = 1)
x3 = array(x3)
y3 = array(y3)
x3_train = x3[:80]
y3_train = y3[:80]
x3_test = x3[80:]
y3_test = y3[80:]
```

（4）调包进行模型训练。

```
#引入模型 线性 SVM
# 模型 1 训练
```

```
model1 = svm.SVC(C = 1e9, kernel = 'linear')
model1.fit(x1_train, y1_train)

# 模型 2 训练
model2 = svm.SVC(C = 1e9, kernel = 'linear')
model2.fit(x2_train, y2_train)

# 模型 3 训练
model3 = svm.SVC(C = 0.8, gamma = 0.1, kernel = 'rbf')
model3.fit(x3_train, y3_train)
```

（5）输出三个二分类模型各自的训练准确率。

```
print("SVM1 - 输出训练集的准确率为: ", model1.score(x1_train, y1_train))
print("SVM1 - 输出测试集的准确率为:", model1.score(x1_test,y1_test))

print("SVM2 - 输出训练集的准确率为: ", model2.score(x2_train, y2_train))
print("SVM2 - 输出测试集的准确率为:", model2.score(x2_test,y2_test))

print("SVM3 - 输出训练集的准确率为: ", model3.score(x3_train, y3_train))
print("SVM3 - 输出测试集的准确率为:",model3.score(x3_test,y3_test))
```

（6）绘制三个二分类模型的分类结果图。

```
# 绘制图像
# 确定坐标轴范围,x、y 轴分别表示两个特征
x1_min, x1_max = x1[:, 0].min(), x1[:, 0].max() # 第 0 列的范围为 x[:, 0],其中":"表示所有
                                                # 行,0 表示第 1 列
x2_min, x2_max = x1[:, 1].min(), x1[:, 1].max() # 第 1 列的范围为 x[:, 1],其中":"表示所有
                                                # 行,1 表示第 2 列
x1_grid, x2_grid = np.mgrid[x1_min:x1_max:200j, x2_min:x2_max:200j] # 生成网格采样点
grid_test = np.stack((x1_grid.flat, x2_grid.flat), axis = 1) # 先将网格点 x1 和 x2 展平为列
# 向量,然后按列拼接两个列向量,构建测试样本点
grid_hat = model1.predict(grid_test)       # 预测分类值
grid_hat = grid_hat.reshape(x1_grid.shape) # 使之与输入的形状相同
# 指定默认字体
mpl.rcParams['font.sans - serif'] = [u'SimHei']
mpl.rcParams['axes.unicode_minus'] = False
# 绘制
cm_light = mpl.colors.ListedColormap(['#A0FFA0', '#FFA0A0'])
cm_dark = mpl.colors.ListedColormap(['g', 'r'])

plt.pcolormesh(x1_grid, x2_grid, grid_hat, cmap = cm_light)
plt.scatter(x1[:, 0], x1[:, 1], c = y1, edgecolors = 'k', s = 50, cmap = cm_dark) # 样本
plt.scatter(x1_test[:, 0], x1_test[:, 1], s = 120, facecolors = 'none', zorder = 10) # 圈中测
                                                                                   # 试集样本
plt.xlabel(u'花萼长度', fontsize = 13)
plt.ylabel(u'花萼宽度', fontsize = 13)
plt.xlim(x1_min, x1_max)
plt.ylim(x2_min, x2_max)
plt.title(u'鸢尾花 SVM 二特征分类', fontsize = 15)
# plt.grid()
plt.show()
```

```
# 绘制图像
# 确定坐标轴范围,x、y 轴分别表示两个特征
x1_min, x1_max = x2[:, 0].min(), x21[:, 0].max() # 第 0 列的范围为 x[:, 0] ,其中":"表示所有
                                                 # 行,0 表示第 1 列
x2_min, x2_max = x2[:, 1].min(), x2[:, 1].max() # 第 1 列的范围为 x[:, 1],其中 ":"表示所有
                                                # 行,1 表示第 2 列
x1_grid, x2_grid = np.mgrid[x1_min:x1_max:200j, x2_min:x2_max:200j] # 生成网格采样点
grid_test = np.stack((x1_grid.flat, x2_grid.flat), axis = 1) # 先将网格点 x1 和 x2 展平为列
#向量,然后按列拼接两个列向量,构建测试样本点
grid_hat = model1.predict(grid_test)       # 预测分类值
grid_hat = grid_hat.reshape(x1_grid.shape) # 使之与输入的形状相同
# 指定默认字体
mpl.rcParams['font.sans - serif'] = [u'SimHei']
mpl.rcParams['axes.unicode_minus'] = False
# 绘制
cm_light = mpl.colors.ListedColormap(['#A0FFA0', '#FFA0A0'])
cm_dark = mpl.colors.ListedColormap(['g', 'r'])

plt.pcolormesh(x1_grid, x2_grid, grid_hat, cmap = cm_light)
plt.scatter(x2[:, 0], x2[:, 1], c = y2, edgecolors = 'k', s = 50, cmap = cm_dark)      # 样本
plt.scatter(x2_test[:, 0], x2_test[:, 1], s = 120, facecolors = 'none', zorder = 10) # 圈中测
# 试集样本
plt.xlabel(u'花萼长度', fontsize = 13)
plt.ylabel(u'花萼宽度', fontsize = 13)
plt.xlim(x1_min, x1_max)
plt.ylim(x2_min, x2_max)
plt.title(u'鸢尾花 SVM 二特征分类', fontsize = 15)
# plt.grid()
plt.show()

# 绘制图像
# 确定坐标轴范围,x、y 轴分别表示两个特征
x1_min, x1_max = x3[:, 0].min(), x3[:, 0].max() # 第 0 列的范围为 x[:, 0] ,其中":"表示所有
                                                # 行,0 表示第 1 列
x2_min, x2_max = x3[:, 1].min(), x3[:, 1].max() # 第 1 列的范围为 x[:, 1] ,其中":"表示所有
                                                # 行,1 表示第 2 列
x1_grid, x2_grid = np.mgrid[x1_min:x1_max:200j, x2_min:x2_max:200j] # 生成网格采样点
grid_test = np.stack((x1_grid.flat, x2_grid.flat), axis = 1)
# 先将网格点 x1 和 x2 展平为列向量,然后按列拼接两个列向量,构建测试样本点
grid_hat = model1.predict(grid_test)       # 预测分类值
grid_hat = grid_hat.reshape(x1_grid.shape) # 使之与输入的形状相同
# 指定默认字体
mpl.rcParams['font.sans - serif'] = [u'SimHei']
mpl.rcParams['axes.unicode_minus'] = False
# 绘制
cm_light = mpl.colors.ListedColormap(['#A0FFA0', '#FFA0A0'])
cm_dark = mpl.colors.ListedColormap(['g', 'r'])

plt.pcolormesh(x1_grid, x2_grid, grid_hat, cmap = cm_light)
plt.scatter(x3[:, 0], x3[:, 1], c = y3, edgecolors = 'k', s = 50, cmap = cm_dark) # 样本
plt.scatter(x3_test[:, 0], x3_test[:, 1], s = 120, facecolors = 'none', zorder = 10) # 圈中测
                                                                                     # 试集样本
```

```
plt.xlabel(u'花萼长度', fontsize = 13)
plt.ylabel(u'花萼宽度', fontsize = 13)
plt.xlim(x1_min, x1_max)
plt.ylim(x2_min, x2_max)
plt.title(u'鸢尾花 SVM 二特征分类', fontsize = 15)
# plt.grid()
plt.show()
```

(7) 将测试集合并后,放到三个模型中。

```
# 将每个二分类器的测试集合并
x_test = np.concatenate((x1_test,x2_test),axis = 0)
x_test = np.concatenate((x_test,x3_test),axis = 0)
y_test = np.concatenate((y1_test,y2_test),axis = 0)
y_test = np.concatenate((y_test,y3_test),axis = 0)

# 将合并后的测试集分别放进三个二分类器中进行测试
y1_test_hat = model1.predict(x_test)
y2_test_hat = model2.predict(x_test)
y3_test_hat = model3.predict(x_test)
```

(8) 根据三个二分类模型的预测结果通过投票产生最终三分类结果。

```
# 根据三个 SVM 二分类器结果,通过投票产生最终分类结果
y_test_hat = np.zeros(60)
for i in range(60):
  if y1_test_hat[i] == y2_test_hat[i]:
    y_test_hat[i] = y1_test_hat[i]
  elif y1_test_hat[i] == y3_test_hat[i]:
    y_test_hat[i] = y1_test_hat[i]
  elif y2_test_hat[i] == y3_test_hat[i]:
    y_test_hat[i] = y2_test_hat[i]
  else:
    y_test_hat[i] = -1
y_test_hat = y_test_hat.astype(int)
```

(9) 计算最终三分类结果的准确率。

```
# 计算一对一模型的测试集准确率
count = 0
for i in range(60):
  if y_test[i] == y_test_hat[i]:
    count += 1
ac = count / len(y_test)
print('测试集的准确率为:',ac)
```

(10) 将测试集实际三分类与预测三分类进行可视化对比。

```
# 将一对多模型测试集的分类情况进行可视化

comp = zip(y_test,y_test_hat) # 用 zip 把原始结果和预测结果放在一起,显示如下
print('鸢尾花测试集原始结果与预测结果显示如下:',list(comp))
```

```
# 还可以通过图像进行可视化
plt.figure()
plt.subplot(121)
plt.scatter(x_test[:,0],x_test[:,1],c = y_test.reshape((-1)),edgecolors = 'k',s = 50)
plt.title(u'鸢尾花测试集实际分类情况', fontsize = 13)
plt.subplot(122)
plt.scatter(x_test[:,0],x_test[:,1],c = y_test_hat.reshape((-1)),edgecolors = 'k',s = 50)
plt.title(u'鸢尾花测试集一对一分类情况', fontsize = 13)
```

2）一对一结果展示

三个二分类器的准确率如下。

```
SVM1 - 输出训练集的准确率为: 1.0
SVM1 - 输出测试集的准确率为: 1.0
SVM2 - 输出训练集的准确率为: 1.0
SVM2 - 输出测试集的准确率为: 1.0
SVM3 - 输出训练集的准确率为: 0.725
SVM3 - 输出测试集的准确率为: 0.7
```

三个二分类器的分类结果如图 3.48 所示。

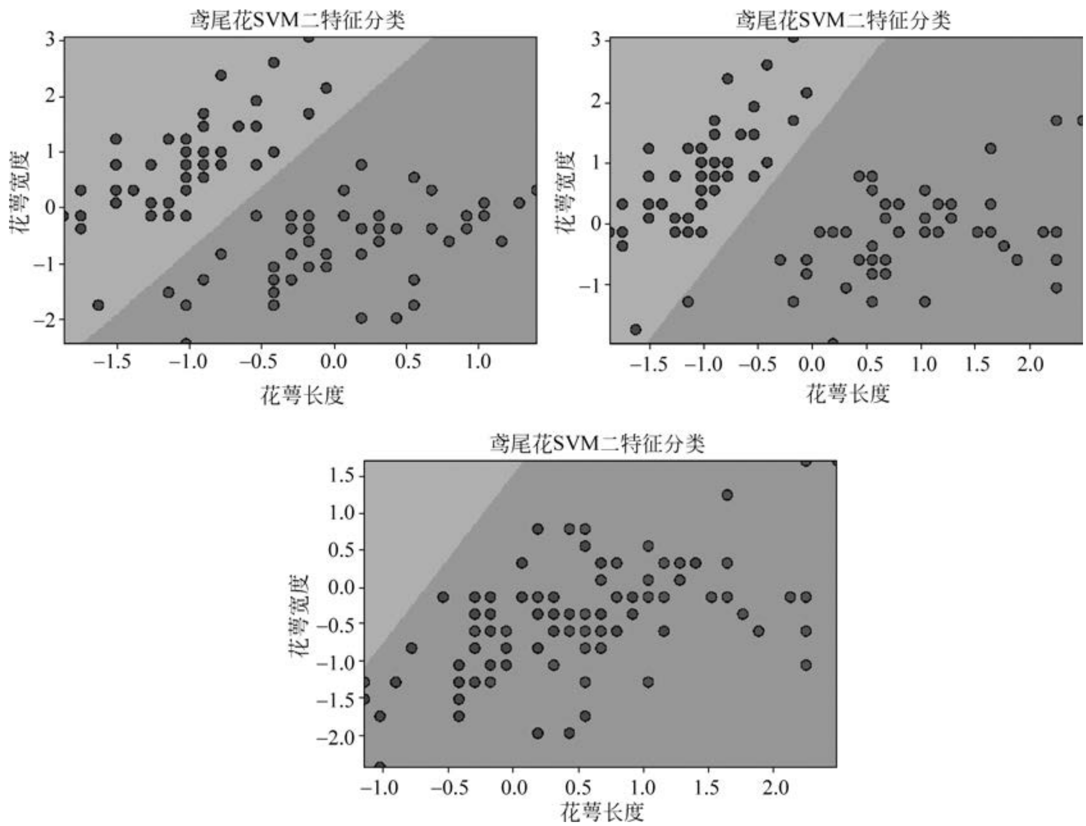


图 3.48　自编一对一模型中三个分类器结果图

将三个二分类器组合后，其对测试集的三分类准确率为：

```
测试集的准确率为: 0.8
```

将测试集实际结果与预测结果可视化对比如图 3.49 所示。

```
鸢尾花测试集实际结果与预测结果显示如下：[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (1, 2),
(0, 0), (0, 0), (1, 1), (1, 2), (0, 0), (0, 0), (1, 1), (1, 1), (0, 0), (1, 2), (0, 0), (1, 1),
(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (2, 2), (0, 0), (0, 0), (2, 1), (2, 1),
(0, 0), (0, 0), (2, 1), (2, 2), (0, 0), (2, 2), (0, 0), (2, 2), (0, 0), (0, 0), (1, 1), (1, 1),
(1, 1), (1, 1), (1, 1), (2, 2), (1, 2), (1, 2), (2, 1), (2, 1), (1, 2), (1, 1), (2, 1), (2, 2),
(1, 1), (2, 2), (1, 1), (2, 2), (1, 1), (1, 1)]
```

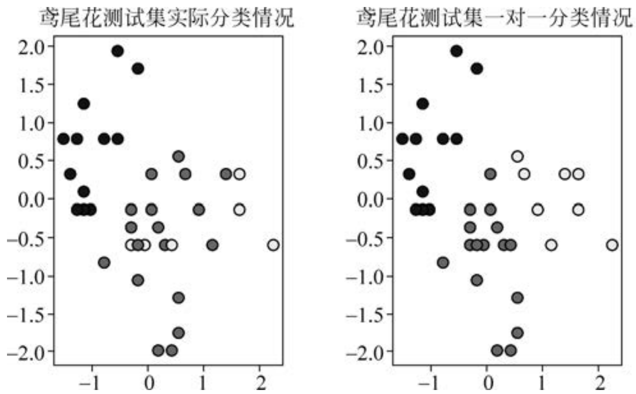


图 3.49　自编一对一模型的测试集结果对比

3）一对多自编代码

导入模块如下：

```
# 导入模块
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib.colors import ListedColormap
from sklearn import svm, datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

（1）数据处理。

```
# 鸢尾花数据处理
iris = datasets.load_iris()
# x = iris.data
# y = iris.target
# X = x[y<2,:2] # 为便于绘图仅选择两个特征
# y = y[y<2] # 为提供线性可分数据集,只选取了前两类鸢尾花数据集
X = iris.data[:, :2] # 为便于绘图仅选择两个特征
y = iris.target # 类别

# 数据预处理：数据归一化
scaler = StandardScaler()
scaler.fit(X)
x = scaler.transform(X)
```

```
# x_test, y_test = shuffle(x_test, y_test, random_state = 1)# 随机划分训练集与测试集,测试
# 集占训练集的 60%(train_size = 0.6),random_state 是随机数种子
x_train, x_test, y_train, y_test = train_test_split(X, y, random_state = 1, train_size = 0.6)

# 为了便于画图,将打乱后的特征合并在一起
x1 = np.concatenate((x_train,x_test),axis = 0)
# 将 0 类作为正例,其他两类作为反例
y1_train = np.zeros(len(y_train))
for i in range(len(y_train)):
  if y_train[i] != 0:
     y1_train[i] = -1
  else:
     y1_train[i] = y_train[i]

    y1_test = np.zeros(len(y_test))
    for i in range(len(y_test)):
  if y_test[i] != 0:
     y1_test[i] = -1
  else:
     y1_test[i] = y_test[i]

# 为了便于画图,将重新定义后的标签合并在一起
y1 = np.concatenate((y1_train, y1_test), axis = 0)
# 将 1 类作为正例,其他两类作为反例
y2_trai n = np.zeros(len(y_train))
for i in range(len(y_train)):
  if y_train[i] != 1:
     y2_train[i] = -1
  else:
     y2_train[i] = y_train[i]

    y2_test = np.zeros(len(y_test))
    for i in range(len(y_test)):
  if y_test[i] != 1:
     y2_test[i] = -1
  else:
     y2_test[i] = y_test[i]

# 为了便于画图,将重新定义后的标签合并在一起
y2 = np.concatenate((y2_train, y2_test), axis = 0)
# 将 2 类作为正例,其他两类作为反例
y3_train = np.zeros(len(y_train))
for i in range(len(y_train)):
  if y_train[i] != 2:
     y3_train[i] = -1
  else:
     y3_train[i] = y_train[i]

y3_test = np.zeros(len(y_test))
for i in range(len(y_test)):
  if y_test[i] != 2:
     y3_test[i] = -1
```

```
    else:
        y3_test[i] = y_test[i]

# 为了便于画图,将重新定义后的标签合并在一起
y3 = np.concatenate((y3_train, y3_test), axis = 0)
```

(2) 调包进行模型训练。

```
#引入模型 线性 SVM
# 模型 1 训练
model1 = svm.SVC(C = 1e9, kernel = 'linear')
model1.fit(x_train, y1_train)

# 模型 2 训练
# model2 = svm.SVC(C = 0.01, gamma = 0.5, kernel = 'rbf')
model2 = svm.SVC(C = 1e9, kernel = 'linear')
model2.fit(x_train, y2_train)

# 模型 3 训练
model3 = svm.SVC(C = 0.8, gamma = 0.1, kernel = 'rbf')
model3.fit(x_train, y3_train)
```

(3) 输出三个二分类模型的准确率。

```
print("SVM1 - 输出训练集的准确率为: ", model1.score(x_train, y1_train))
print("SVM1 - 输出测试集的准确率为:", model1.score(x_test,y1_test))

print("SVM2 - 输出训练集的准确率为: ", model2.score(x_train, y2_train))
print("SVM2 - 输出测试集的准确率为:", model2.score(x_test,y2_test))

print("SVM3 - 输出训练集的准确率为: ", model3.score(x_train, y3_train))
print("SVM3 - 输出测试集的准确率为:",model3.score(x_test,y3_test))
```

(4) 绘制三个二分类模型的分类结果图。

```
# 绘制图像
# 确定坐标轴范围,x、y 轴分别表示两个特征
x1_min, x1_max = x1[:, 0].min(), x1[:, 0].max()
# 第 0 列的范围为 x[:, 0],其中":"表示所有行,0 表示第 1 列
x2_min, x2_max = x1[:, 1].min(), x1[:, 1].max()
# 第 1 列的范围为 x[:, 1],其中":"表示所有行,1 表示第 2 列
x1_grid, x2_grid = np.mgrid[x1_min:x1_max:200j, x2_min:x2_max:200j]
# 生成网格采样点
grid_test = np.stack((x1_grid.flat, x2_grid.flat), axis = 1)
# 先将网格点 x1 和 x2 展平为列向量,然后按列拼接两个列向量,构建测试样本点
grid_hat = model1.predict(grid_test)       # 预测分类值
grid_hat = grid_hat.reshape(x1_grid.shape) # 使之与输入的形状相同
# 指定默认字体
mpl.rcParams['font.sans - serif'] = [u'SimHei']
mpl.rcParams['axes.unicode_minus'] = False
# 绘制
cm_light = mpl.colors.ListedColormap(['#A0FFA0', '#FFA0A0'])
cm_dark = mpl.colors.ListedColormap(['g', 'r'])
```

```
plt.pcolormesh(x1_grid, x2_grid, grid_hat, cmap = cm_light)
plt.scatter(x1[:, 0], x1[:, 1], c = y1, edgecolors = 'k', s = 50, cmap = cm_dark)   # 样本
plt.scatter(x_test[:, 0], x_test[:, 1], s = 120, facecolors = 'none', zorder = 10) # 圈中测试
# 集样本
plt.xlabel(u'花萼长度', fontsize = 13)
plt.ylabel(u'花萼宽度', fontsize = 13)
plt.xlim(x1_min, x1_max)
plt.ylim(x2_min, x2_max)
plt.title(u'鸢尾花 SVM 二特征分类', fontsize = 15)
# plt.grid()
plt.show()

# 绘制图像
# 确定坐标轴范围,x、y 轴分别表示两个特征
x1_min, x1_max = x1[:, 0].min(), x1[:, 0].max()
# 第 0 列的范围为 x[:, 0],其中":"表示所有行,0 表示第 1 列
x2_min, x2_max = x1[:, 1].min(), x1[:, 1].max()
# 第 1 列的范围为 x[:, 0],其中":"表示所有行,1 表示第 2 列
x1_grid, x2_grid = np.mgrid[x1_min:x1_max:200j, x2_min:x2_max:200j]
# 生成网格采样点
grid_test = np.stack((x1_grid.flat, x2_grid.flat), axis = 1) # 先将网格点 x1 和 x2 展平为列
# 向量,然后按列拼接两个列向量,构建测试样本点
grid_hat = model2.predict(grid_test)                        # 预测分类值
grid_hat = grid_hat.reshape(x1_grid.shape)                  # 使之与输入的形状相同
# 指定默认字体
mpl.rcParams['font.sans - serif'] = [u'SimHei']
mpl.rcParams['axes.unicode_minus'] = False
# 绘制
cm_light = mpl.colors.ListedColormap(['#A0FFA0', '#FFA0A0'])
cm_dark = mpl.colors.ListedColormap(['g', 'r'])

plt.pcolormesh(x1_grid, x2_grid, grid_hat, cmap = cm_light)
plt.scatter(x1[:, 0], x1[:, 1], c = y2, edgecolors = 'k', s = 50, cmap = cm_dark)   # 样本
plt.scatter(x_test[:, 0], x_test[:, 1], s = 120, facecolors = 'none', zorder = 10) # 圈中测试
# 集样本
plt.xlabel(u'花萼长度', fontsize = 13)
plt.ylabel(u'花萼宽度', fontsize = 13)
plt.xlim(x1_min, x1_max)
plt.ylim(x2_min, x2_max)
plt.title(u'鸢尾花 SVM 二特征分类', fontsize = 15)
# plt.grid()
plt.show()

# 绘制图像
# 确定坐标轴范围,x、y 轴分别表示两个特征
x1_min, x1_max = x1[:, 0].min(), x1[:, 0].max()
# 第 0 列的范围为 x[:, 0],其中":"表示所有行,0 表示第 1 列
x2_min, x2_max = x1[:, 1].min(), x1[:, 1].max()
# 第 1 列的范围为 x[:, 1],其中":"表示所有行,1 表示第 2 列
x1_grid, x2_grid = np.mgrid[x1_min:x1_max:200j, x2_min:x2_max:200j]
# 生成网格采样点
grid_test = np.stack((x1_grid.flat, x2_grid.flat), axis = 1)
# 先将网格点 x1 和 x2 展平为列向量,然后按列拼接两个列向量,构建测试样本点
grid_hat = model3.predict(grid_test)       # 预测分类值
```

```
grid_hat = grid_hat.reshape(x1_grid.shape) # 使之与输入的形状相同
# 指定默认字体
mpl.rcParams['font.sans-serif'] = [u'SimHei']
mpl.rcParams['axes.unicode_minus'] = False
# 绘制
cm_light = mpl.colors.ListedColormap(['#A0FFA0', '#FFA0A0'])
cm_dark = mpl.colors.ListedColormap(['g', 'r'])

plt.pcolormesh(x1_grid, x2_grid, grid_hat, cmap=cm_light)
plt.scatter(x1[:, 0], x1[:, 1], c=y3, edgecolors='k', s=50, cmap=cm_dark) # 样本
plt.scatter(x_test[:, 0], x_test[:, 1], s=120, facecolors='none', zorder=10) # 圈中测试
                                                                              # 集样本
plt.xlabel(u'花萼长度', fontsize=13)
plt.ylabel(u'花萼宽度', fontsize=13)
plt.xlim(x1_min, x1_max)
plt.ylim(x2_min, x2_max)
plt.title(u'鸢尾花 SVM 二特征分类', fontsize=15)
# plt.grid()
plt.show()
```

（5）将测试集放进三个二分类器中进行测试。

```
# 将合并后的测试集分别放进三个二分类器中进行测试
y1_test_hat = model1.predict(x_test)
y2_test_hat = model2.predict(x_test)
y3_test_hat = model3.predict(x_test)
```

（6）根据三个二分类器的预测结果，得到最终分类结果。

```
y_test_hat = np.ones(60)
# 根据三个 SVM 二分类器结果,产生最终分类结果
for i in range(60):
  if y1_test_hat[i] != -1:
     y_test_hat[i] = y1_test_hat[i]
  elif y2_test_hat[i] != -1:
     y_test_hat[i] = y2_test_hat[i]
  elif y3_test_hat[i] != -1:
     y_test_hat[i] = y3_test_hat[i]
  else:
     y_test_hat[i] = -1
y_test_hat = y_test_hat.astype(int)
```

（7）输出最终三分类结果的准确率。

```
# 计算一对多模型的测试集准确率
count = 0
for i in range(60):
  if y_test[i] == y_test_hat[i]:
     count += 1
    ac = count / len(y_test)
print('测试集的准确率为:',ac)
```

（8）将最终测试集三分类结果与实际三分类情况进行可视化对比。

```
# 将一对多模型测试集的分类情况进行可视化
comp = zip(y_test,y_test_hat) # 用 zip 把实际结果和预测结果放在一起,显示如下
print('鸢尾花测试集实际结果与预测结果显示如下:',list(comp))
# 还可以通过图像进行可视化
plt.figure()
plt.subplot(121)
plt.scatter(x_test[:,0],x_test[:,1],c = y_test.reshape((-1)),edgecolors = 'k',s = 50)
plt.title(u'鸢尾花测试集实际分类情况', fontsize = 13)
plt.subplot(122)
plt.scatter(x_test[:,0],x_test[:,1],c = y_test_hat.reshape((-1)),edgecolors = 'k',s = 50)
plt.title(u'鸢尾花测试集一对多分类情况', fontsize = 13)
```

4）一对多运行结果

三个二分类器的准确率如下。

```
SVM1 - 输出训练集的准确率为: 1.0
SVM1 - 输出测试集的准确率为: 1.0
SVM2 - 输出训练集的准确率为: 0.7444444444444445
SVM2 - 输出测试集的准确率为: 0.6166666666666667
SVM3 - 输出训练集的准确率为: 0.8
SVM3 - 输出测试集的准确率为: 0.8166666666666667
```

三个二分类器的分类结果如图 3.50 所示。

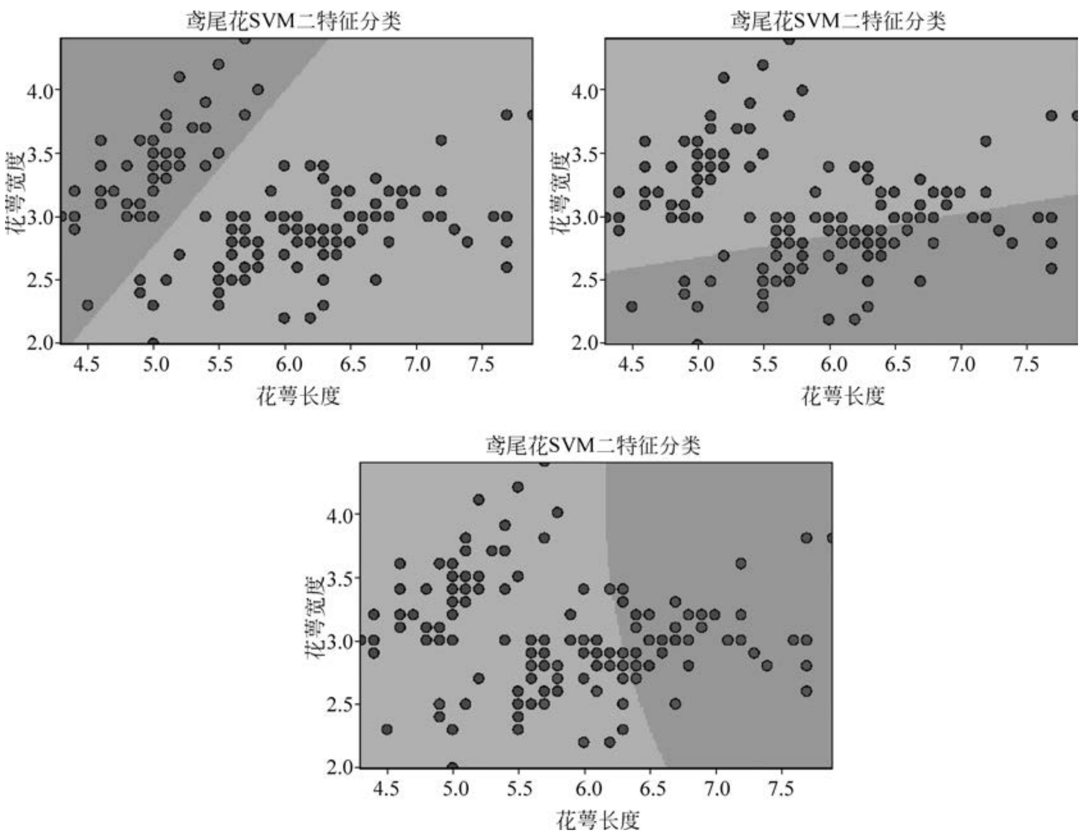


图 3.50　自编一对多模型中三个分类器结果图

将三个二分类器组合后，其对测试集三分类准确率如下：

```
测试集的准确率为: 0.6166666666666667
```

测试集实际结果与预测结果显示如下，可视化对比如图 3.51 所示。

```
鸢尾花测试集实际结果与预测结果显示如下: [(0, 0), (1, 1), (1, 2), (0, 0), (2, 2), (1, 2),
(2, 2), (0, 0), (0, 0), (2, 2), (1, 1), (0, 0), (2, 1), (1, 2), (1, -1), (0, 0), (1, 1), (1, -
1), (0, 0), (0, 0), (1, 1), (1, -1), (1, 2), (0, 0), (2, 2), (1, 1), (0, 0), (0, 0), (1, 1),
(2, 1), (1, -1), (2, 1), (1, 1), (2, 1), (2, 1), (0, 0), (1, 1), (0, 0), (1, 1), (2, 1), (2,
2), (0, 0), (2, 1), (2, 1), (1, 1), (2, 1), (0, 0), (0, 0), (0, 0), (1, 1), (0, 0), (0, 0), (2,
2), (2, 2), (2, 1), (2, 1), (2, 1), (1, -1), (2, 1), (1, -1)]
```

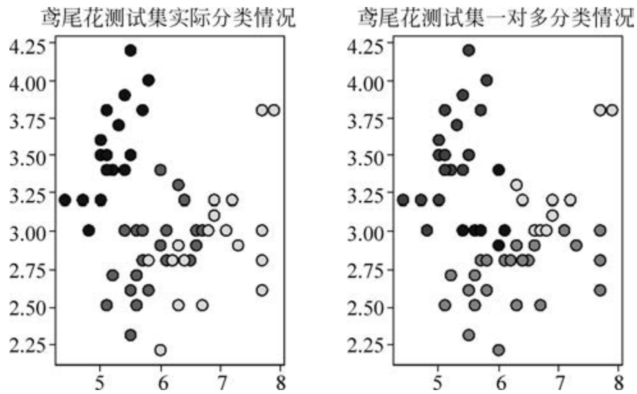


图 3.51　自编一对多模型测试集结果对比图

3.8.4　实验

1. 实验目的

(1) 理解机器学习的基本理论，训练运用机器学习的思想对软件问题进行分析、设计、实践的基本技术，掌握科学的实验方法。

(2) 培养学生观察问题、分析问题和独立解决问题的能力。

(3) 通过实验使学生能够正确使用开发环境，掌握编译、编辑、连接、调试程序的基本能力。

(4) 通过综合性、设计性实验训练，使学生初步掌握一般应用软件的设计方法；培养良好的编程习惯，注意源码管理、程序可读性、规范化等问题。

(5) 培养正确记录实验数据和现象，正确处理实验数据和分析实验结果及调试程序的能力，以及正确书写实验报告的能力。

2. 实验数据

手写数字 MNIST 数据集。

3. 实验要求

利用手写数字 MNIST 数据集，任选其中四类进行多类分类，要求分别使用一对一和一对多进行实现。

第4章

回归问题

4.1 线性回归算法

4.1.1 原理简介

1. 线性回归模型

回归模型建立了输入 $\boldsymbol{X}$(自变量)与输出 $\boldsymbol{Y}$(因变量)之间的关系,即 $\boldsymbol{Y}=f(\boldsymbol{X})$。通常 $\boldsymbol{Y}$ 是连续型变量,否则这个模型应该是分类模型而不是回归模型。这里的关系式 $f(\boldsymbol{X})$既可以是显性的函数关系式,如高斯函数、多项式函数;也可以是隐性的对应关系,如神经网络结构。若 $f(\boldsymbol{X})$是线性函数,则称这类回归模型为线性回归模型。即

$$\boldsymbol{Y}=\theta_0+\theta_1 X_1+\cdots+\theta_p X_p \tag{4.1}$$

$\boldsymbol{X}=(X_1,\cdots,X_p)$是 p 维向量,特别地,若 $p=1$,则称其为一元回归模型;否则称其为多元回归模型。其中 $\theta_0,\theta_1,\cdots,\theta_p$ 为未知模型参数。建立线性回归模型的本质就是通过已知数据学习出这些模型参数,因此线性回归模型是参数学习模型。

线性回归模型在现实中有广泛的应用,一方面在于其思想简单,易于实现,同时具有强大的可解释性,既可以用线性回归模型进行预测,依据式(4.1),输入 $\boldsymbol{X}$ 的值可以获得预测的 $\boldsymbol{Y}$ 值,还能根据式(4.1)右边各个系数的大小得到各特征对因变量影响程度的重要性,用于控制分析;另一方面,线性回归模型往往是其他非线性回归模型的基础,如多项式回归、非线性回归模型就可以通过数据预处理转换为线性模型,而不加激活函数的神经网络模型,其本质上也是线性模型。从而线性模型的诸多基本思想都可以推广到很多机器学习模型中去。

2. 线性回归模型的参数估计

已知带标签的训练数据集为 $\boldsymbol{D}=\{(x_{i1},x_{i2},\cdots,x_{ip},y_i)\mid i=1,2,\cdots,n\}$,建立线性回归模型的本质是学习式(4.1)中的参数$\boldsymbol{\theta}=[\theta_0,\theta_1,\cdots,\theta_p]$,从而建立线性回归方程。下面以身高-体重数据为例,展示这个学习过程中的相关问题。

分析一组人群的身高 $\boldsymbol{x}$(cm)和体重 $\boldsymbol{y}$(kg)之间的对应关系。数据如下:

$$\boldsymbol{x}=[171,175,159,155,152,158,154,164,168,166,159,165]$$