第3章

sklearn机器学习分类器

在Python环境下，sklearn（全称是scikit-learn）是目前最好用的机器学习函数库。sklearn库中几乎集成了所有经典的机器学习算法，同时配以非常简单的实现语句（通常为一两行代码）及模式化的调参过程，使得用户可以花费更多时间在特征工程及数据解决上，并且使得建模的过程集中于算法比较及模型选择上。

当然，sklearn也是有弊端的，如过于简单的语句使得其功能相对固定，这就使构建定制化的模型变得相对困难。

本章将介绍sklearn机器学习的常用分类器。

3.1 分类器的选择

在机器学习中，分类器的作用是在标记好类别的训练数据基础上判断一个新的观察样本所属的类别。分类器依据学习的方式可以分为无监督学习和监督学习。

本节的目的是分类器的选择。可以依据下面四个要点来选择合适的分类器。

1. 泛化能力和拟合之间的权衡

分类器在训练样本上的性能是过拟合评估，如果一个分类器在训练样本上的正确率很高，这说明分类器能够很好地拟合训练数据。但一个好的拟合训练数据的分类器会存在很大的偏置，所以在测试数据上不一定能够得到好的效果。如果一个分类器在训练数据上能够得到很好的效果，但在测试数据上效果下降严重，这说明分类器过拟合了训练数据。从另一个方面分析，如果分类器在测试数据上能够取得好的效果，那么就说明分类器的泛化能力强。分类器的泛化和拟合是一个此消彼长的过程，泛化能力强的分类器拟合能力一般很弱，反之则相反。所以分类器需要在泛化能力和拟合能力间取得平衡。

2. 分类函数的复杂度和训练数据的大小

分类器对于训练数据大小的选择也是至关重要的，如果是一个简单的分类问题，那么拟合能力强、泛化能力弱的分类器就可以通过很小的一部分训练数据来得到。反之，如果是一个复杂的分类问题，那么分类器学习就需要大量的训练数据和泛化能力强的学习算法。一个好的分类器应该能够根据问题的复杂度和训练数据的大小自动地调整拟合能力和泛化能力之间的平衡。

3. 输入的特征空间的维数

如果输入特征空间的向量维数很高，就会造成分类问题变得复杂，即使最后的分类函数只

需几个特征来决定。这是因为过高的特征维数会混淆学习算法并会导致分类器的泛化能力过强，而泛化能力过强会使得分类器变化太大，性能下降。因此，一般高维特征向量输入的分类器都需要调节参数，使其泛化能力变弱，拟合能力变强。

4. 输入的特征向量之间的均一性和相互之间的关系

如果特征向量包含多种类型的数据(如离散、连续)，那么如 SVM、线性回归、逻辑回归等多分类就不适用。这些分类器要求输入的特征必须是数字而且要归一化到相似的范围内。但是决策树分类器却能够很好地处理这些不归一的数据。如果有多个输入特征向量，每个特征向量之间相互独立，即当前特征向量的分类器输出仅仅和当前的特征向量输入有关，那么最优的分类器即是基于线性函数和距离函数的分类器，如线性回归、SVM、朴素贝叶斯等。反之，如果特征向量之间存在复杂的相互关系，那么决策树和神经网络更加适合于这类问题。

3.2 训练感知器

了解 sklearn 库的第一步就是训练感知器，下面通过实例来演示，具体实现步骤如下。

(1) 把 150 个鸢尾花样本的花瓣长度和花瓣宽度存入特征矩阵 $\boldsymbol{X}$，把相应的品种分类标签存入向量 $\boldsymbol{y}$：

```
from sklearn import datasets
import numpy as np

iris = datasets.load_iris()
X = iris.data[:, [2, 3]]
y = iris.target
print('Class labels:', np.unique(y))
Class labels: [0 1 2]
```

(2) 为了评估经过训练的模型对未知数据处理的效果，再进一步将数据集分裂成单独的训练集和测试集：

```
from sklearn.model_selection import train_test_split

#train_test_split 函数把 X 和 y 随机分为 30 % 的测试数据和 70 % 的训练数据
#在分割前已经在内部训练,random_state 为固定的随机数种子,确保结果可
#重复,通过定义 stratify = y 获得内置的分层支持,将各子数据集中不同分类标签
#的数据比例设置为总数据集比例
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size = 0.3, random_state = 1, stratify = y)
```

(3) 调用 NumPy 的 bincount 函数来对阵列中的每个值进行统计，以验证数据。

```
#计算每种标签的样本数量有多少,分别是 50 个
print('y 的标签计数:', np.bincount(y))
#计算训练集中每种标签样本数量有多少,分别是 35 个
print('y_train 的标签计数:', np.bincount(y_train))
#计算测试集中每种标签样本数量有多少,分别是 15 个
print('y_test 的标签计数:', np.bincount(y_test))
y 的标签计数: [50 50 50]
y_train 的标签计数: [35 35 35]
y_test 的标签计数: [15 15 15]
```

(4) 调用 sklearn 库中预处理模块 preprocessing 中的类 StanderScaler 来对特征进行标准化：

```
#对特征进行标准化
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
```

```
sc.fit(X_train)
X_train_std = sc.transform(X_train)
X_test_std = sc.transform(X_test)
```

在以上代码中，调用 StanderScaler 的 fit 方法对训练数据的每个特征维度参数 μ 和 σ 进行估算。调用 transform 方法，利用估计的参数 μ 和 σ 对训练数据进行标准化。

注意：在标准化测试集时，要注意使用相同的特征调整参数以确保训练集与测试集的数值具有可比性。

(5) 训练感知器模型。

```
from sklearn.linear_model import Perceptron

ppn = Perceptron(max_iter = 40, eta0 = 0.1, random_state = 1)
ppn.fit(X_train_std, y_train)
Perceptron(alpha = 0.0001, class_weight = None, eta0 = 0.1, fit_intercept = True,
    max_iter = 40, n_iter = None, n_jobs = 1, penalty = None, random_state = 1,
    shuffle = True, tol = None, verbose = 0, warm_start = False)
```

(6) 调用 predict 方法做预测。

```
y_pred = ppn.predict(X_test_std)
print('错误分类的样本: %d' % (y_test != y_pred).sum())
```

此处结果为“错误分类的样本:3”，当然也有其他的性能指标，如分类准确度：

```
#计算分类准确度
from sklearn.metrics import accuracy_score
print('准确性: %.2f' % accuracy_score(y_test, y_pred))
```

此处结果为“准确性：0.93”。

(7) 利用 plot_decision_regions 函数绘制新训练感知器的模型决策区，并以可视化的方式展示区分不同花朵样本的效果，可以通过圆圈来突出显示来自测试集的样本：

```
from matplotlib.colors import ListedColormap
from matplotlib import pyplot as plt
plt.rcParams['font.sans-serif'] = ['SimHei']          #显示中文
plt.rcParams['axes.unicode_minus'] = False            #显示负号

def plot_decision_regions(X, y, classifier, test_idx = None, resolution = 0.02):
    #setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])
    #绘制决策面
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha = 0.3, cmap = cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x = X[y == cl, 0],
                    y = X[y == cl, 1],
                    alpha = 0.8,
                    c = colors[idx],
```

```
                        marker = markers[idx],
                        label = cl,
                        edgecolor = 'black')
    #突出显示测试样本
    if test_idx:
        #绘制所有样本
        X_test, y_test = X[test_idx, :], y[test_idx]
        plt.scatter(X_test[:, 0],
                        X_test[:, 1],
                        #将测试集数据显示为粉色标记
                        c = 'pink',
                        edgecolor = 'black',
                        alpha = 1.0,
                        linewidth = 1,
                        marker = 'o',
                        s = 100,
                        label = '测试集')
X_combined_std = np.vstack((X_train_std, X_test_std))
y_combined = np.hstack((y_train, y_test))

plot_decision_regions(X = X_combined_std, y = y_combined,
                        classifier = ppn, test_idx = range(105, 150))
plt.xlabel('花瓣长度 [标准化]')
plt.ylabel('花瓣宽度 [标准化]')
plt.legend(loc = '左上角')
plt.tight_layout()
plt.show()
```

运行程序,效果如图 3-1 所示。

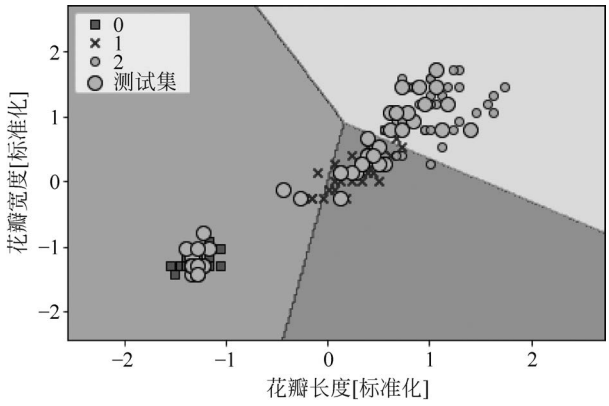


图 3-1 绘制新训练感知器的模型决策区

如图 3-1 中所看到的,三种花不能被线性决策边界完全分离,所以实践中通常不推荐使用感知器算法。

3.3 基于逻辑回归的分类概率建模

逻辑回归一般用于估计某种事物的可能性("可能性"而非数学上的"概率"),不可以直接当作概率值来用。逻辑回归可以用于预测系统或产品的故障的可能性,还可用于市场营销应用程序,例如预测客户购买产品或中止订购的倾向等。在经济学中它可以用来预测一个人选择进入劳动力市场的可能性,而商业应用则可以用来预测房主拖欠抵押贷款的可能性。还可以根据逻辑回归模型,预测在不同的自变量情况下,发生某种情况的概率。

3.3.1　几个相关定义

逻辑回归是一种分类模型而非回归模型，在介绍逻辑回归前先来了解几个相关定义。

(1) 让步比：$\frac{p}{(1-p)}$，代表阳性事件的概率，它指的是要预测的事件的可能性，例如：病人有某种疾病的可能性、某人买彩票中了的可能性等。

(2) 让步比的对数形式：$\text{logit}(p)=\text{lb}\frac{p}{(1-p)}$。logit 函数输入值的取值范围在 0 到 1 之间，转换或计算的结果值为整个实数范围，可以用它来表示特征值和对数概率之间的线性关系：

$$\text{logit}(p(y=1\mid x))=w_0x_0+w_1x_1+\cdots+w_mx_m=\sum_{i=0}^{m}w_ix_i=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}$$

此处，$p(y=1|x)$是某个特定样本属于 x 类给定特征标签为 1 的条件概率。

(3) sigmoid 函数：$\Phi(z)=\frac{1}{1+\mathrm{e}^{-z}}$，它是 logit 函数的逆形式。sigmoid 函数的形状如图 3-2 所示。

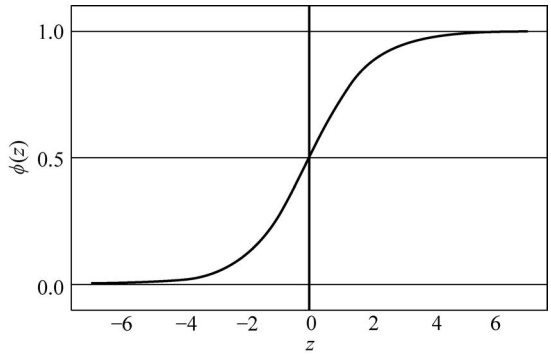


图 3-2　sigmoid 函数的形状

3.3.2　逻辑代价函数的权重

在建立逻辑回归模型时，想要最大化 L 的可能性，先要假设数据集中的样本都是相互独立的个体。公式如下：

$$L(\boldsymbol{w})=p(\boldsymbol{y}\mid\boldsymbol{x};\boldsymbol{w})=\prod_{i}^{n}p(y^{(i)}\mid x^{(i)};\boldsymbol{w})=\prod_{i}^{n}(\phi(z^{(i)}))^{y^{(i)}}(1-\phi(z^{(i)}))^{1-y^{(i)}}$$

在实践中，最大化该方程的自然对数，也被称为对数似然函数：

$$l(\boldsymbol{w})=\log(L(\boldsymbol{w}))=\sum_{i=1}^{n}(y^{(i)}\log(\phi(z^{(i)}))+(1-y^{(i)})\log(1-\phi(z^{(i)})))$$

用梯度下降方法最小化代价函数 J：

$$J(\boldsymbol{w})=\sum_{i=1}^{n}(-y^{(i)}\log(\phi(z^{(i)}))-(1-y^{(i)})\log(1-\phi(z^{(i)})))$$

为更好地理解这个代价函数，计算一个样本训练实例的代价如下：

$$J(\phi(z),y;w)=-y\log(\phi(z))-(1-y)\log(1-\phi(z))$$

从方程中可以看到，如果 $y=0$，第一项为 0，如果 $y=1$，第二项为 0：

$$J(\phi(z),y;w)=\begin{cases}-\log(\phi(z)), & y=1\\ -\log(1-\phi(z)), & y=0\end{cases}$$

下面代码实现绘制一张图，用于说明 $\phi(z)$不同样本实例分类的代价：

```
import numpy as np
from matplotlib import pyplot as plt

def cost_1(z):
    return - np.log(sigmoid(z))

def cost_0(z):
    return - np.log(1 - sigmoid(z))
z = np.arange( - 10, 10, 0.1)
phi_z = sigmoid(z)
c1 = [cost_1(x) for x in z]
plt.plot(phi_z, c1, label = 'J(w) if y = 1')
c0 = [cost_0(x) for x in z]
plt.plot(phi_z, c0, linestyle = '-- ', label = 'J(w) if y = 0')
plt.ylim(0.0, 5.1)
plt.xlim([0, 1])
plt.xlabel('$\phi$(z)')
plt.ylabel('J(w)')
plt.legend(loc = 'best')
plt.tight_layout()
plt.show()
```

运行程序，效果如图 3-3 所示。

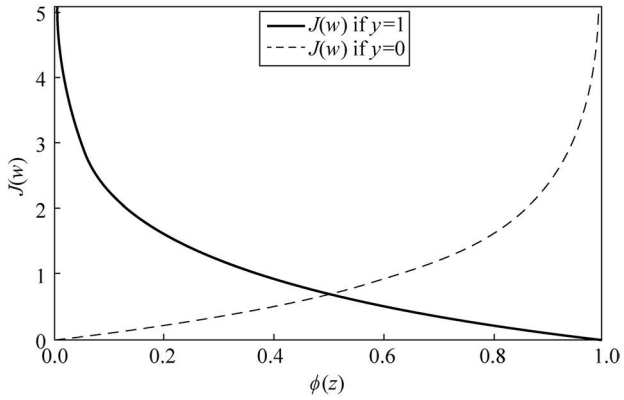


图 3-3　不同样本实例分类的代价

由结果可得出结论：如果分类为 1，则概率越小表示分类错误程度越高；如果分类为 0，则概率越大表示分类错误程度越高。

【例 3-1】　用 sklearn 训练逻辑回归模型。

```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(C = 100.0, random_state = 1)
lr.fit(X_train_std, y_train)
plot_decision_regions(X_combined_std, y_combined,
                      classifier = lr, test_idx = range(105, 150))
plt.xlabel('花瓣长度[标准化]')
plt.ylabel('花瓣宽度 [标准化]')
plt.legend(loc = '左上角')
plt.tight_layout()
plt.show()
```

运行程序，效果如图 3-4 所示。

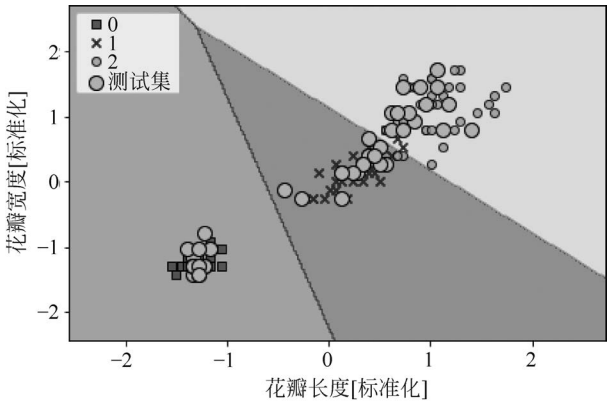
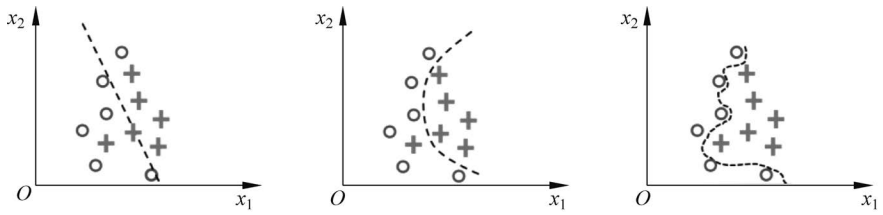


图 3-4　sklearn 训练逻辑回归模型效果

3.3.3　正则化解决过拟合问题

什么是过拟合？什么是欠拟合？过拟合是指模型在训练数据上表现良好，但无法概括未见过的新数据或测试数据；欠拟合是指模型不足以捕捉训练数据中的复杂模式，因此对未见过的数据表现不良。图 3-5 可以很好地阐明过拟合与欠拟合的情况。



(a) 未拟合数据　(b) 过拟合数据　(c) 欠拟合数据

图 3-5　过拟合与欠拟合

正则化又是什么？正则化是处理共线性(特征之间的高相关性)，消除数据中的噪声，并最终能避免过拟合的非常有效的方法。正则化的逻辑是引入额外偏置来惩罚极端的权重。

最常见的正则化是 L2 正则化，具体如下：

$$\frac{\lambda}{2}\|\boldsymbol{w}\|^2=\frac{\lambda}{2}\sum_{j=1}^{m}w_j^2$$

其中，λ 为正则化参数。

逻辑回归的代价函数可以通过增加一个简单的正则项来调整，这将在模型训练的过程中缩小权重：

$$J(\boldsymbol{w})=\sum_{i=1}^{n}(-y^{(i)}\log(\phi(z^{(i)}))-(1-y^{(i)})\log(1-\phi(z^{(i)})))+\frac{\lambda}{2}\|\boldsymbol{w}\|^2$$

可通过绘制两个权重系数的 L2 正则化路径实现可视化，代码如下：

```
weights, params = [], []
for c in np.arange(-5, 5):
    lr = LogisticRegression(C=10.**c, random_state=1)
    lr.fit(X_train_std, y_train)
    weights.append(lr.coef_[1])
    params.append(10.**c)

weights = np.array(weights)
plt.plot(params, weights[:, 0],
```

```
            label = '花瓣长度')
plt.plot(params, weights[:, 1], linestyle = '--',
            label = '花瓣宽度')
plt.ylabel('权重系数')
plt.xlabel('C')
plt.legend(loc = '左上角')
plt.xscale('log')
plt.show()
```

运行程序,效果如图 3-6 所示。

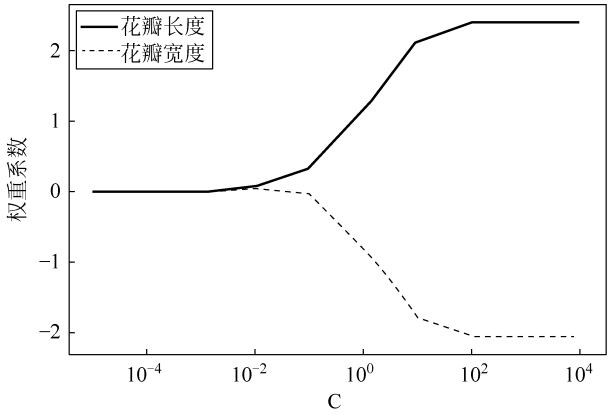


图 3-6　两个权重系数的 L2 正则化

如图 3-6 所示,减小逆正则化参数 C 可以增大正则化的强度,权重系数会变小。

3.4　支持向量机最大化分类间隔

支持向量机是一种二分类模型,它是通过在特征空间中建立间隔最大的分类器,这与感知机模型不同。

3.4.1　超平面

对二分类的逻辑回归,假设特征数为 2,那么训练模型的过程通过梯度下降不断更新参数逼近全局最优解,拟合出一条直线作为决策边界,使得这个决策边界划分出来的分类结果误差最低。

当特征数量超过 2 时,这时用来分割不同类别的“线”就成为一个面,简称超平面(hyperplane),“超”即是多维的意思(二维就是一条线,三维就是一个面,多维就是超平面)。划分超平面可用如下线性方程表示:

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

其中,$\boldsymbol{w}=(w_1,w_2,\cdots,w_d)^{\mathrm{T}}$ 为法向量,b 为位移。

如果要用一条直线将图 3-7 的两种类别(“+”和“−”)分开,可看到可分离的直线有多条。

直观上应该选红色线,因为它是“最能”分开这两种类别的。如果选择黑色线,那么可能存在一些点刚好越过黑色的线,导致被错误分类,但是红色线的容错率会更好,也就不容易出错。

如图 3-8 这种情况,如果选择绿色线,新来一个需要预测的样本(蓝色的点)本来属于“+”,但是却被分到“−”这一类,但是红色线就不会,即红色线所产生的分类结果是最健壮的,对未见示例的泛化能力最强。

红色线的这条决策边界就是通过间隔最大化求得的,并且是唯一的。在了解间隔最大化之前先了解一下函数间隔和几何间隔的概念。

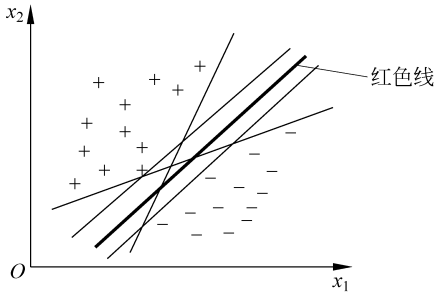


图 3-7　两种类别分离效果

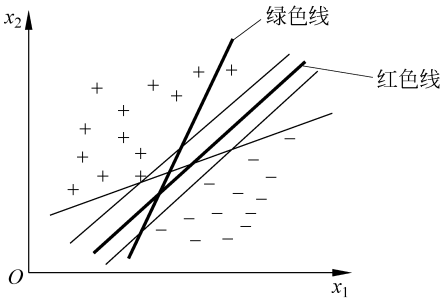


图 3-8　分类效果

3.4.2　函数间隔和几何间隔

一般来说，一个点距离超平面的远近可以表示分类预测的确信程度。

(1) 函数间隔：对于给定训练集和超平面$(\boldsymbol{w},b)$，定义超平面$(\boldsymbol{w},b)$关于样本点(x_i,y_i)的函数间隔为

$$\hat{r}_i = y_i(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i + b)$$

定义超平面$(\boldsymbol{w},b)$关于训练集的函数间隔为超平面关于训练集中所有样本点的函数间隔的最小值：

$$\hat{r} = \min_{i=1,2,\cdots,N} \hat{r}_i$$

可以看到，当$\boldsymbol{w}$、b成比例变化时，超平面没有改变但是函数间隔变了，因此可以对$\boldsymbol{w}$、b做相应的约束，就得到了几何间隔。

(2) 几何间隔：对于给定训练集和超平面$(\boldsymbol{w},b)$，定义超平面$(\boldsymbol{w},b)$关于样本点(x_i,y_i)的集合间隔为

$$r_i = \frac{y_i(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i + b)}{\|\boldsymbol{w}\|}$$

定义超平面$(\boldsymbol{w},b)$关于训练集的函数间隔为超平面关于训练集中所有样本点的几何间隔的最小值：

$$r = \min_{i=1,2,\cdots,N} r_i$$

这里的几何间隔就是点到平面的距离公式，因为y为1或-1，且当前数据集线性可分，所以和$\dfrac{|\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i + b|}{\|\boldsymbol{w}\|}$是等价的。

3.4.3　间隔最大化

当$\boldsymbol{w}$、b成比例变化时，函数间隔也会成比例变化，而几何间隔是不变的，所以要考虑几何间隔最大化，即求解间隔最大化的超平面的问题就变成了求解如下带约束的优化问题：

$$\begin{cases} \max\limits_{\boldsymbol{w},b} \ r \\ \text{s.t.} \ \dfrac{y_i(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i + b)}{\|\boldsymbol{w}\|} \geqslant r, i = 1,2,\cdots,N \end{cases}$$

上面的约束条件表示对于训练集中所有样本关于超平面的几何距离都至少是r。又因为函数间隔和几何间隔之间存在这样的关系：$r = \dfrac{\hat{r}}{\|\boldsymbol{w}\|}$，所以上面的优化问题可以写成如下形式：

$$\begin{cases}\max\limits_{w,b} & \dfrac{\hat{r}}{\|\boldsymbol{w}\|}\\ \text{s.t.} & (\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i+b)\geqslant\hat{r},i=1,2,\cdots,N\end{cases}$$

又因为 $\boldsymbol{w}$、b 成比例变为 $\lambda\boldsymbol{w}$、λb，函数间隔变为 $\lambda\hat{r}(\lambda>0)$，虽然改变了函数间隔但是不等式约束依然满足，并且超平面也没有变，所以 $\hat{r}$ 的取值并不影响目标函数的优化。为了方便计算，可以令 $\hat{r}=1$，并且由于最大化 $\dfrac{\hat{r}}{\|\boldsymbol{w}\|}$ 和最小化 $\dfrac{\|\boldsymbol{w}\|^2}{2}$ 是等价的，所以优化问题可以写成如下形式：

$$\begin{cases}\min\limits_{w,b} & \dfrac{\|\boldsymbol{w}\|^2}{2}\\ \text{s.t.} & y_i(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i+b)\geqslant 1,i=1,2,\cdots,N\end{cases}$$

使得上面等式成立的点也被称为支持向量(support vector)。

【例 3-2】 训练 SVM 模型。

```
import numpy as np
from sklearn import svm
from sklearn.svm import SVC
import matplotlib.pyplot as plt

clf = svm.SVC()
# 支持向量机的最大边界分类
def SVM():
    svm = SVC (kernel = 'linear', C = 1.0, random_state = 0)
    svm.fit(X_train_std, y_train)
    plot_decision_regions(X_combined_std, y_combined,
                          classifier = svm, test_idx = range(105, 150))
    plt.xlabel('花瓣长度(标准化)')
    plt.ylabel('花瓣宽度(标准化)')
    plt.legend(loc = 'upper left')
    plt.tight_layout()
    plt.show()
SVM()
```

运行程序，效果如图 3-9 所示。

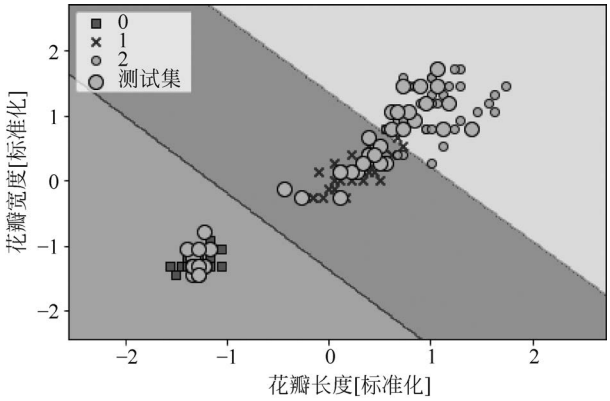


图 3-9 SVC 实现分类

3.5 核 SVM 解决非线性分类问题

支持向量机(SVM)算法除了能对线性问题进行分类外，还可以对非线性可分的问题进行分类，可以很容易地使用“核技巧”来解决非线性可分问题。

3.5.1　处理非线性不可分数据的核方法

在非线性问题中，最经典的非线性问题莫过于对于异或问题的分类，下面通过一个例子来演示。

【例 3-3】　通过 Python 生成一个异或数据集。

```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(1)
X_xor = np.random.randn(200, 2)
y_xor = np.logical_xor(X_xor[:, 0] > 0,
                       X_xor[:, 1] > 0)
y_xor = np.where(y_xor, 1, -1)

plt.scatter(X_xor[y_xor == 1, 0],
            X_xor[y_xor == 1, 1],
            c = 'b', marker = 'x',
            label = '1')
plt.scatter(X_xor[y_xor == -1, 0],
            X_xor[y_xor == -1, 1],
            c = 'r',
            marker = 's',
            label = '-1')
plt.xlim([-3, 3])
plt.ylim([-3, 3])
plt.legend(loc = 'best')
plt.tight_layout()
plt.show()
```

运行程序，得到如图 3-10 所示的随机噪声的异或数据集。

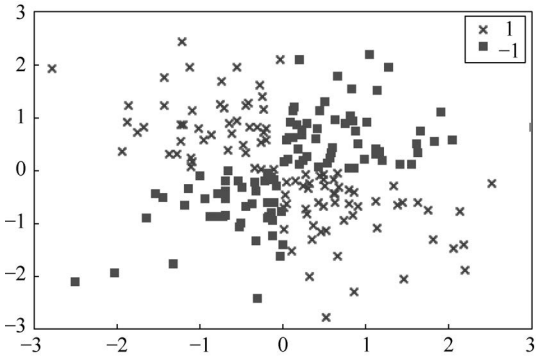


图 3-10　异或数据集

从图 3-10 可看出，不能用前面讨论过的线性逻辑回归或线性 SVM 模型所产生的线性超平面作为决策边界来分隔样本的正类和负类。然而，核方法的逻辑是针对线性不可分数据，通过映射函数 Φ 把原始特征投影到一个高维空间，特征在该空间变得线性可分，如图 3-11 所示。

3.5.2　核函数实现高维空间的分离超平面

为了使用 SVM 解决非线性问题，需要调用映射函数 Φ 将训练数据变成在高维空间上表示的特征，然后训练 SVM 模型对新特征空间的数据进行分类。可以用相同的映射函数 Φ 对

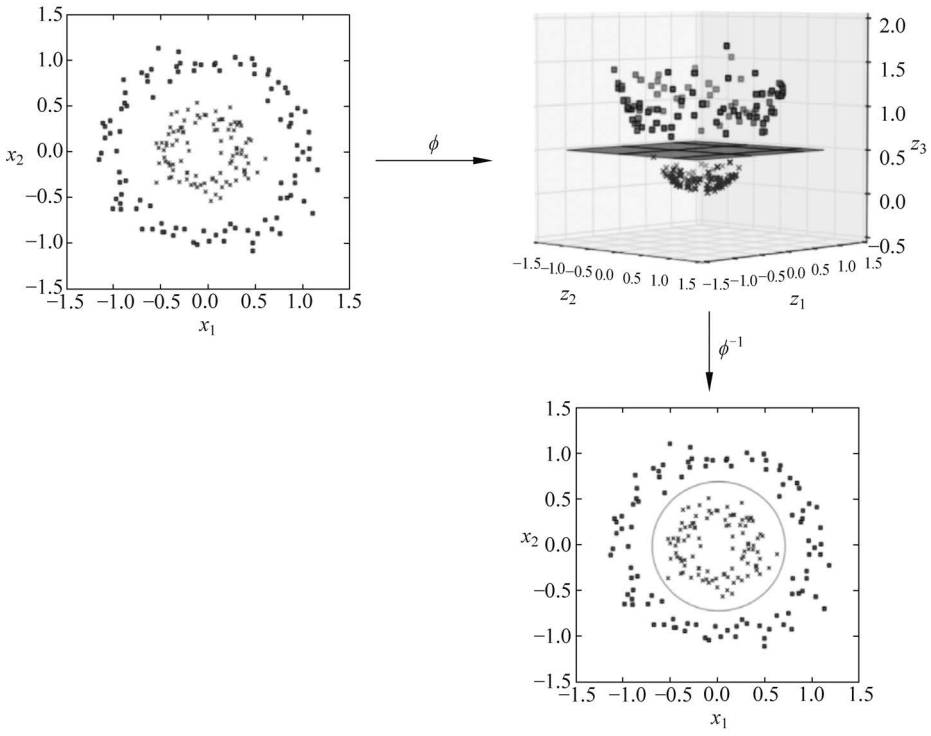


图 3-11　原始特征投影到一个高维空间效果

新的、未见过的数据进行变换，用线性 SVM 模型进行分类。

定义核函数如下：

$$\kappa(x^{(i)},x^{(j)})=\exp\left(-\frac{\|x^{(i)}-x^{(j)}\|^2}{2\sigma^2}\right)$$

通过下面代码看能否训练核 SVM 来画出非线性决策边界，以区分异或数据。

```
svm = SVC(kernel = 'rbf', random_state = 1, gamma = 0.10, C = 10.0)
svm.fit(X_xor, y_xor)
plot_decision_regions(X_xor, y_xor,
                      classifier = svm)
plt.legend(loc = 'upper left')
plt.tight_layout()
plt.show()
```

运行程序，效果如图 3-12 所示。

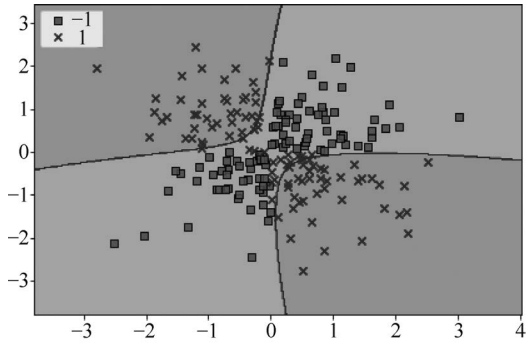


图 3-12　划分非线性决策边界效果

γ 值的不同，可能导致决策边界紧缩和波动，此处将 γ 值改为 0.5，则决策边界效果如图 3-13 所示。

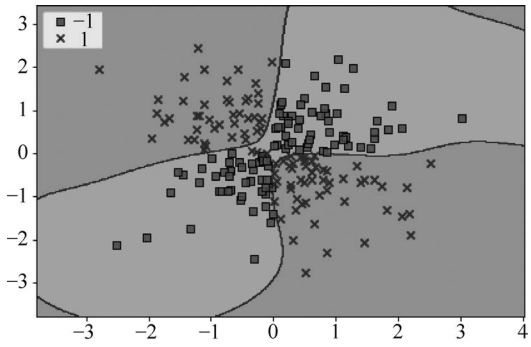


图 3-13　决策边界效果

3.6　决策树

应用最广的归纳推理算法之一是决策树(decision tree)学习，它是一种逼近离散值函数的方法。在这种方法中学习到的函数被表示为一棵决策树，学习得到的决策树也能再被表示为多个 if-then 规则，以提高可读性。

决策树学习算法有很多，如 ID3、C4.5、ASSISTANT 等。决策树学习方法为：搜索一个完整表示的假设空间，从而避免受限假设空间的不足。决策树学习的归纳偏置是优先选择较小的树。

3.6.1　何为决策树

决策树就是一棵树，一棵决策树包含一个根节点、若干内部节点和若干叶节点；叶节点对应于决策结果，其他每个节点则对应于一个属性测试；每个节点包含的样本集合根据属性测试的结果被划分到子节点中；根节点包含样本全集，从根节点到每个叶子节点的路径对应了一个判定测试序列。

【例 3-4】 在一个水果的分类问题中，采用特征向量为(颜色，尺寸，形状，味道)，其中：

颜色属性的取值范围：红、绿、黄；

尺寸属性的取值范围：大、中、小；

形状属性的取值范围：圆、细；

味道属性的取值范围：甜、酸；

样本集：一批水果，知道其特征向量及类别；

问题：一个新的水果，观测到了其特征向量，应该将其分为哪一类？

整个决策树效果如图 3-14 所示。

通常决策树代表实例属性值约束的合取(conjunction)的析取式(disjunction)。从树根到树叶的每一条路径都对应一组属性测试的合取，树本身对应这些合取的析取。

上述例子可对应如下析取式：

（颜色＝绿∧尺寸＝大）

∨（颜色＝绿∧尺寸＝中）

∨（颜色＝绿∧尺寸＝小）

∨（颜色＝黄∧形状＝圆∧尺寸＝大）

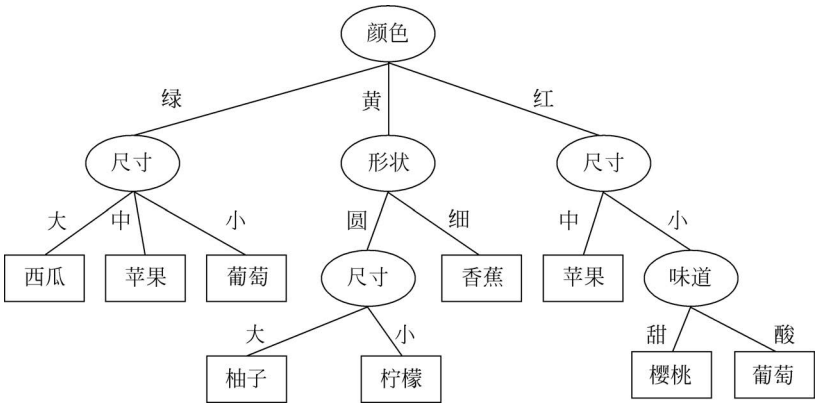


图 3-14　决策树效果

∨(颜色＝黄∧形状＝圆∧尺寸＝小)
∨(颜色＝黄∧形状＝细)
∨(颜色＝红∧尺寸＝中)
∨(颜色＝红∧尺寸＝小∧味道＝甜)
∨(颜色＝红∧尺寸＝小∧味道＝酸)

实际上,构造一棵决策树要解决的如下 4 个问题:

(1) 收集待分类的数据,这些数据的所有属性应该是完全标注的。

(2) 设计分类原则,即数据的哪些属性可以被用来分类,以及如何将该属性量化。

(3) 分类原则的选择,即在众多分类准则中,每一步选择哪一准则使最终的树更令人满意。

(4) 设计分类停止条件。

- 该节点包含的数据太少不足以分裂。
- 继续分裂数据集对树生成的目标没有贡献。
- 树的深度过大不宜再分。

3.6.2　决策树生成

1. 信息增益

决策树学习的关键在于如何选择最优的划分属性。所谓最优划分属性,对于二元分类而言,就是尽量使划分的样本属于同一类别,即“纯度”最高的属性。那么如何来度量特征的纯度,这时候就要用到“信息熵”。先来看看信息熵的定义:假如当前样本集 D 中第 k 类样本所占的比例为 $p_k(k=1,2,\cdots,|y|)$,$|y|$ 为类别的总数(对于二元分类来说,$|y|=2$)。则样本集的信息熵为

$$\mathrm{Ent}(D)=-\sum_{k=1}^{|y|}p_k\,\mathrm{lb}\,p_k$$

其中,$\mathrm{Ent}(D)$的值越小,则 D 的纯度越高。

假定离散属性 a 有 V 个可能的取值$\{a^1,a^2,\cdots,a^V\}$,如果使用特征 a 对数据集 D 进行划分,则会产生 V 个分支节点,其中第 $v(v=1,2,\cdots,V)$个节点包含了数据集 D 中所有在特征 a 上取值为 $a^v(v=1,2,\cdots,V)$的样本总数,记为 D^v。因此,可以根据上面信息熵的公式计算出信息熵。再考虑不同的分支节点所包含的样本数量不同,给分支节点赋予权重$\frac{|D^v|}{|D|}$,从这个

公式能够发现包含的样本数越多的分支节点的影响越大(这是信息增益的一个缺点,即信息增益对可取值数目多的特征有偏好,即该属性能取的值越多,信息增益越偏向这个)。因此,能够计算出特征对样本集 D 进行划分所获得的"信息增益":

$$\mathrm{Gain}(D,a)=\mathrm{Ent}(D)-\sum_{v=1}^{V}\frac{|D^v|}{|D|}\mathrm{Ent}(D^v)$$

一般而言,信息增益越大,表示使用特征对数据集划分所获得的"纯度提升"越大。所以信息增益可以用于决策树划分属性的选择,其实就是选择信息增益最大的属性。

2. 信息增益率

信息增益率的定义为

$$\mathrm{Gain\_ratio}=\frac{\mathrm{Gain}(D,a)}{\mathrm{IV}(a)}$$

其中

$$\mathrm{IV}(a)=-\sum_{v=1}^{V}\frac{|D^v|}{|D|}\mathrm{lb}\frac{|D^v|}{|D|}$$

$\mathrm{IV}(a)$称为属性 a 的"固有值",属性 a 的可能取值数目越多(即 V 越大),则 $\mathrm{IV}(a)$的值通常会越大。但增益率也可能产生一个问题就是,对可取数值数目较少的属性有所偏好,因此算法并不是直接选择使用增益率最大的候选划分属性,而是使用了一个启发式算法:先从候选划分属性中找出信息增益高于平均水平的属性,再从中选择信息增益率最高的。

3. 基尼指数

基尼指数(Gini index)也可以用于选择划分特征,如 CART(Classification And Regression Tree,分类和回归树,分类和回归都可以用)就是使用基尼指数来选择划分特征。基尼值可表示为

$$\mathrm{Gini}(D)=\sum_{k=1}^{|y|}\sum_{k'\neq k}p_k p_{k'}=1-\sum_{k=1}^{|y|}p_k^2$$

$\mathrm{Gini}(D)$反映了从数据集 D 中随机抽取两个样本,其类别标记不一致的概率,因此,$\mathrm{Gini}(D)$越小,则数据集 D 的纯度越高。则属性 a 的基尼指数定义为

$$\mathrm{Gain\_inder}=(D,a)=\sum_{v=1}^{V}\frac{|D^v|}{|D|}\mathrm{Gini}(D^v)$$

所以在候选属性集中,选择使得划分后基尼指数最小的属性作为最优划分属性。

3.6.3　决策树的剪枝

剪枝作为决策树后期处理的重要步骤,是必不可少的。没有剪枝,就是一个完全生长的决策树,是过拟合的,通常需要去掉一些不必要的节点以使得决策树模型更具有泛化能力。

决策树的剪枝方法主要有两种,分别为预剪枝和后剪枝。

1. 预剪枝(pre-pruning)

预剪枝是在构造决策树的同时进行剪枝。所有决策树的构建方法,都是在无法进一步降低熵的情况下才会停止创建分支的过程,为了避免过拟合,可以设定一个阈值,熵减小的数量小于这个阈值,即使还可以继续降低熵,也停止继续创建分支。但是这种方法实际中的效果并不好,因为在实际中,面对不同问题,很难有一个明确的阈值可以保证树模型足够好。

2. 后剪枝(post-pruning)

后剪枝的剪枝过程是删除一些子树,然后用其叶节点代替。这个叶节点所标识的类别用这棵子树中大多数训练样本所属的类别来标识。

决策树构造完成后进行剪枝。剪枝的过程是对拥有同样父节点的一组节点进行检查，判断如果将其合并，熵的增加量是否小于某一阈值。如果确实小，则这一组节点可以合并为一个节点，其中包含了所有可能的结果。后剪枝是目前最普遍的做法。

3.6.4 使用 sklearn 预测个人情况

构建决策树的算法有很多，本节实例只使用 ID3 算法构建决策树。

ID3 算法的核心是在决策树各个节点上对应信息增益准则选择特征，递归地构建决策树。具体方法：从根节点开始，对节点计算所有可能的特征的信息增益，选择信息增益最大的特征作为节点的特征，由该特征的不同取值建立子节点；再对子节点递归地调用以上方法，构建决策树；直到所有特征的信息增益均很小或没有特征可以选择为止，最后得到一棵决策树。ID3 算法相当于用极大似然法进行概率模型的选择。

在使用 ID3 算法构造决策树之前，先分析表 3-1 中的数据。

表 3-1 数据信息

ID	年龄类别	是否有工作	是否有自己的房子	信贷情况	类别(是否有贷款)
1	青年	否	否	一般	否
2	青年	否	否	好	否
3	青年	是	否	好	是
3	青年	是	是	一般	是
4	青年	否	否	一般	否
5	青年	否	否	一般	否
6	中年	否	否	一般	否
7	中年	是	是	好	是
8	中年	否	是	好	是
9	中年	否	是	非常好	是
10	中年	否	是	非常好	是
11	老年	否	是	非常好	是
12	老年	否	是	好	是
13	老年	是	否	好	是
14	老年	是	否	非常好	是
15	老年	否	否	一般	否

因为特征 A_3(是否有自己的房子)的信息增益值最大，所以选择特征 A_3 作为根节点的特征。它将训练集 D 划分为两个子集 D_1(A_3 取值为“是”)和 D_2(A_3 取值为“否”)。因为 D_1 只有同一类的样本点，所以它成为一个叶节点，节点的类标记为“是”。

对 D_2 则需要从特征 A_1(年龄类别)，A_2(是否有工作)和 A_4(信贷情况)中选择新的特征，计算各个特征的信息增益：

$$g(D_2,A_1)=H(D_2)-H(D_2\mid A_1)=0.251$$

$$g(D_2,A_2)=H(D_2)-H(D_2\mid A_2)=0.918$$

$$g(D_2,A_4)=H(D_2)-H(D_2\mid A_4)=0.474$$

根据计算，选择信息增益最大的特征 A_2(是否有工作)作为节点的特征。因为 A_2 有两个可能取值，从这一节点引出两个子节点：一个对应“是”(有工作)的子节点，包含 3 个样本，它们属于同一类，所以这是一个叶节点，类标记为“是”；另一个对应“否”(无工作)的子节点，包含 6 个样本，它们也属于同一类，所以这也是一个叶节点，类标记为“否”。这样就生成了一棵

决策树，该决策树只用了两个特征（有两个内部节点），生成的决策树如图 3-15 所示。

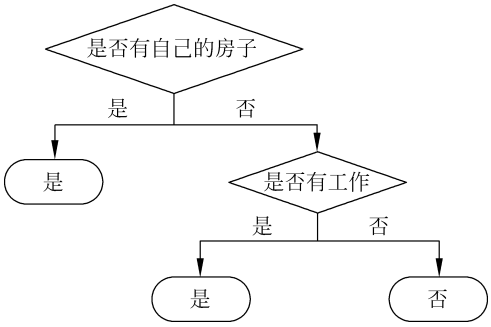


图 3-15　生成的决策树

这样就使用 ID3 算法构建出决策树，接下来，看看如何进行代码实现。

（1）构建决策树。

创建函数 majorityCnt 统计 classList 中出现此处最多的元素（类标签），创建函数 createTree 用来递归构建决策树。编写代码如下：

```
# -*- coding: UTF-8 -*-
from math import log
import operator

def calcShannonEnt(dataSet):
    """
    函数说明：计算给定数据集的经验熵(香农熵)
        dataSet：数据集
        shannonEnt：经验熵(香农熵)
    """
    numEntires = len(dataSet)                          #返回数据集的行数
    labelCounts = {}                                   #保存每个标签出现次数的字典
    for featVec in dataSet:                            #对每组特征向量进行统计
        currentLabel = featVec[-1]                     #提取标签信息
        if currentLabel not in labelCounts.keys():     #如果标签没有放入统计次数的字典，
                                                       #则添加进去
            labelCounts[currentLabel] = 0
        labelCounts[currentLabel] += 1                 #标签计数
    shannonEnt = 0.0                                   #经验熵(香农熵)
    for key in labelCounts:                            #计算香农熵
        prob = float(labelCounts[key]) / numEntires    #选择该标签的概率
        shannonEnt -= prob * log(prob, 2)              #利用公式计算
    return shannonEnt                                  #返回经验熵(香农熵)

def createDataSet():
    """
    函数说明：创建测试数据集
        dataSet：数据集
        labels：特征标签
    """
    dataSet = [[0, 0, 0, 0, 'no'],                     #数据集
               [0, 0, 0, 1, 'no'],
               [0, 1, 0, 1, 'yes'],
               [0, 1, 1, 0, 'yes'],
               [0, 0, 0, 0, 'no'],
               [1, 0, 0, 0, 'no'],
               [1, 0, 0, 1, 'no'],
               [1, 1, 1, 1, 'yes'],
```

```
                [1, 0, 1, 2, 'yes'],
                [1, 0, 1, 2, 'yes'],
                [2, 0, 1, 2, 'yes'],
                [2, 0, 1, 1, 'yes'],
                [2, 1, 0, 1, 'yes'],
                [2, 1, 0, 2, 'yes'],
                [2, 0, 0, 0, 'no']]
    labels = ['年龄类别', '是否有工作', '是否有自己的房子', '信贷情况']        #特征标签
    return dataSet, labels                                    #返回数据集和分类属性

def splitDataSet(dataSet, axis, value):
    """
    函数说明:按照给定特征划分数据集
    dataSet:待划分的数据集
      axis:划分数据集的特征
      value:需要返回的特征的值
    """
    retDataSet = []                                           #创建返回的数据集列表
    for featVec in dataSet:                                   #遍历数据集
        if featVec[axis] == value:
            reducedFeatVec = featVec[:axis]                   #去掉 axis 特征
            reducedFeatVec.extend(featVec[axis+1:])           #将符合条件的添加到返回的数据集
            retDataSet.append(reducedFeatVec)
    return retDataSet                                         #返回划分后的数据集

def chooseBestFeatureToSplit(dataSet):
    """
    函数说明: 选择最优特征
        dataSet: 数据集
      bestFeature: 信息增益最大的(最优)特征的索引值
    """
    numFeatures = len(dataSet[0]) - 1                         #特征数量
    baseEntropy = calcShannonEnt(dataSet)                     #计算数据集的香农熵
    bestInfoGain = 0.0                                        #信息增益
    bestFeature = -1                                          #最优特征的索引值
    for i in range(numFeatures):                              #遍历所有特征
        #获取 dataSet 的第 i 个特征
        featList = [example[i] for example in dataSet]
        uniqueVals = set(featList)                            #创建 set 集合{},元素不可重复
        newEntropy = 0.0                                      #经验条件熵
        for value in uniqueVals:                                  #计算信息增益
            subDataSet = splitDataSet(dataSet, i, value)          #subDataSet 划分后的子集
            prob = len(subDataSet) / float(len(dataSet))          #计算子集的概率
            newEntropy += prob * calcShannonEnt(subDataSet)       #根据公式计算经验条件熵
        infoGain = baseEntropy - newEntropy                       #信息增益
            if (infoGain > bestInfoGain):                         #计算信息增益
            bestInfoGain = infoGain                 #更新信息增益,找到最大的信息增益
            bestFeature = i                         #记录信息增益最大的特征的索引值
    return bestFeature                              #返回信息增益最大的特征的索引值

def majorityCnt(classList):
    """
    函数说明: 统计 classList 中出现最多的元素(类标签)
        classList: 类标签列表
      sortedClassCount[0][0]:出现最多的元素(类标签)
    """
    classCount = {}
    for vote in classList:                          #统计 classList 中每个元素出现的次数
        if vote not in classCount.keys():classCount[vote] = 0
```

```
        classCount[vote] += 1
    sortedClassCount = sorted(classCount.items(), key = operator.itemgetter(1), reverse = True)
                                                    #根据字典的值降序排序
    return sortedClassCount[0][0]                   #返回 classList 中出现次数最多的元素

def createTree(dataSet, labels, featLabels):
    """
    函数说明: 创建决策树
      dataSet: 训练数据集
       labels: 分类属性标签
       featLabels: 存储选择的最优特征标签
       myTree: 决策树
    """
    classList = [example[-1] for example in dataSet]    #取分类标签(是否有贷款:yes or no)
    if classList.count(classList[0]) == len(classList):   #如果类别完全相同则停止继续划分
        return classList[0]
    if len(dataSet[0]) == 1 or len(labels) == 0:        #遍历完所有特征时返回出现次数最多
                                                        #的类标签
        return majorityCnt(classList)
    bestFeat = chooseBestFeatureToSplit(dataSet)        #选择最优特征
    bestFeatLabel = labels[bestFeat]                    #最优特征的标签
    featLabels.append(bestFeatLabel)
    myTree = {bestFeatLabel:{}}                         #根据最优特征的标签生成树
    del(labels[bestFeat])                               #删除已经使用的特征标签
    featValues = [example[bestFeat] for example in dataSet]      #得到训练集中所有最优特征的
                                                                 #属性值
    uniqueVals = set(featValues)                        #去掉重复的属性值
    for value in uniqueVals:                            #遍历特征,创建决策树
        subLabels = labels[:]
        myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet, bestFeat, value),
subLabels, featLabels)
    return myTree

if __name__ == '__main__':
    dataSet, labels = createDataSet()
    featLabels = []
    myTree = createTree(dataSet, labels, featLabels)
    print(myTree)
```

运行程序,输出如下:

```
{'是否有自己的房子': {0: {'是否有工作': {0: 'no', 1: 'yes'}}, 1: 'yes'}}
```

递归创建决策树时,递归有两个终止条件:第一个停止条件是所有的类标签完全相同,则直接返回该类标签;第二个停止条件是使用完了所有特征,仍然不能将数据划分仅包含唯一类别的分组,即决策树构建失败,特征不够用。此时说明数据维度不够,由于第二个停止条件无法简单地返回唯一的类标签,这里挑选出现数量最多的类别作为返回值。

由结果可见,决策树已经构建完成了。为了使结果更直观,可以使用强大的 Matplotlib 绘制决策树。

(2) 决策树可视化。

在实现可视化代码中,需要用到的 Matplotlib 可视化函数有:

- getNumLeafs:获取决策树叶节点的数目。
- getTreeDepth:获取决策树的层数。
- plotNode:绘制节点。
- plotMidText:标注有向边属性值。

- plotTree：绘制决策树。
- createPlot：创建绘制面板。

代码编写如下：

```
# -*- coding: UTF-8 -*-
from matplotlib.font_manager import FontProperties
import matplotlib.pyplot as plt
from math import log
import operator

... #此处省略与上面相同的代码
def getNumLeafs(myTree):
    """
    函数说明:获取决策树叶节点的数目
        myTree: 决策树
        numLeafs: 决策树的叶节点的数目
    """
    numLeafs = 0                                        #初始化叶节点
    firstStr = next(iter(myTree))        #Python 3 中 myTree.keys 返回的是 dict_keys,不是 list
          #所以不能使用 myTree.keys()[0]的方法获取节点属性,可以使用 list(myTree.keys())[0]
    secondDict = myTree[firstStr]                       #获取下一组字典
    for key in secondDict.keys():
        if type(secondDict[key]).__name__ == 'dict':    #测试该节点是否为字典,如果不是字典
                                                        #则代表此节点为叶节点
            numLeafs += getNumLeafs(secondDict[key])
        else: numLeafs += 1
    return numLeafs

def getTreeDepth(myTree):
    """
    函数说明:获取决策树的层数
        myTree: 决策树
        maxDepth: 决策树的层数
    """
    maxDepth = 0                                        #初始化决策树深度
    firstStr = next(iter(myTree))        #Python 3 中 myTree.keys 返回的是 dict_keys,不是 list
           #所以不能使用 myTree.keys()[0]方法获取节点属性,可以使用 list(myTree.keys())[0]
    secondDict = myTree[firstStr]                       #获取下一个字典
    for key in secondDict.keys():
        if type(secondDict[key]).__name__ == 'dict':    #测试该节点是否为字典,如果不是字典,
                                                        #则代表此节点为叶节点
            thisDepth = 1 + getTreeDepth(secondDict[key])
        else: thisDepth = 1
        if thisDepth > maxDepth: maxDepth = thisDepth   #更新层数
    return maxDepth

def plotNode(nodeTxt, centerPt, parentPt, nodeType):
    """
    函数说明:绘制节点
        nodeTxt: 节点名
        centerPt: 文本位置
        parentPt: 标注的箭头位置
        nodeType: 节点格式
    """
    arrow_args = dict(arrowstyle="<-")                  #定义箭头格式
    font = FontProperties(fname=r"c:\windows\fonts\simsun.ttc", size=14)        #设置中文字体
    createPlot.ax1.annotate(nodeTxt, xy=parentPt, xycoords='axes fraction',     #绘制节点
        xytext=centerPt, textcoords='axes fraction',
        va="center", ha="center", bbox=nodeType, arrowprops=arrow_args, FontProperties=font)
```

```
def plotMidText(cntrPt, parentPt, txtString):
    """
    函数说明:标注有向边属性值
        cntrPt、parentPt:用于计算标注位置
        txtString:标注的内容
    """
    xMid = (parentPt[0] - cntrPt[0])/2.0 + cntrPt[0]
     #计算标注位置
    yMid = (parentPt[1] - cntrPt[1])/2.0 + cntrPt[1]
    createPlot.ax1.text(xMid, yMid, txtString, va = "center", ha = "center", rotation = 30)

def plotTree(myTree, parentPt, nodeTxt):
    """
    函数说明: 绘制决策树
        myTree: 决策树(字典)
        parentPt: 标注的内容
        nodeTxt: 节点名
    """
        decisionNode = dict(boxstyle = "sawtooth", fc = "0.8")    #设置节点格式
    leafNode = dict(boxstyle = "round4", fc = "0.8")              #设置叶节点格式
    numLeafs = getNumLeafs(myTree)       #获取决策树叶节点数目,决定了树的宽度
    depth = getTreeDepth(myTree)                                  #获取决策树层数
    firstStr = next(iter(myTree))                                 #下一个字典
    cntrPt = (plotTree.xOff + (1.0 + float(numLeafs))/2.0/plotTree.totalW, plotTree.yOff)
                                                                  #中心位置
    plotMidText(cntrPt, parentPt, nodeTxt)                        #标注有向边属性值
    plotNode(firstStr, cntrPt, parentPt, decisionNode)            #绘制节点
    secondDict = myTree[firstStr]        #下一个字典,也就是继续绘制子树
    plotTree.yOff = plotTree.yOff - 1.0/plotTree.totalD           #y 偏移
    for key in secondDict.keys():
        if type(secondDict[key]).__name__ == 'dict':
    #测试该节点是否为字典,如果不是字典,则代表此节点为叶节点
            plotTree(secondDict[key],cntrPt,str(key))
            #不是叶节点,递归调用继续绘制
            else:
            #如果是叶节点,则绘制叶节点,并标注有向边属性值
                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

def createPlot(inTree):
    """
    函数说明: 创建绘制面板
        inTree: 决策树(字典)
    """
    fig = plt.figure(1, facecolor = 'white')
#创建图形
    fig.clf()
#清空图形
    axprops = dict(xticks = [], yticks = [])
    createPlot.ax1 = plt.subplot(111, frameon = False, **axprops)
#去掉 x、y 轴
    plotTree.totalW = float(getNumLeafs(inTree))
#获取决策树叶节点数目
    plotTree.totalD = float(getTreeDepth(inTree))
#获取决策树层数
    plotTree.xOff = -0.5/plotTree.totalW; plotTree.yOff = 1.0;
```

```
    #x偏移
        plotTree(inTree, (0.5,1.0), '')
    #绘制决策树
        plt.show()
    #显示绘制结果

if __name__ == '__main__':
    dataSet, labels = createDataSet()
    featLabels = []
    myTree = createTree(dataSet, labels, featLabels)
    print(myTree)
    createPlot(myTree)
```

运行程序,输出如下,得到决策树效果如图 3-16 所示。

```
{'是否有自己的房子': {0: {'是否有工作': {0: 'no', 1: 'yes'}}, 1: 'yes'}}
```

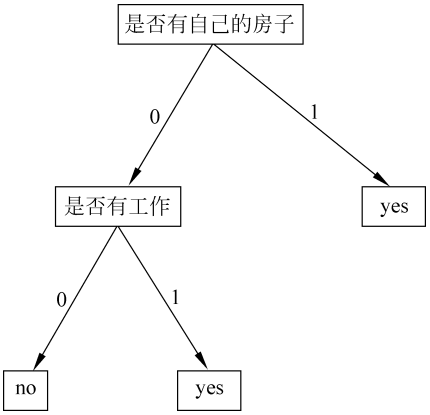


图 3-16　生成的决策树

代码中,plotNode 函数的工作就是绘制各个节点,如是否有自己的房子、是否有工作、yes、no,包括内节点和叶节点。plotMidText 函数的工作就是绘制各个有向边的属性,如各个有向边的 0 和 1。

(3) 使用决策树执行分类。

依靠训练数据构造了决策树之后,可以将它用于实际数据的分类。首先,在执行数据分类时,需要决策树及用于构造树的标签向量;然后,比较测试数据与决策树上的数值,递归执行该过程直到进入叶节点;最后,将测试数据定义为叶节点所属的类型。在构建决策树的代码中,可以看到,有一个 featLabels 参数,它是用来记录各个分类节点的,在用决策树做预测时,按顺序输入需要的分类节点的属性值即可。用决策树做分类的代码很简单,具体代码如下:

```
#  - * - coding: UTF - 8 - * -
from math import log
import operator
... #此处省略与上面相同的代码

def classify(inputTree, featLabels, testVec):
    """
    函数说明: 使用决策树分类
        inputTree: 已经生成的决策树
        featLabels: 存储选择的最优特征标签
        testVec: 测试数据列表,顺序对应最优特征标签
        classLabel: 分类结果
    """
    firstStr = next(iter(inputTree))
    #获取决策树节点
    secondDict = inputTree[firstStr]
    #下一个字典
    featIndex = featLabels.index(firstStr)
    for key in secondDict.keys():
        if testVec[featIndex] == key:
            if type(secondDict[key]).__name__ == 'dict':
                classLabel = classify(secondDict[key], featLabels, testVec)
            else: classLabel = secondDict[key]
```

```
    return classLabel

if __name__ == '__main__':
    dataSet, labels = createDataSet()
    featLabels = []
    myTree = createTree(dataSet, labels, featLabels)
    testVec = [0,1]           #测试数据
    result = classify(myTree, featLabels, testVec)
    if result == 'yes':
        print('放贷')
    if result == 'no':
        print('不放贷')
```

这里只增加了 classify 函数，用于决策树分类。输入测试数据[0,1]，它代表没有房子，但是有工作，分类结果如下所示：

```
放贷
```

那是不是每次做预测都要训练一次决策树呢？并不是这样的，可通过决策树的存储方法解决此问题。

(4) 决策树的存储。

即使处理很小的数据集，构造决策树也是很耗时的，如前面的样本数据，也要花费几秒的时间，如果数据集很大，耗费的计算时间将会很长。然而，用创建好的决策树解决分类问题，则可以很快完成。因此，为了节省计算时间，最好在每次执行分类时调用已经构造好的决策树。为了解决这个问题，需要使用 Python 模块 pickle 序列化对象。序列化对象后即可在磁盘上保存对象，并在需要时将其读取出来。

假设已经得到决策树{'是否有自己的房子'：{0：{'是否有工作'：{0：'no'，1：'yes'}}，1：'yes'}}，使用 pickle.dump 存储决策树。

```
# -*- coding: UTF-8 -*-
import pickle

def storeTree(inputTree, filename):
    """
    函数说明: 存储决策树
        inputTree: 已经生成的决策树
        filename: 决策树的存储文件名
    """
    with open(filename, 'wb') as fw:
        pickle.dump(inputTree, fw)

if __name__ == '__main__':
    myTree = {'是否有自己的房子': {0: {'是否有工作': {0: 'no', 1: 'yes'}}, 1: 'yes'}}
    storeTree(myTree, 'classifierStorage.txt')
```

运行代码，在该 Python 文件的相同目录下，会生成一个名为 classifierStorage.txt 的文本文件，这个文件二进制存储着决策树，可以使用 Sublime Text 将文件打开查看存储结果，如图 3-17 所示。

```
1  8003 7d71 0058 1200 0000 e69c 89e8 87aa
2  e5b7 b1e7 9a84 e688 bfe5 ad90 7101 7d71
3  0228 4b00 7d71 0358 0900 0000 e69c 89e5
4  b7a5 e4bd 9c71 047d 7105 284b 0058 0200
5  0000 6e6f 7106 4b01 5803 0000 0079 6573
6  7107 7573 4b01 6807 7573 2e
```

图 3-17　存储的 classifierStorage.txt 文件

图 3-17 所示的内容是一个二进制存储的文件，我们不需要看懂里面的内容，会存储、会用即可。如果下次需要使用这个存储完的二进制文件，使用 pickle.load 进行载入即可。编写代码如下：

```
# - * - coding: UTF - 8 - * -
import pickle

def grabTree(filename):
    """
    函数说明: 读取决策树
        filename: 决策树的存储文件名
        pickle.load(fr): 决策树字典
    """
    fr = open(filename, 'rb')
    return pickle.load(fr)

if __name__ == '__main__':
    myTree = grabTree('classifierStorage.txt')
    print(myTree)
```

运行程序，输出如下：

```
{'是否有自己的房子': {0: {'是否有工作': {0: 'no', 1: 'yes'}}, 1: 'yes'}}
```

从上述结果中可以看到，已顺利加载了存储决策树的二进制文件。

3.7 K 近邻算法

一种最经典和最简单的有监督学习方法之一是 K 近邻（K-Nearest Neighbor，KNN）算法。K 近邻算法是最简单的分类器，没有显式的学习过程或训练过程，属于懒惰学习（lazy learning）。当对数据的分布只有很少或者没有任何先验知识时，K 近邻算法是一个不错的选择。

3.7.1 K 近邻算法的原理

K 近邻算法除了可以用来解决分类问题，还可用来解决回归问题。它有着非常简单的原理：当对测试样本进行分类时，首先通过扫描训练样本集，找到与该测试样本最相似的 k 个训练样本，根据这个样本的类别进行投票确定测试样本的类别。也即可通过单个样本与测试样本的相似程度进行加权。如果需要以测试样本对应每类的概率的形式输出，可以通过 k 个样本中不同类别的样本数量分布来进行估计。

K 近邻算法三要素分别为：距离度量、k 值的选择、分类决策规则。

1. 距离度量

特征空间中两个实例点之间的距离是二者相似程度的反映，所以 K 近邻算法中一个重要的问题是计算样本之间的距离，以确定训练样本中哪些样本与测试样本更加接近。

在实际应用中，距离计算方法往往需要根据应用的场景和数据本身的特点来选择。当已有的距离方法不能满足实际应用需求时，还需要有针对性地提出适合具体问题的距离度量方法。

设特征空间 χ 是 n 维实数向量空间，$\boldsymbol{x}_i, \boldsymbol{x}_j \in \chi$，$\boldsymbol{x}_i = (x_i^{(1)}, x_i^{(2)}, \cdots, x_i^{(n)})^{\mathrm{T}}$，$\boldsymbol{x}_j = (x_j^{(1)}, x_j^{(2)}, \cdots, x_j^{(n)})^{\mathrm{T}}$，则 $\boldsymbol{x}_i$、$\boldsymbol{x}_j$ 的 L_p 距离定义为

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

- 当 $p=2$ 时，为欧氏距离(Euclidean distance)。
- 当 $p=1$ 时，为曼哈顿距离(Manhattan distance)。
- 当 $p=\infty$ 时，为各个坐标距离的最大值。

图 3-18 为二维空间中，与原点的 L_p 距离为 1 的点的图形($L_p=1$)。

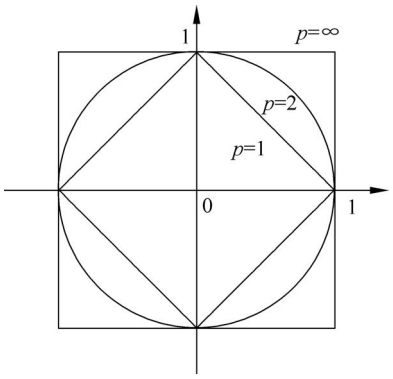


图 3-18　L_p 距离间的关系

2. k 值的选择

正常情况下，从 $k=1$ 开始，随着 k 的逐渐增大，K 近邻算法的分类效果会逐渐提升；在增大到某个值后，随着 k 的进一步增大，K 近邻算法的分类效果会逐渐下降。

k 值较小，相当于用较小的邻域中的训练实例进行预测，只有距离近的(相似的)起作用：

- 单个样本影响大。
- “学习”的近似误差(approximation error)会减小，但估计误差(estimation error)会增大。
- 噪声敏感。
- 整体模型变得复杂，容易发生过拟合。

k 值较大，这时距离远的(不相似的)也会起作用：

- 近似误差会增大，但估计误差会减小。
- 整体的模型变得简单。

3. 分类决策规则

分类决策规则一般都是多数表决规则(majority voting rule)，为新数据点距离最近的数据点的多数类决定新数据点的类别，实现函数如下：

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

3.7.2　K 近邻算法的实现

K 近邻算法的完整实现过程如下：

(1) 确定 k 的大小和距离计算方法。

(2) 从训练样本中得到 k 个与测试最相似的样本。

① 计算测试数据与各个训练数据之间的距离；

② 按照距离的递增关系进行排序；

③ 选取距离最小的 k 个点；

④ 确定前 k 个点所在类别的出现频率；

⑤ 返回前 k 个点中出现频率最高的类别作为测试数据的预测分类。

(3) 根据 k 个组相似样本的类别，通过投票的方式来确定测试样本的类别。

【例 3-5】 sklearn 的 K 近邻算法实现。

实现步骤如下：

(1) 导入包、导入数据。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
#加载分类模型
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

iris = datasets.load_iris()
X = iris.data[:, :2]                    #加载 Iris 数据集的目标
y = iris.target                         #加载 Iris 数据集的前两个特征
```

(2) 划分数据。

```
from sklearn.neighbors import KNeighborsClassifier
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify = y, random_state = 0)
```

(3) 交叉验证。

```
from sklearn.model_selection import cross_val_score          #导入包
knn_3_clf = KNeighborsClassifier(n_neighbors = 3)            #实例化对象,k 取 3,最近的 3 个点
knn_5_clf = KNeighborsClassifier(n_neighbors = 5)
knn_3_scores = cross_val_score(knn_3_clf, X_train, y_train, cv = 10)     #训练,10 折
knn_5_scores = cross_val_score(knn_5_clf, X_train, y_train, cv = 10)
print("knn_3 平均分数: ", knn_3_scores.mean(), "knn_3 标准: ",knn_3_scores.std())
print("knn_3 平均分数: ", knn_5_scores.mean(), " knn_3 标准: ",knn_5_scores.std())
knn_3 平均分数: 0.7983333333333333 knn_3 标准: 0.09081421817216852
knn_3 平均分数: 0.8066666666666666 knn_3 标准: 0.05593205754956987

all_scores = []
for n_neighbors in range(3,9,1):
    knn_clf = KNeighborsClassifier(n_neighbors = n_neighbors)
    all_scores.append((n_neighbors, cross_val_score(knn_clf, X_train, y_train, cv = 10).mean()))

print(sorted(all_scores, key = lambda x:x[0], reverse = True))          #按索引输出
print(sorted(all_scores, key = lambda x:x[1], reverse = True))          #从高分到低分输出
[(8, 0.7983333333333333), (7, 0.8261111111111111), (6, 0.8233333333333335),
(5, 0.8066666666666666), (4, 0.8511111111111112), (3, 0.7983333333333333)]
[(4, 0.8511111111111112), (7, 0.8261111111111111), (6, 0.8233333333333335),
(5, 0.8066666666666666), (3, 0.7983333333333333), (8, 0.7983333333333333)]
```

(4) 图解。

```
import mglearn mglearn.plots.plot_knn_classification(n_neighbors = 1)
```

当 $k=3$ 时，效果如图 3-19 所示。

(5) 分类。

```
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
plt.rcParams['font.sans-serif'] = ['SimHei']      #显示中文

X, y = mglearn.datasets.make_forge()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 0)
print(X_test.shape)
print(y_test.shape)
```

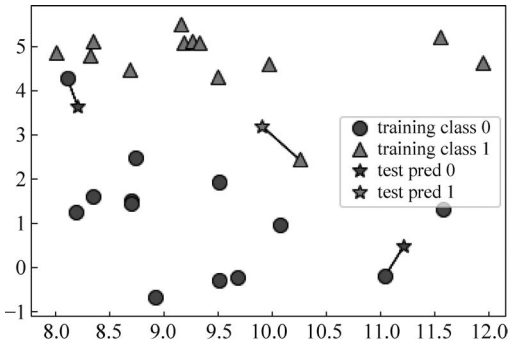


图 3-19　数据图解效果

```
print(X_test)
print(y_test)                                    #在测试集上真实的值

from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors = 3)
clf.fit(X_train, y_train)
print("测试集预测:", clf.predict(X_test))        #在测试集上预测的值
print("测试集准确性: {:.2f}".format(clf.score(X_test, y_test)))       #精度

fig, axes = plt.subplots(1, 3, figsize = (10, 3))    #1 行 3 列
for n_neighbors, ax in zip([1, 3, 9], axes):         #n_neighbors = [1, 3, 9], ax = 1,2,3 (循环取值)
    clf = KNeighborsClassifier(n_neighbors = n_neighbors).fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill = True, eps = 0.5, ax = ax, alpha = .4)
    #产生可视化的决策边界
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax = ax)
    ax.set_title("{} 近邻(s)".format(n_neighbors))
    ax.set_xlabel("特征 0")
    ax.set_ylabel("特征 1")
axes[0].legend(loc = 3)
X.shape
y.shape
```

运行程序,输出如下,效果如图 3-20 所示。

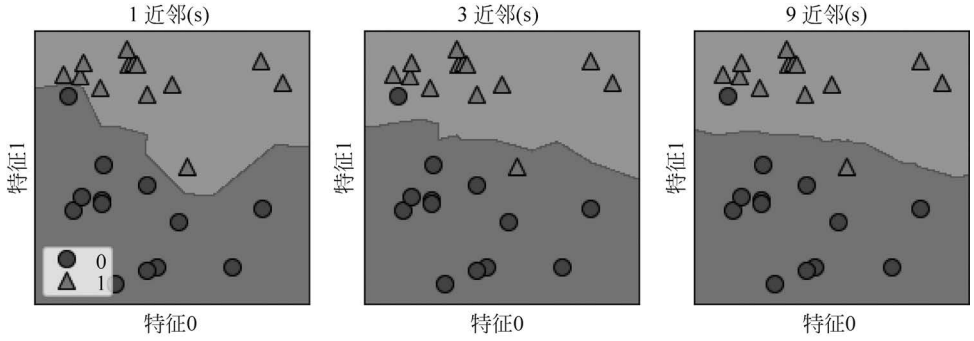


图 3-20　分类效果

```
(7, 2)
(7,)
[[11.54155807  5.21116083]
 [10.06393839  0.99078055]
 [ 9.49123469  4.33224792]
 [ 8.18378052  1.29564214]
 [ 8.30988863  4.80623966]
 [10.24028948  2.45544401]
```

```
 [ 8.34468785  1.63824349]]
[1 0 1 0 1 1 0]
测试集预测: [1 0 1 0 1 0 0]
测试集准确性: 0.86
(26,)
```

(6) 回归。

```
mglearn.plots.plot_knn_regression(n_neighbors = 1)
```

运行程序,效果如图 3-21 所示。

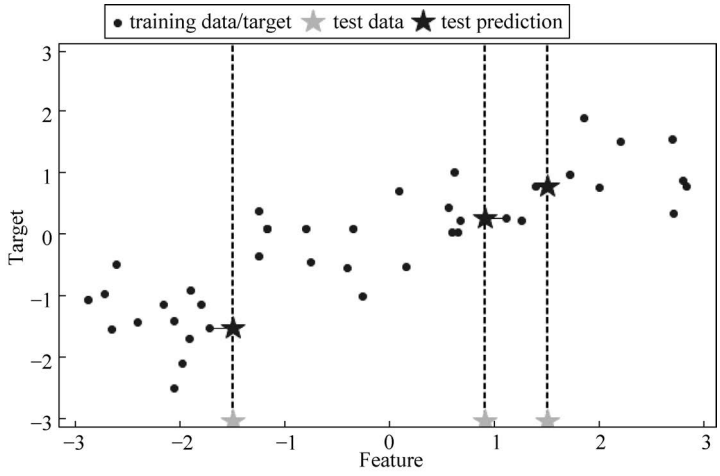


图 3-21　回归图 1

从图 3-21 中可以看出,从 test data 中产生 test prediction,然后找出最近的一个点:

```
mglearn.plots.plot_knn_regression(n_neighbors = 3)
```

运行程序,效果如图 3-22 所示。

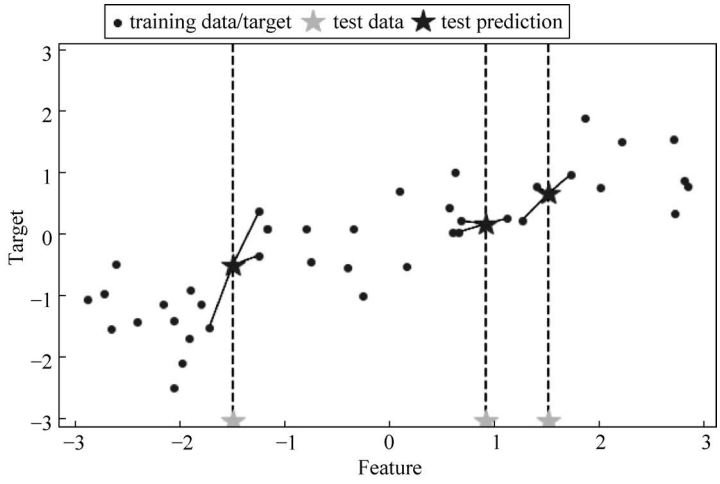


图 3-22　找出最近的一个点

从图 3-22 中可以看出,从 test data 中产生 test prediction,然后找出最近的 3 个点:

```
#步骤: 导入包、实例化、训练、预测、打分
from sklearn.neighbors import KNeighborsRegressor

X, y = mglearn.datasets.make_wave(n_samples = 40)
```

```
#将 wave 数据集拆分为训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 0)
#实例化模型,并将要考虑的邻居数量设置为 3
reg = KNeighborsRegressor(n_neighbors = 3)
#使用训练数据和训练目标拟合模型
reg.fit(X_train, y_train)
print("测试集预测:\n", reg.predict(X_test))
print("预测集: {:.2f}".format(reg.score(X_test, y_test)))
```

运行程序,输出如下:

```
测试集预测:
 [ -0.05396539   0.35686046   1.13671923   -1.89415682   -1.13881398   -1.63113382
  0.35686046   0.91241374   -0.44680446   -1.13881398]
预测集: 0.83
```

```
import matplotlib
matplotlib.rcParams['axes.unicode_minus'] = False
fig, axes = plt.subplots(1, 3, figsize = (15, 4))
#创建 1000 个数据点,均匀分布在 -3 和 3 之间
line = np.linspace( -3, 3, 1000).reshape( -1, 1)
for n_neighbors, ax in zip([1, 3, 9], axes):
    #使用 1、3 或 9 个邻居进行预测
    reg = KNeighborsRegressor(n_neighbors = n_neighbors)          #实例化
    reg.fit(X_train, y_train)                                     #用 reg 对象的 fit 方法训练
    ax.plot(line, reg.predict(line))                              #用 reg 对象的 predict 预测
    ax.plot(X_train, y_train, '^', c = mglearn.cm2(0), markersize = 8)
    ax.plot(X_test, y_test, 'v', c = mglearn.cm2(1), markersize = 8)
    ax.set_title(
        "{} 近邻(s)\n 训练分数: {:.2f} 测试分数: {:.2f}".format(
            n_neighbors, reg.score(X_train, y_train),             #循环设置标题
            reg.score(X_test, y_test)))
    ax.set_xlabel("特征")
    ax.set_ylabel("目标")
axes[0].legend(["预测模型", "训练数据/目标",
                "测试数据/目标"], loc = "best")                   #图例、说明
```

运行程序,效果如图 3-23 所示。

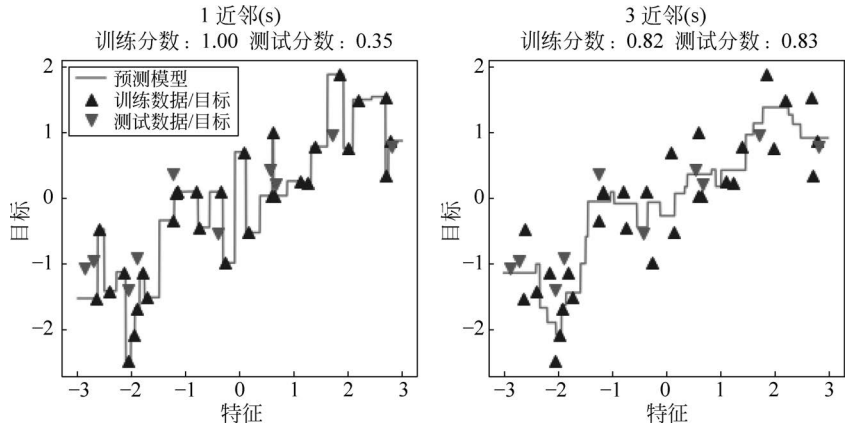


图 3-23　数据拟合效果

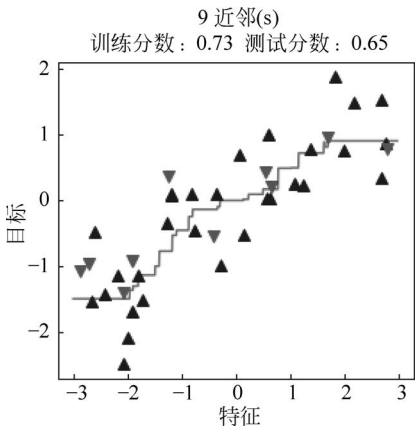


图 3-23(续)

3.8 贝叶斯算法

贝叶斯算法与大多数机器学习算法不同，如决策树、逻辑回归、支持向量机等算法，这些算法都是判别方法，可通过一个决策函数 $y=f(x)$ 或者条件分布 $p(y|x)$ 直接学习出特征输出 y 和特征 x 之间的关系。而贝叶斯算法的生成方法为：找出特征输出 y 和特征 x 的联合分布 $p(x,y)$，然后用 $p(y|x)=\dfrac{p(x,y)}{p(x)}$ 得出。

3.8.1 贝叶斯算法的基本思想

朴素贝叶斯算法假设自变量特征之间条件独立，可以概括为：先验概率＋数据＝后验概率。

1. 条件独立

如果 x、y 互相独立，则有

$$p(x,y)=p(x)\times p(y)$$

2. 条件概率

条件概率为

$$p(y\mid x)=\frac{p(x,y)}{p(x)}$$

$$p(x\mid y)=\frac{p(x,y)}{p(y)}$$

$$p(y\mid x)p(x)=p(x\mid y)p(y)$$

最后得到贝叶斯公式为

$$p(y\mid x)=\frac{p(x\mid y)p(y)}{p(x)}$$

3.8.2 贝叶斯算法的模型

假设有 m 个样本数据：

$$(x_1^{(1)},x_2^{(1)}\cdots,x_n^{(1)},y_1),(x_1^{(2)},x_2^{(2)}\cdots,x_n^{(2)},y_2),\cdots,(x_1^{(m)},x_2^{(m)}\cdots,x_n^{(m)},y_m)$$

每一个样本特征 $\boldsymbol{x}$ 有 n 个体征，标签 y 有 k 个类别，定义为 $c_1,c_2,\cdots,c_k$。从已有的样本中很容易得到先验概率分布 $p(y=c_k)(k=1,2,\cdots,m)$。

而条件概率分布为

$$p(\boldsymbol{x}=\boldsymbol{X}\mid y=c_k)=p(x_1=X_1,x_2=X_2,x_n=X_n\mid y=c_k)$$

然后就可以用贝叶斯公式得到 $\boldsymbol{x}$、y 的联合分布 $P(\boldsymbol{x},y)$，联合分布 $p(\boldsymbol{x},y)$ 定义为

$$p(\boldsymbol{x},y=c_k)=p(y=c_k)p(\boldsymbol{x}=\boldsymbol{X}\mid y=c_k)=p(y=c_k)p(x_1=X_1,$$
$$x_2=X_2,\cdots,x_n=X_n\mid y=c_k)$$

从上面可以看出 $p(y=c_k)$ 很容易得到，统计一下各类被占的比例（频数）即可求出。但是 $p(x_1=X_1,x_2=X_2,\cdots,x_n=X_n\mid y=c_k)$ 很难求出，因为这是一个很复杂的有 n 个维度的条件分布，因此朴素贝叶斯算法在这里做了一个大胆的假设：$\boldsymbol{x}$ 的 n 个维度之间相互独立（也就是特征之间条件独立），则可得到：

$$p(x_1=X_1,x_2=X_2,\cdots,x_n=X_n\mid y=c_k)$$
$$=p(x_1=X_1\mid y=c_k)p(x_2=X_2\mid y=c_k)\cdots p(x_n=X_n\mid y=c_k)$$

这大大简化了 n 维条件概率分布的难度，图 3-24 给出了贝叶斯算法的整体模型。

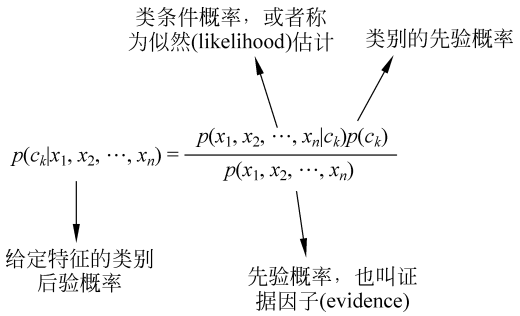


图 3-24　贝叶斯算法的整体模型

3.8.3　用 sklearn 实现贝叶斯分类

sklearn 提供了 3 个朴素贝叶斯的分类器，分别为：

- naive_bayes.GaussianNB：高斯分布下的朴素贝叶斯。
- naive_bayes.MultinomialNB：多项式分布下的朴素贝叶斯。
- naive_bayes.BernoulliNB：伯努利分布下的朴素贝叶斯。

1. 高斯朴素贝叶斯

GaussianNB 实现了运用于分类的高斯朴素贝叶斯算法。概率假设为高斯分布：

$$p(x_i\mid y)=\frac{1}{\sqrt{2\pi\sigma_y^2}}\exp\left(-\frac{(x_i-\mu_y)^2}{2\sigma_y^2}\right)$$

参数均值和方差都使用最大似然估计。

2. 多项式朴素贝叶斯

MultinomialNB 假设特征的先验概率为多项式分布：

$$p(x_j=X_{jl}\mid y=c_k)=\frac{X_{jl}+\lambda}{m_k+n\lambda}$$

MultinomialNB 参数比 GaussianNB 多，一共有 3 个参数。其中，参数 alpha 即为公式中的常数 λ，默认值为 1。如果发现拟合得不好，需要调优时，可以选择稍大于 1 或者稍小于 1 的数。参数 fit_prior 表示是否要考虑先验概率，如果是 false，则所有的样本类别输出都有相同的类别先验概率。否则可以用第 3 个参数 class_prior 输入先验概率，或者不输入第 3 个参数 class_

prior 让 MultinomialNB 自己从训练集样本来计算先验概率，此时的先验概率为 $p(y=c_k)=\frac{m_k}{m}$。其中，m 为训练集样本总数量，m_k 为输出第 k 类别的训练集样本数。

3. 伯努利朴素贝叶斯

BernoulliNB 应用于多重伯努利分布数据的朴素贝叶斯训练和分类算法，指有多个特征，但每个特征都假设是一个二元(Bernoulli，boolean)变量。因此，这类算法要求样本以二元值特征向量表示；如果样本含有其他类型的数据，一个 BernoulliNB 实例会将其二值化(取决于 binarize 参数)。

伯努利朴素贝叶斯的决策规则基于：

$$p(x_i \mid y)=p(i \mid y)x_i+(1-p(i \mid y))(1-x_i)$$

BernoulliNB 与多项式朴素贝叶斯的规则不同，伯努利朴素贝叶斯明确地惩罚类 y 中没有出现作为预测因子的特征 i，而多项式朴素贝叶斯只是简单地忽略没出现的特征。

【例 3-6】 使用 Iris 数据集做的一个使用正态朴素贝叶斯分类。

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.naive_bayes import GaussianNB
import matplotlib

plt.rcParams['font.sans-serif'] = ['SimHei']          #显示中文
#生成所有测试样本点
def make_meshgrid(x, y, h=.02):
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),np.arange(y_min, y_max, h))
    return xx, yy

#对测试样本进行预测,并显示
def plot_test_results(ax, clf, xx, yy, **params):
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    #画等高线
    ax.contourf(xx, yy, Z, **params)

#载入 Iris 数据集
iris = datasets.load_iris()
#只使用前面两个特征
X = iris.data[:, :2]
#样本标签值
y = iris.target
#创建并训练正态朴素贝叶斯分类器
clf = GaussianNB()
clf.fit(X,y)

title = ('高斯朴素贝叶斯分类器')
fig, ax = plt.subplots(figsize = (5, 5))
plt.subplots_adjust(wspace=0.4, hspace=0.4)

#分别取出两个特征
X0, X1 = X[:, 0], X[:, 1]
#生成所有测试样本点
xx, yy = make_meshgrid(X0, X1)
#显示测试样本的分类结果
```

```
plot_test_results(ax, clf, xx, yy, cmap = plt.cm.coolwarm, alpha = 0.8)
#显示训练样本
ax.scatter(X0, X1, c = y, cmap = plt.cm.coolwarm, s = 20, edgecolors = 'k')
ax.set_xlim(xx.min(), xx.max())
ax.set_ylim(yy.min(), yy.max())
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_xticks(())
ax.set_yticks(())
ax.set_title(title)
plt.show()
```

运行程序,效果如图 3-25 所示。

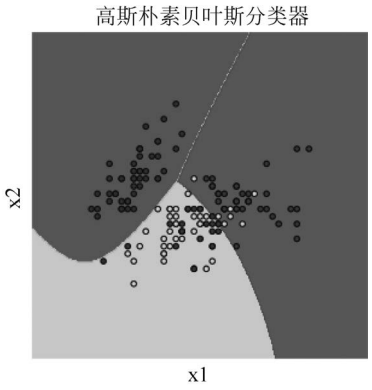


图 3-25　高斯朴素贝叶斯分类效果