第 3 章

机器学习的基础

在第 2 章已经知道如何用神经网络解决分类问题和回归问题，而且也看到了机器学习的核心难题：过拟合。为了解决过拟合问题本章会对机器学习进行介绍。

3.1 机器学习概述

机器学习是 20 多年前兴起的一门多领域交叉学科，它的理论主要是设计和分析一些让计算机可以自动“学习”的算法。机器学习算法是一类从数据中自动分析获得规律，并利用规律对未知数据进行预测的算法。

3.1.1 机器学习的历程

机器学习是人工智能研究较为年轻的分支，它的发展过程大体上可分为 4 个阶段。

- 第一阶段是在 20 世纪 50 年代中叶到 20 世纪 60 年代中叶，属于热烈时期。
- 第二阶段是在 20 世纪 60 年代中叶至 20 世纪 70 年代中叶，称为机器学习的冷静时期。
- 第三阶段是从 20 世纪 70 年代中叶至 20 世纪 80 年代中叶，称为复兴时期。
- 机器学习的第四个阶段也即最新阶段始于 1986 年。

机器学习进入新阶段重要表现在下列几方面。

(1) 机器学习已成为新的边缘学科并在高校形成一门课程。它综合应用心理学、生物学和神经生理学以及数学、自动化和计算机科学形成机器学习理论基础。

(2) 结合各种学习方法，取长补短的多种形式的集成学习系统研究正在兴起。特别是连接学习符号学习的耦合因可以更好地解决连续性信号处理中知识与技能的获取和求精问题而受到重视。

(3) 机器学习与人工智能各种基础问题的统一性观点正在形成。例如，学习与问题求解结合进行、知识表达便于学习的观点产生了通用智能系统 SOAR 的组块学习。类比学习与问题求解结合的基于案例方法已成为经验学习的重要方向。

(4) 各种学习方法的应用范围不断扩大,一部分已形成商品。归纳学习的知识获取工具已在诊断分类型专家系统中广泛使用。连接学习在声音、图文识别中占优势。分析学习已用于设计综合型专家系统。遗传算法与强化学习在工程控制中有较好的应用前景。与符号系统耦合的神经网络连接学习将在企业的智能管理与智能机器人运动规划中发挥作用。

(5) 与机器学习有关的学术活动空前活跃。国际上除每年一次的机器学习研讨会外,还有计算机学习理论会议以及遗传算法会议。

3.1.2 机器学习的4个分支

二分类问题、多分类问题和标量回归问题都是监督学习(supervised learning)的例子,其目标是学习训练输入与训练目标之间的关系。

监督学习只是冰山一角——机器学习是非常宽泛的领域,其子领域的划分非常复杂。机器学习算法大致可分4类,下面进行简单介绍。

1. 监督学习

监督学习是目前最常见的机器学习类型。给定一组样本(通常由人工标注),它可以学会将输入数据映射到已知目标(也叫标注,annotation)。一般来说,近年来广受关注的深度学习应用几乎都属于监督学习,如光学字符识别、语音识别、图像分类和语音翻译。

虽然监督学习主要包括分类和回归,但还有更多的奇特变体,主要包括如下几种。

(1) 序列生成(sequence generation)。给定一张图像,预测描述图像的文字。序列生成有时可以被重新表示为一系列分类问题,如反复预测序列中的单词或标记。

(2) 语法树预测(syntax tree prediction)。给定一个句子,预测其分解生成的语法树。

(3) 目标检测(object detection)。给定一张图像,在图中特定目标的周围画一个边界框。这个问题也可以表示为分类问题(给定多个候选边界框,对每个框内的目标进行分类)或分类与回归联合问题(用向量回归来预测边界框的坐标)。

(4) 图像分割(image segmentation)。给定一张图像,在特定物体上画一个像素级的掩膜(mask)。

2. 无监督学习

无监督学习(unsupervised learning)指在没有目标的情况下寻找输入数据的变换,其目的在于数据可视化、数据压缩、数据去噪或更好地理解数据中的相关性。无监督学习是数据分析的必备技能,在解决监督学习问题之前,为了更好地了解数据集,它通常是一个必要步骤。降维(dimensionality reduction)和聚类(clustering)都是众所周知的无监督学习方法。

3. 自监督学习

自监督学习(self-supervised learning)是监督学习的一个特例,它与众不同,值得单独归为一类。自监督学习是没有人工标签监督学习,可以将它看作没有人类参与的监督学习。标签仍然存在(用于监督学习过程),但它们是从输入数据中生成的,通常是使用启发式算法生成的。

例如，自编码器(autoencoder)是有名的自监督学习的例子，其生成的目标就是未经修改的输入。同样，给定视频中过去帧来预测下一帧，或者给定文本中前面的词来预测下一个词，都是自监督学习的例子(这两个例子也属于时序监督学习(temporally supervised learning)，即用未来的输入数据作为监督)。注意，监督学习、无监督学习和自监督学习之间的区别有时很模糊，这三个类别更像是没有明确界限的连续体。自监督学习可以被重新解释为监督学习或无监督学习，这取决于关注的是学习机制还是应用场景。

4. 强化学习

强化学习(reinforcement learning)是机器学习中的一个领域，是学习什么(即如何把当前的情景映射成动作)才能使得数值化的收益最大化，学习者不会被告知应该采取什么动作，而是必须自己通过尝试去发现哪些动作会产生最丰厚的收益。

强化学习同机器学习领域中的监督学习和无监督学习不同，监督学习是从外部监督者提供的带标注训练集中进行学习(任务驱动型)；无监督学习是一个典型的寻找未标注数据中隐含结构的过程(数据驱动型)。

强化学习是与两者并列的第三种机器学习范式，强化学习带来了一个独有的挑战——探索与利用之间的折中权衡，智能体必须利用已有的经验来获得收益，同时也要进行探索，使得未来可以获得更好的动作选择空间(即从错误中学习)。

强化学习的主要角色是智能体和环境，环境是智能体存在和互动的世界。智能体在每一步的交互中，都会获得对于所处环境状态的观察(有可能只是一部分)，然后决定下一步要执行的动作。环境会因为智能体对它的动作而改变，也可能自己改变。

智能体也会从环境中感知到奖励信号，一个表明当前状态好坏的数字。智能体的目标是最大化累计奖励，也就是回报。强化学习就是智能体通过学习来完成目标的方法。

3.1.3　机器学习的步骤

机器学习是一门人工智能的学科，该领域的主要研究对象是人工智能，特别是如何在经验中改善具体算法的性能。它的一般步骤如图 3-1 所示。

(1) 收集数据。

收集到的数据的质量和数量将直接决定预测模型是否能够建好。需要将收集的数据去重、标准化、修正错误等，保存成数据库文件或者 csv 格式文件，为下一步数据的加载做准备。

(2) 分析数据。

分析数据主要是数据发现，如对每列的最大值、最小值、平均值、方差、中位数、三分位数、四分位数、某些特定值(如零值)所占比例或者分布规律等都要有一个大致的了解。另外要确定自变量(x_1，x_2，…，x_n)和因变量 y，找出因变量和自变量的相关性，确定相关系数。

特征的好坏很大程度上决定了分类器的效果。将上面确定的自变量进行筛选，可以手工选择或者模型选择，选择合适的特征，然后对变量进行命名以便更好地标记。命名文件要存下来，在预测阶段会用到。

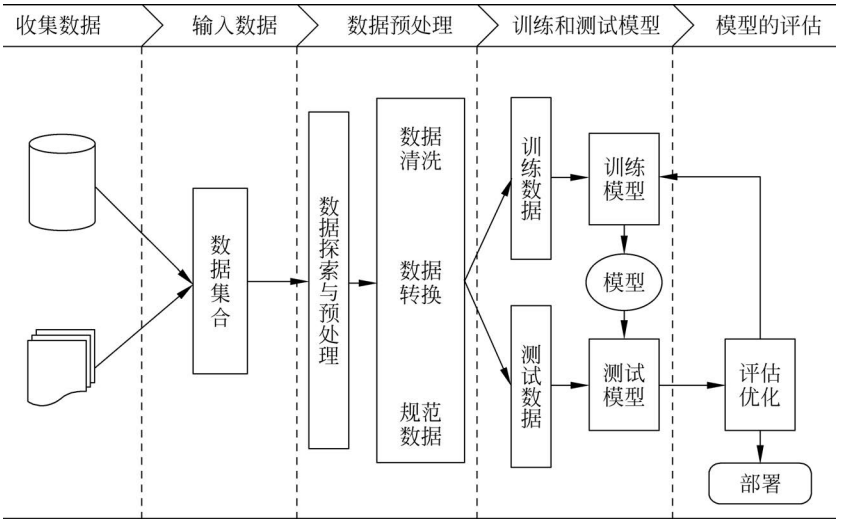

图 3-1　机器学习的步骤

向量化是对特征提取结果的再加工，目的是增强特征的表示能力，防止模型过于复杂和学习困难，如对连续的特征值进行离散化，标签值映射成枚举值，用数字进行标识。这一阶段将产生一个很重要的文件：标签和枚举值对应关系，在预测阶段同样会用到。

（3）数据预处理。

需要将数据分为两部分。用于训练模型的第一部分将是数据集的大部分。第二部分将用于评估我们训练有素的模型的表现。通常以 8∶2 或者 7∶3 进行数据划分，不能直接使用训练数据来进行评估，因为模型只能记住“问题”。

（4）训练和测试模型。

进行模型训练之前，要确定合适的算法，如线性回归、决策树、随机森林、逻辑回归、梯度提升、SVM 等。选择算法时最佳方法是测试各种不同的算法，然后通过交叉验证选择最好的一个。但是，如果只是为问题寻找一个“足够好”的算法，或者一个起点，也有一些较好的一般准则，如果训练集很小，那么高偏差/低方差分类器（如朴素贝叶斯分类器）要优于低偏差/高方差分类器（如 K 近邻分类器），因为后者容易过拟合。然而，随着训练集的增大，低偏差/高方差分类器将开始胜出（它们具有较低的渐近误差），因为高偏差分类器不足以提供准确的模型。

（5）模型的评估。

训练完成之后，通过拆分出来的训练的数据来对模型进行评估、通过真实数据和预测数据进行对比来判定模型的好坏。模型评估常见的 5 个方法为：混淆矩阵、提升图 &. 洛伦兹图、基尼系数、KS 曲线、ROC 曲线。混淆矩阵不能作为评估模型的唯一标准，混淆矩阵是模型其他指标的基础。完成评估后，如果想进一步改善训练，可以通过调整模型的参数来实现，然后重复训练和评估的过程。

模型训练完之后，要整理出 4 类文件，确保模型能够正确运行，这 4 类文件分别为模型文件、标签编码文件、元数据文件（算法、参数和结果）、变量文件（自变量名称列表、因变量名称列表）。

完成模型后就可以在互联网上发布。

3.2　过拟合和欠拟合

机器学习中一个重要的话题便是模型的泛化能力，泛化能力强的模型才是好模型。对于训练好的模型，若在训练集表现差，在测试集表现同样会很差，这可能是欠拟合导致的。欠拟合是指模型拟合程度不高，数据距离拟合曲线较远，或指模型没有很好地捕捉到数据特征，不能够很好地拟合数据。

为了防止模型从训练数据中记住错误或无关紧要的模式，最优解决方法是获取更多的训练数据。模型的训练数据越多，泛化能力自然也越好。如果无法获取更多数据，次优解决方法是调节模型允许存储的信息量，或对模型允许存储的信息加以约束。如果一个网络只能记住几个模式，那么优化过程会迫使模型集中学习最重要的模式，这样更可能得到良好的泛化。

3.2.1　减小模型大小

防止过拟合的最简单的方法就是减小模型大小，即减小模型中可学习参数的个数（这由层数和每层的单元个数决定）。在深度学习中，模型可学习参数的个数通常被称为模型的容量（capacity）。从直观上来看，参数更多的模型拥有更大的记忆容量（memorization capacity），因此能够在训练样本和目标之间轻松地学会完美的字典式映射，这种映射没有任何泛化能力。例如，拥有 500 000 个二进制参数的模型，能够轻松学会 MNIST 训练集中所有数字对应的类别——只需让 50 000 个数字每个都对应 10 个二进制参数。但这种模型对于新数字样本的分类毫无用处。

与此相反，如果网络的记忆资源有限，则无法轻松学会这种映射。因此，为了让损失最小化，网络必须学会对目标具有很强的预测能力的压缩表示，这也正是人们感兴趣的数据表示。需要记住的是，使用的模型应该具有足够多的参数，以防欠拟合，即模型应避免记忆资源不足。在容量过大与容量不足之间要找到一个折中。要找到合适的模型大小，一般的工作流程是开始时选择相对较少的层和参数，然后逐渐增加层的大小或增加新层，直到这种增加对验证损失的影响变得很小。

【例 3-1】 在电影评价上尝试减小模型大小。

```
from keras.datasets import imdb
import numpy as np

'''原始模型'''
from keras import models
from keras import layers
original_model = models.Sequential()
original_model.add(layers.Dense(16, activation = 'ReLU', input_shape = (10000,)))
original_model.add(layers.Dense(16, activation = 'ReLU'))
original_model.add(layers.Dense(1, activation = 'sigmoid'))
```

```
'''容量更小的模型'''
smaller_model = models.Sequential()
smaller_model.add(layers.Dense(4, activation = 'ReLU', input_shape = (10000,)))
smaller_model.add(layers.Dense(4, activation = 'ReLU'))
smaller_model.add(layers.Dense(1, activation = 'sigmoid'))

smaller_model.compile(optimizer = 'rmsprop',
                      loss = 'binary_crossentropy',
                      metrics = ['acc'])
```

图 3-2 比较了原始模型与更小模型的验证损失。圆点是更小模型的验证损失值，十字是原始模型的验证损失值。

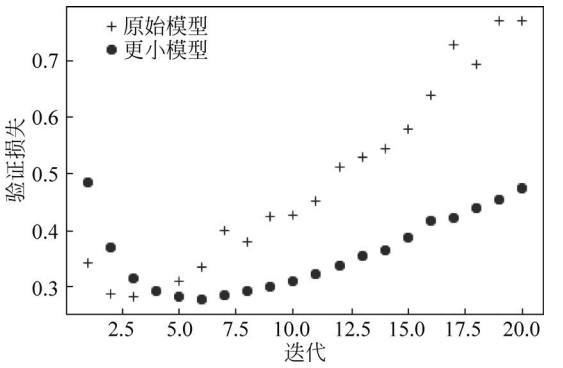

图 3-2　换用更小的模型验证损失效果

由图 3-2 可见，更小的模型开始过拟合的时间要易于原始模型（前 6 轮后开始过拟合，而后者 4 轮后开始），且开始过拟合后，它的速度也更慢。

下面再向这个基准中添加一个容量更大的模型（容量远大于问题所需）。

```
bigger_model = models.Sequential()
bigger_model.add(layers.Dense(512, activation = 'ReLU', input_shape = (10000,)))
bigger_model.add(layers.Dense(512, activation = 'ReLU'))
bigger_model.add(layers.Dense(1, activation = 'sigmoid'))
```

图 3-3 显示了更大模型与原始模型的性能对比。圆点是更大模型的验证损失值，十字是原始模型的验证损失值。更大模型只进行一轮就开始过拟合，过拟合也更严重，其验证损失的波动也更大。

图 3-4 同时给出了这两个模型的训练损失。从图中可见，更大模型的训练损失很快就接近于零。模型的容量越大，它拟合训练数据（即得到很小的训练损失）的速度就越快，但也更容易过拟合（导致训练损失和验证损失有很大差异）。

3.2.2　添加权重正则化

一种常见的降低过拟合的方法就是强制让模型权重只能取较小的值，从而限制模型的复杂度，这使得权重值的分布更加规则（regular），这种方法叫作权重正则化（weight regularization），其实现方法是向网络损失函数中添加与较大权重值相关的成本（cost）。

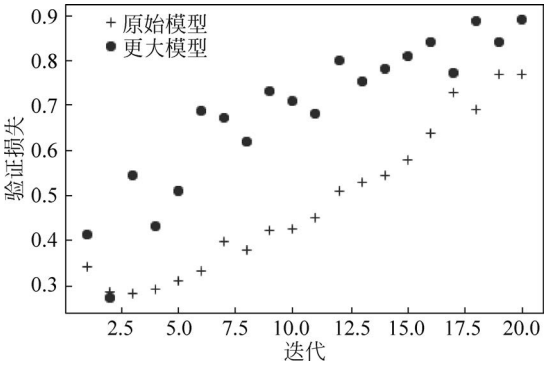

图 3-3　换用更大的模型验证损失效果

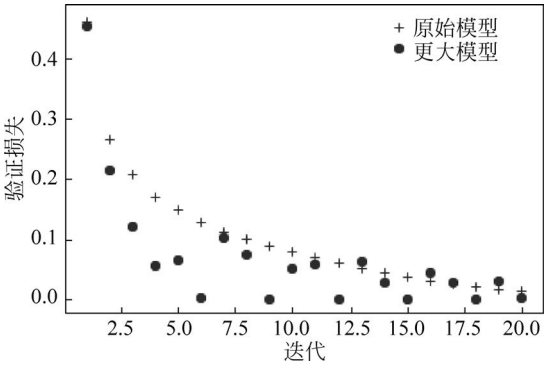

图 3-4　同时对比更小与更大模型的训练损失效果

这个成本有两种形式。

(1) L1 正则化(L1 regularization)：添加的成本与权重系数的绝对值(权重的 L1 范数(norm))成正比。

(2) L2 正则化(L2 regularization)：添加的成本与权重系数的平方(权重的 L2 范数)成正比。神经网络的 L2 正则化也叫权重衰减(weight decay)。权重衰减与 L2 正则化在数学上是完全相同的。

在 Keras 中，添加权重正则化的方法是向层传递权重正则化项实例(weigh regularizer instance)作为关键字参数。

【例 3-2】 将向电影评论分类网络中添加 L2 正则化。

```
'''向模型添加 L2 正则化'''
from keras import regularizers

l2_model = models.Sequential()
l2_model.add(layers.Dense(16, kernel_regularizer = regularizers.l2(0.001),
                          activation = 'ReLU', input_shape = (10000,)))
l2_model.add(layers.Dense(16, kernel_regularizer = regularizers.l2(0.001),
                          activation = 'ReLU'))
l2_model.add(layers.Dense(1, activation = 'sigmoid'))
```

l2(0.001)的作用为该层权重矩阵的每个系数都会使网络总损失增加 0.001 * weight_coefficient_value。注意,因为这个惩罚项只在训练时添加,所以这个网络的训练损失会比测试损失大很多。

图 3-5 显示了 L2 正则化惩罚的影响。如图中所示,即使两个模型的参数个数相同,具有 L2 正则化的模型(圆点)比原始模型(十字)更不容易过拟合。

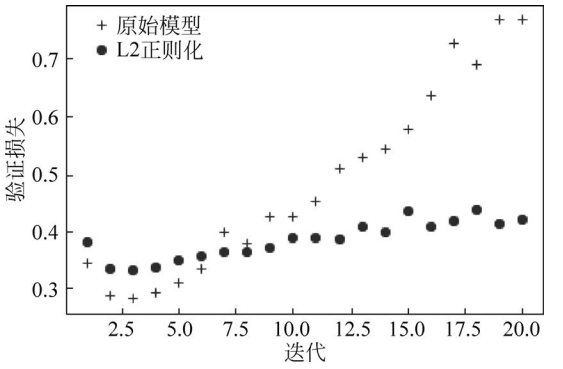

图 3-5 L2 正则化惩罚的影响

还可以用 Keras 中以下权重正则化项来代替 L2 正则化。

```
from keras import regularizers

# L1 正则化
regularizers.l1(0.001)
#同时进行 L1 与 L2 正则化
regularizers.l1_l2(l1 = 0.001, l2 = 0.001)
```

3.2.3 添加 dropout 正则化

dropout 是神经网络最有效也最常用的正则化方法之一。对某一层使用 dropout,就是在训练过程中随机将该层的一些输出特征舍弃(设置为 0)。假设在训练过程中,某一层对给定输入样本的返回值应该是向量(0.2,0.5,1.3,0.8,1.1),使用 dropout 后,这个向量会有几个元素随机地变成 0,如(0,0.5,1.3,0,1.1),dropout 比率(dropout rate)是被设为 0 的特征所占的比例,通常在 0.2～0.5 范围内。测试时没有单元被舍弃,而该层的输出值需要按 dropout 比率缩小,因为这时比训练时有更多的单元被激活,需要加以平衡。

假设有一个包含某层输出的 NumPy 矩阵 layer_output,其形状为[batch_size, features],训练时,随机将矩阵中一部分值设为 0:

```
#训练时,舍弃 50 % 的输出单元
layer_output * = np.randint(0, high = 2, size = layer_output.shape)
```

测试时,将输出按 dropout 比率缩小。此处乘以 0.5(前面舍弃了一半的单元)。

```
#测试时
ayer_output * = 0.5
```

注意，为了实现这一过程，还可以让两个运算都在训练时进行，而测试时输出保持不变。这通常也是实践中的实现方式。

```
# 训练时
layer_output *= np.randint(0, high=2, size=layer_output.shape)
# 注意，是成比例放大而不是成比例缩小
layer_output /= 0.5
```

在 Keras 中，可以通过 dropout 层向网络中引入 dropout，dropout 将被应用于前面一层的输出。

```
model.add(layers.Dropout(0.5))
```

向 IMDB 网络中添加两个 dropout 层，观察它们降低过拟合的效果。

```
dpt_model = models.Sequential()
dpt_model.add(layers.Dense(16, activation='ReLU', input_shape=(10000,)))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(16, activation='ReLU'))
dpt_model.add(layers.Dropout(0.5))
dpt_model.add(layers.Dense(1, activation='sigmoid'))
```

图 3-6 给出了结果的图示，可再次看到，这种方法的性能相比原始模型有明显提高。

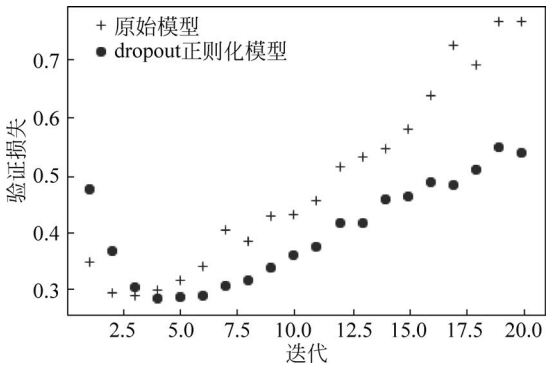

图 3-6　dropout 对验证损失的影响

因此，可得防止神经网络过拟合的常用方法主要如下。

(1) 获取更多的训练数据。

(2) 减小网络容量。

(3) 添加权重正则化。

(4) 添加 dropout 正则化。

3.3　监督学习

监督学习算法可以分为以下几类。

(1) 线性模型：例如线性回归、逻辑回归等。

(2) 基于核函数的模型：例如支持向量机(SVM)。

(3) 决策树和基于集成的方法：例如随即森林、Adaboost 等。

(4) 人工神经网络和深度学习：例如全连接神经网络、卷积神经网络(CNN)、递归神经网络(RNN)及其变种模型。

3.3.1 线性模型

在监督学习中，线性模型的代表有线性回归、逻辑回归等。

1. 线性回归

下面是一组用于回归的方法，其中目标值 $\boldsymbol{y}$ 是输入变量 $\boldsymbol{x}$ 的线性组合。在数学概念中，如果 $\hat{y}$ 预测值，即

$$\hat{y}(\boldsymbol{w},\boldsymbol{x}) = w_0 + w_1 x_1 + \cdots + w_p x_p$$

在整个模块中，定义向量 $\boldsymbol{w} = (w_1, w_2, \cdots, w_p)$ 作为 coef_，定义向量 $\boldsymbol{x} = (x_1, x_2, \cdots, x_p)$ 作为 intercept_。

2. 最小二乘法

最小二乘法(least square method)拟合一个带有系数向量 $\boldsymbol{w} = (w_1, w_2, \cdots, w_p)$ 的线性模型，使得数据集实际观测数据和预测数据(估计值)之间的残差平方和最小。其数学表达式为

$$\min_{\boldsymbol{w}} \| \boldsymbol{X}\boldsymbol{w} - \boldsymbol{y} \|_2^2$$

最小二乘会调用 fit 方法来拟合 $\boldsymbol{X}$ 和 $\boldsymbol{Y}$，并且将线性模型的系数 $\boldsymbol{w}$ 存储在其成员变量 coef_中：

```
>>> from sklearn import linear_model
>>> reg = linear_model.LinearRegression()
>>> reg.fit ([[0, 0], [1, 1], [2, 2]], [0, 1, 2])
LinearRegression(copy_X = True, fit_intercept = True,
                    n_jobs = 1, normalize = False)
>>> reg.coef_
array([ 0.5, 0.5])
```

对于最小二乘的系数估计问题，其依赖于模型各项的相互独立性。当各项是相关的，且设计矩阵 $\boldsymbol{x}$ 的各列近似线性相关，那么，设计矩阵会趋向于奇异矩阵，这种特性导致最小二乘估计对于随机误差非常敏感，可能产生很大的方差。

【例 3-3】 最小二乘拟合效果。

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets, linear_model
from sklearn.metrics import mean_squared_error, r2_score
#加载糖尿病数据集
diabetes_X, diabetes_y = datasets.load_diabetes(return_X_y = True) #Python 自带的数据集

#仅使用一个函数
diabetes_X = diabetes_X[:, np.newaxis, 2]
```

```
#将数据分割为训练/测试集
diabetes_X_train = diabetes_X[:-20]
diabetes_X_test = diabetes_X[-20:]
#将目标划分为训练/测试集
diabetes_y_train = diabetes_y[:-20]
diabetes_y_test = diabetes_y[-20:]
#创建线性回归对象
regr = linear_model.LinearRegression()
#使用训练集来训练模型
regr.fit(diabetes_X_train, diabetes_y_train)
#使用测试集进行预测
diabetes_y_pred = regr.predict(diabetes_X_test)
#系数
print("系数:\n", regr.coef_)
#均方误差
print("均方误差: %.2f" % mean_squared_error(diabetes_y_test, diabetes_y_pred))
#决定系数:1是完美的预测
print("完美的预测: %.2f" % r2_score(diabetes_y_test, diabetes_y_pred))

#绘图
plt.scatter(diabetes_X_test, diabetes_y_test, color = "black")
plt.plot(diabetes_X_test, diabetes_y_pred, color = "blue", linewidth = 3)
plt.xticks(())
plt.yticks(())
plt.show() #显示图形
```

运行程序,输出如下,拟合效果如图 3-7 所示。

```
系数:
 [938.23786125]
均方误差: 2548.07
完美的预测: 0.47
```

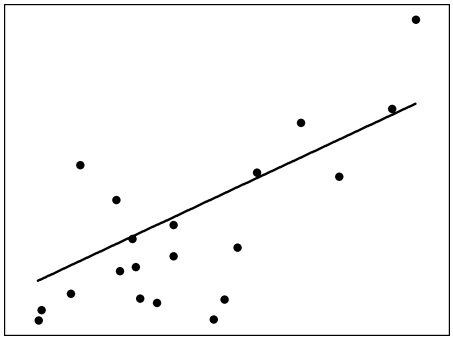

图 3-7　最小二乘拟合效果

3. 岭回归

岭回归(Ridge Regression,RR)通过对系数的大小施加惩罚来解决最小二乘法的一些问题。岭回归最小化的是带惩罚项的残差平方和。

$$\min_{w} \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|_2^2 + \alpha \|\boldsymbol{w}\|_2^2$$

其中，$\alpha \geqslant 0$ 是控制系数收缩量的复杂性参数，α 的值越大，收缩量越大，模型对共线性的健壮性也更强。

与其他线性模型一样，岭回归用 fit 方法完成拟合，并将模型系数 $\boldsymbol{w}$ 存储在其 coef_ 成员中：

```
>>> from sklearn import linear_model
>>> reg = linear_model.Ridge (alpha = .5)
>>> reg.fit ([[0, 0], [0, 0], [1, 1]], [0, .1, 1])
Ridge(alpha = 0.5, copy_X = True, fit_intercept = True, max_iter = None,
 normalize = False, random_state = None, solver = 'auto', tol = 0.001)
>>> reg.coef_
array([0.34545455, 0.34545455])
>>> reg.intercept_
0.13636...
```

【例 3-4】 岭系数对回归系数的影响。

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

plt.rcParams['font.sans-serif'] = ['SimHei']      #显示中文
plt.rcParams['axes.unicode_minus'] = False        #显示负号

# X是10×10的希尔伯特矩阵
X = 1.0 / (np.arange(1, 11) + np.arange(0, 10)[:, np.newaxis])
y = np.ones(10)

'''计算路径'''
n_alphas = 200
alphas = np.logspace(-10, -2, n_alphas)
coefs = []
for a in alphas:
    ridge = linear_model.Ridge(alpha=a, fit_intercept=False)
    ridge.fit(X, y)
    coefs.append(ridge.coef_)
'''绘制结果'''
ax = plt.gca()
ax.plot(alphas, coefs)
ax.set_xscale("log")
ax.set_xlim(ax.get_xlim()[::-1])                  #reverse axis
plt.xlabel("alpha")
plt.ylabel("权重")
plt.title("岭系数作为正则化的函数")
plt.axis("tight")
plt.show()
```

运行程序，效果如图 3-8 所示。

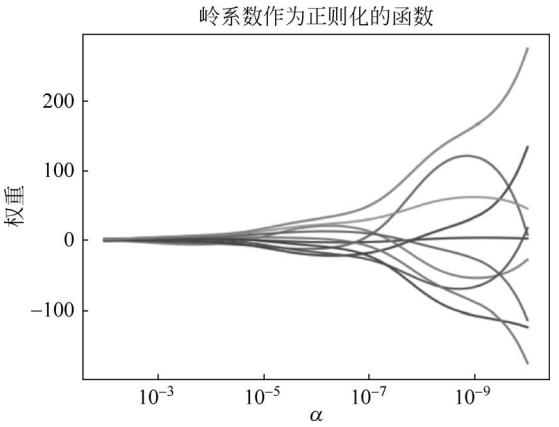

图 3-8　岭回归效果

4. Lasso

Lasso(套索)是拟合稀疏系数的线性模型。它在一些情况下是有用的,因为它倾向于使用具有较少参数值的情况,有效地减少给定解决方案所依赖变量的数量。因此,Lasso 及其变体是压缩感知领域的基础。在一定条件下,它可以恢复一组非零权重的精确集。

在数学公式表达上,它由一个带有 λ_1 先验的正则项的线性模型组成。其最小化的目标函数是

$$\min_{w} \frac{1}{2n_{\text{samples}}} \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|_2^2 + \alpha \|\boldsymbol{w}\|_1$$

Lasso 估计解决了加上罚项 $\alpha\|\boldsymbol{w}\|_1$ 的最小二乘法的最小化,其中,α 是一个常数,$\|\boldsymbol{w}\|_1$ 是参数向量的 λ_1 范数。

Lasso 类的实现使用了坐标下降算法(coordinate descent)来拟合系数。

```
>>> from sklearn import linear_model
>>> reg = linear_model.Lasso(alpha = 0.1)
>>> reg.fit([[0, 0], [1, 1]], [0, 1])
Lasso(alpha = 0.1, copy_X = True, fit_intercept = True, max_iter = 1000,
       normalize = False, positive = False, precompute = False,
       random_state = None, selection = 'cyclic', tol = 0.0001,
       warm_start = False)
>>> reg.predict([[1, 1]])
array([0.8])
```

对于较简单的任务,有用的是函数 lasso_path。它能够通过搜索所有可能路径上的值来计算系数。

【例 3-5】 演示一个具有正则化参数的固定值的 Lasso。

```
from time import time
from sklearn.linear_model import Lasso
from sklearn.metrics import r2_score
```

```
t0 = time()
lasso = Lasso(alpha = 0.14).fit(X_train, y_train)
print(f"Lasso fit done in {(time() - t0): .3f}s")

y_pred_lasso = lasso.predict(X_test)
r2_score_lasso = r2_score(y_test, y_pred_lasso)
print(f"Lasso r^2 on test data : {r2_score_lasso: .3f}")
```

运行程序,输出如下:

```
Lasso fit done in 0.001s
Lasso r^2 on test data : 0.480
```

在实践中,应该通过传递时间序列分割交叉验证策略来选择最优参数 alpha。为了保持实例的简单和快速执行,在实例直接设置了 alpha 的最优值。

5. 弹性网络

弹性网络(elastic network)是一种使用 L1 和 L2 范数作为先验正则项训练的线性回归模型。这种组合允许拟合到一个只有少量参数是非零稀疏的模型,就像 Lasso 一样,但是它仍然保持了一些类似于岭回归的正则性质。可利用 l1_ratio 参数控制 L1 和 L2 范数的凸组合。

弹性网络在很多特征互相联系的情况下是非常有用的。Lasso 很可能只随机考虑这些特征中的一个,而弹性网络更倾向于选择两个。

在实践中,Lasso 和岭回归之间权衡的一个优势是它允许在循环过程中继承岭回归的稳定性。

此处,最小化的目标函数为

$$\min_{w} \frac{1}{2n_{\text{samples}}} \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|_2^2 + \alpha \|\boldsymbol{w}\|_1 + \frac{\alpha(1-\rho)}{2} \|\boldsymbol{w}\|_2^2$$

ElasticNetCV 类可以通过交叉验证来设置参数 alpha(α)和 lr_ratio(ρ)。

【例 3-6】 Lasso 和弹性网络。

```
from itertools import cycle
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets
from sklearn.linear_model import enet_path, lasso_path

X, y = datasets.load_diabetes(return_X_y = True)
X /= X.std(axis = 0)      # Standardize data (easier to set the l1_ratio parameter)
#完整路径
eps = 5e-3                          #它越小,路径就越长
print("使用套索计算正则化路径...")
alphas_lasso, coefs_lasso, _ = lasso_path(X, y, eps = eps)

print("使用正套索计算正则化路径...")
alphas_positive_lasso, coefs_positive_lasso, _ = lasso_path(
```

```
    X, y, eps = eps, positive = True
)
print("使用弹性网络计算正则化路径...")
alphas_enet, coefs_enet, _ = enet_path(X, y, eps = eps, l1_ratio = 0.8)

print("使用正弹性网络计算正则化路径...")
alphas_positive_enet, coefs_positive_enet, _ = enet_path(
    X, y, eps = eps, l1_ratio = 0.8, positive = True
)
#显示结果
plt.figure(1)
colors = cycle(["b", "r", "g", "c", "k"])
neg_log_alphas_lasso = -np.log10(alphas_lasso)
neg_log_alphas_enet = -np.log10(alphas_enet)
for coef_l, coef_e, c in zip(coefs_lasso, coefs_enet, colors):
    l1 = plt.plot(neg_log_alphas_lasso, coef_l, c = c)
    l2 = plt.plot(neg_log_alphas_enet, coef_e, linestyle = "--", c = c)
#效果如图 3-9 所示
plt.xlabel("-Log(alpha)")
plt.ylabel("系数")
plt.title("套索和弹性网络")
plt.legend((l1[-1], l2[-1]), ("Lasso", "Elastic-Net"), loc = "lower left")
plt.axis("tight")
plt.figure(2)                    #效果如图 3-10 所示
neg_log_alphas_positive_lasso = -np.log10(alphas_positive_lasso)
for coef_l, coef_pl, c in zip(coefs_lasso, coefs_positive_lasso, colors):
    l1 = plt.plot(neg_log_alphas_lasso, coef_l, c = c)
    l2 = plt.plot(neg_log_alphas_positive_lasso, coef_pl, linestyle = "--", c = c)

plt.xlabel("-Log(alpha)")
plt.ylabel("系数")
plt.title("套索和正套索")
plt.legend((l1[-1], l2[-1]), ("Lasso", "正 Lasso"), loc = "lower left")
plt.axis("tight")
plt.figure(3)                    #效果如图 3-11 所示
neg_log_alphas_positive_enet = -np.log10(alphas_positive_enet)
for coef_e, coef_pe, c in zip(coefs_enet, coefs_positive_enet, colors):
    l1 = plt.plot(neg_log_alphas_enet, coef_e, c = c)
    l2 = plt.plot(neg_log_alphas_positive_enet, coef_pe, linestyle = "--", c = c)

plt.xlabel("-Log(alpha)")
plt.ylabel("系数")
plt.title("弹性网络和正弹性网络")
plt.legend((l1[-1], l2[-1]), ("弹性网络", "正弹性网络"), loc = "lower left")
plt.axis("tight")
plt.show()
```

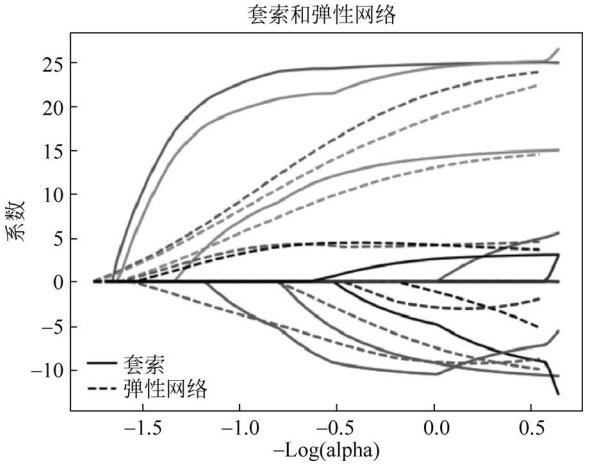

图 3-9　套索和弹性网络效果

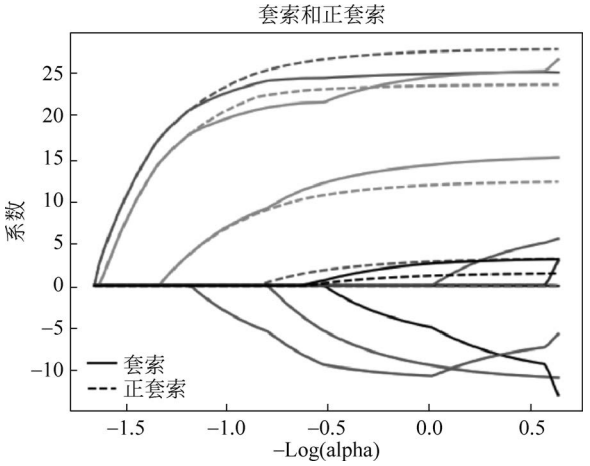

图 3-10　套索与正套索

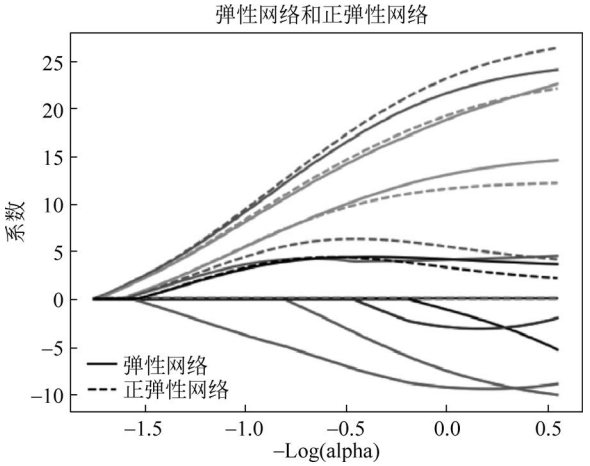

图 3-11　弹性网络与正弹性网络

同时，程序输出如下：

```
使用套索计算正则化路径...
使用正套索计算正则化路径...
使用弹性网络计算正则化路径...
使用正弹性网络计算正则化路径...
```

6. 贝叶斯回归

贝叶斯回归可以用于在预估阶段的参数正则化：正则化参数的选择不是通过人为的选择，而是通过手动调节数据值来实现的。

上述过程可以通过引入无信息先验模型中的超参数来完成。在岭回归中使用的 λ_2 正则项相当于在 w 为高斯先验条件且此先验的精确度为 λ^{-1} 时，求最大后验估计。在这里，没有手工调参数 λ，而是让它作为一个变量，通过在数据中估计得到。

为了得到一个全概率模型，输出 y 也被认为是关于 X_w 的高斯分布。

$$p(y \mid X_w, w, \alpha) = N(y \mid X_w, \alpha)$$

其中，α 作为一个变量，通过在数据中估计得到；$N()$为高斯分布。

1）贝叶斯岭回归

贝叶斯岭(Bayesian ridge)回归利用概率模型估算了上述回归问题，其先验参数 w 是由以下球面高斯公式得出的：

$$p(w \mid \lambda) = N(w \mid 0, \lambda^{-1}\boldsymbol{I}_p)$$

先验参数 α 和 λ 一般服从 γ 分布，这个分布与高斯成共轭先验关系。得到的模型一般称为贝叶斯岭回归，并且这个与传统的岭回归非常相似。

参数 w、α 和 λ 是在模型拟合时一起被估算出来的，其中参数 α 和 λ 通过最大似然估计得到。

【例 3-7】 贝叶斯岭回归用来解决回归问题。

```
>>> from sklearn import linear_model
>>> X = [[0., 0.], [1., 1.], [2., 2.], [3., 3.]]
>>> Y = [0., 1., 2., 3.]
>>> reg = linear_model.BayesianRidge()
>>> reg.fit(X, Y)
BayesianRidge(alpha_1 = 1e - 06, alpha_2 = 1e - 06, compute_score = False, copy_X = True,
fit_intercept = True, lambda_1 = 1e - 06, lambda_2 = 1e - 06, n_iter = 300,
normalize = False, tol = 0.001, verbose = False)
```

在模型训练完成后，可以用来预测新值：

```
>>> reg.predict ([[1, 0.]])
array([0.50000013])
```

权值 w 可以被这样访问：

```
>>> reg.coef_
array([0.49999993, 0.49999993])
```

由于贝叶斯框架的缘故，权值与最小二乘法产生的不同。但是，贝叶斯岭回归对病态问题(ill-posed)的健壮性要更好。

2) 主动相关决策理论

主动相关决策理论(Automatic Relevance Determination，ARD)和贝叶斯岭回归非常相似，但是会导致一个更加稀疏的权重 w。

ARD 提出了一个不同的 w 的先验假设。具体来说，就是弱化了高斯分布为球形的假设。它采用 w 分布是与轴平行的椭圆高斯分布。也就是说，每个权值 w_i 是由一个中心在 0 点、精度为 λ_i 的高斯分布中采样得到的。

$$p(w \mid \lambda) = N(w \mid 0, \boldsymbol{A}^{-1})$$

并且 $\mathrm{diag}(\boldsymbol{A}) = \lambda = \{\lambda_1, \lambda_2, \cdots, \lambda_p\}$。

与贝叶斯岭回归不同，每个 w_i 都有一个标准 λ_i。所有 λ_i 的先验分布由超参数 λ_1、λ_2 确定的相同的 γ 分布确定。

ARD 也被称为稀疏贝叶斯学习或相关向量机。

【例 3-8】 带有多项式特征展开式的贝叶斯回归。

```
#实例创建了一个目标,它是输入特征的非线性函数。其中加入了遵循标准均匀分布的噪声
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
#生成合成数据集
rng = np.random.RandomState(0)
n_samples = 110

#对数据进行排序,以便以后更容易绘制
X = np.sort(-10 * rng.rand(n_samples) + 10)
noise = rng.normal(0, 1, n_samples) * 1.35
y = np.sqrt(X) * np.sin(X) + noise
full_data = pd.DataFrame({"input_feature": X, "target": y})
X = X.reshape((-1, 1))

#推断
X_plot = np.linspace(10, 10.4, 10)
y_plot = np.sqrt(X_plot) * np.sin(X_plot)
X_plot = np.concatenate((X, X_plot.reshape((-1, 1))))
y_plot = np.concatenate((y - noise, y_plot))
'''拟合回归
尝试用一个 10 阶的多项式进行过拟合.尽管贝叶斯线性模型正则化了多项式系数的大小,但多项式特征不应该引入附加偏差特征.通过设置 return_std = True,贝叶斯回归变量返回模型参数的后验分布的标准差'''
ard_poly = make_pipeline(
    PolynomialFeatures(degree = 10, include_bias = False),
    StandardScaler(),
    ARDRegression(),
).fit(X, y)
brr_poly = make_pipeline(
    PolynomialFeatures(degree = 10, include_bias = False),
```

```
    StandardScaler(),
    BayesianRidge(),
).fit(X, y)

y_ard, y_ard_std = ard_poly.predict(X_plot, return_std = True)
y_brr, y_brr_std = brr_poly.predict(X_plot, return_std = True)
#用分数的标准数据误差绘制多项式回归图
ax = sns.scatterplot(
    data = full_data, x = "输入特征", y = "目标", color = "black", alpha = 0.75
)
ax.plot(X_plot, y_plot, color = "black", label = "真实值")
ax.plot(X_plot, y_brr, color = "red", label = "具有多项式特征的贝叶斯岭回归")
ax.plot(X_plot, y_ard, color = "navy", label = "具有多项式特征的 ARD")
ax.fill_between(
    X_plot.ravel(),
    y_ard - y_ard_std,
    y_ard + y_ard_std,
    color = "navy",
    alpha = 0.3,
)
ax.fill_between(
    X_plot.ravel(),
    y_brr - y_brr_std,
    y_brr + y_brr_std,
    color = "red",
    alpha = 0.3,
)
ax.legend()
_ = ax.set_title("非线性特征的多项式拟合")
```

运行程序，效果如图 3-12 所示。

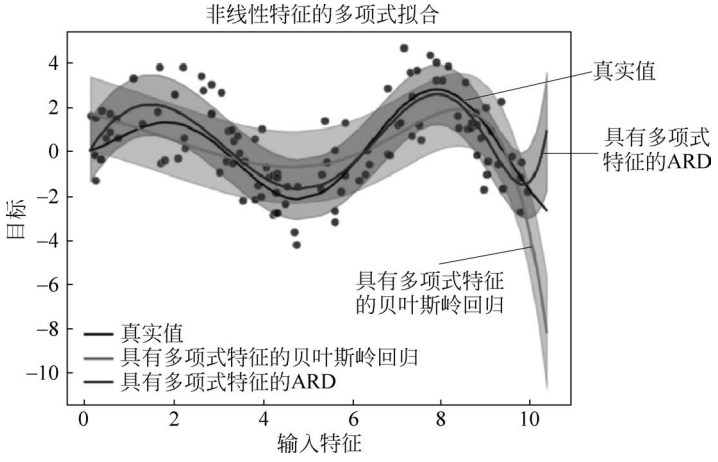

图 3-12　贝叶斯回归效果

3.3.2　逻辑回归

逻辑(logistic)回归虽然名字里有“回归”二字，但实际上是解决分类问题的一类线性

模型。logistic 回归又被称作 logit 回归、最大熵分类(maximum-entropy classification，MaxEnt)，或对数线性分类器(log-linear classifier)。该模型利用 logistic 函数将单次试验(single trial)的可能结果输出为概率。

scikit-learn 中逻辑回归在 LogisticRegression 类中实现了二分类(binary)、一对多分类(one-vs-rest)及多项式逻辑回归，并带有可选的 L1 和 L2 正则化。

作为优化问题，带 L2 惩罚项的二分类逻辑回归要最小化以下代价函数：

$$\min_{\boldsymbol{w},c} \frac{1}{2}\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w} + C\sum_{i=1}^{n}\log(\exp(-y_i(\boldsymbol{X}_i^{\mathrm{T}}\boldsymbol{w}+c))+1)$$

类似地，带 L1 正则化的逻辑回归解决的是如下优化问题：

$$\min_{\boldsymbol{w},c} \|\boldsymbol{w}\|_1 + C\sum_{i=1}^{n}\log(\exp(-y_i(\boldsymbol{X}_i^{\mathrm{T}}\boldsymbol{w}+c))+1)$$

弹性网络正则化是 L1 和 L2 正则化的组合，使如下代价函数最小：

$$\min_{\boldsymbol{w},c} \frac{1-\rho}{2}\boldsymbol{w}^{\mathrm{T}}\boldsymbol{w} + \rho\|\boldsymbol{w}\|_1 + C\sum_{i=1}^{n}\log(\exp(-y_i(\boldsymbol{X}_i^{\mathrm{T}}\boldsymbol{w}+c))+1)$$

其中，ρ 控制 L1 与 L2 正则化的强度(对应于 l1_ratio 参数)。

【例 3-9】 使用多项式逻辑回归和 L1 正则化进行 MNIST 数据集的分类。

```
#在 MNIST 数据集分类任务的一个子集上拟合了一个具有 L1 惩罚的多项式逻辑回归
import time
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.utils import check_random_state

#拒绝,以更快地收敛
t0 = time.time()
train_samples = 5000

#载入数据
X, y = fetch_openml(
    "mnist_784", version=1, return_X_y=True, as_frame=False, parser="pandas"
)

random_state = check_random_state(0)
permutation = random_state.permutation(X.shape[0])
X = X[permutation]
y = y[permutation]
X = X.reshape((X.shape[0], -1))

X_train, X_test, y_train, y_test = train_test_split(
    X, y, train_size=train_samples, test_size=10000
)

scaler = StandardScaler()
```

```
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

#提高快速收敛的公差
clf = LogisticRegression(C=50.0 / train_samples, penalty="l1", solver="saga", tol=0.1)
clf.fit(X_train, y_train)
sparsity = np.mean(clf.coef_ == 0) * 100
score = clf.score(X_test, y_test)
print("L1点球的稀疏: %.2f%%" % sparsity)
print("带有L1点球的测试得分: %.4f" % score)

coef = clf.coef_.copy()
plt.figure(figsize=(10, 5))
scale = np.abs(coef).max()
for i in range(10):
      l1_plot = plt.subplot(2, 5, i + 1)
      l1_plot.imshow(
            coef[i].reshape(28, 28),
            interpolation="nearest",
            cmap=plt.cm.RdBu,
            vmin=-scale,
            vmax=scale,
      )Sparsity with L1 penalty
      l1_plot.set_xticks(())
      l1_plot.set_yticks(())
      l1_plot.set_xlabel("Class %i" % i)
plt.suptitle("Classification vector for...")

run_time = time.time() - t0
print("实例运行中... %.3f s" % run_time)
plt.show()
```

运行程序，输出如下，效果如图 3-13 所示。

```
L1点球的稀疏: 74.57%
带有L1点球的测试得分: 0.8253
实例运行时间 9.360 s
```

3.3.3　支持向量机

支持向量机(Support Vector Machine，SVM)在解决小样本、非线性及高维模式识别中表现出许多特有的优势，并能够推广应用到函数拟合等其他机器学习问题中。

支持向量机算法是建立在统计学习理论的 VC 维(Vapnik-Chervonenkis Dimension，指学习过程一致收敛的速度和推广性)理论和结构风险最小原理基础上的，其实质上是一个类分类器，是一个能够将不同类样本在样本空间分隔的超平面。换句话说，给定一些带标签的训练样本，支持向量机算法输出一个最优化的分割超平面。

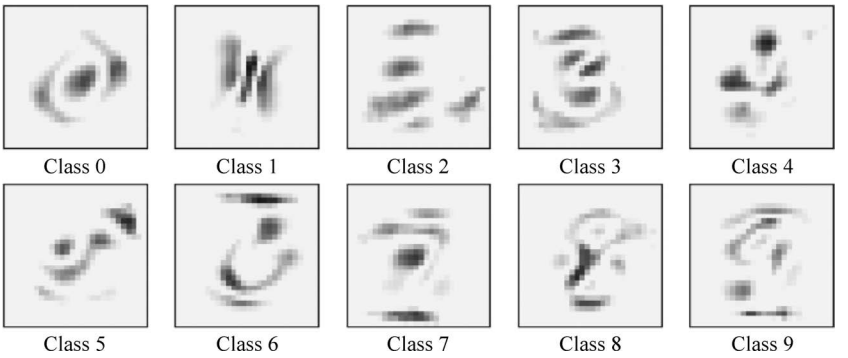

图 3-13　分类效果

1. 支持向量机概述

类似于逻辑回归，这个模型也是基于线性函数 $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}+b$ 的。不同于逻辑回归的是，支持向量机不输出概率，只输出类别。当 $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}+b$ 为正时，支持向量机预测属于正类。类似地，当 $\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}+b$ 为负时，支持向量机预测属于负类。

支持向量机的一个重要创新是核技巧(kernel trick)，核技巧观察到许多机器学习算法都可以写成样本间点积的形式。例如，支持向量机中的线性函数可以重写为

$$\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}+b=b+\sum_{i=1}^{m}\alpha_i\boldsymbol{x}^{\mathrm{T}}x^{(i)}$$

其中，$x^{(i)}$ 是训练样本，α_i 是向量 α 的系数。学习算法重写为这种形式允许将 $\boldsymbol{x}$ 替换为特征函数 $\phi(\boldsymbol{x})$ 的输出，点积替换为被称为核函数(kernel function)的函数 $k(\boldsymbol{x},x^{(i)})=\phi(\boldsymbol{x})\cdot\phi(x^{(i)})$。运算符 · 表示类似于 $\phi(\boldsymbol{x})\cdot\phi(x^{(i)})$ 的点积。

使用核估计替换点积之后，我们可以使用如下函数进行预测：

$$f(x)=b+\sum_{i}\alpha_i k(x,x^{(i)})$$

该函数是关于 x 非线性的，关于 $\phi(x)$ 是线性的。$\boldsymbol{\alpha}$ 和 $f(x)$ 之间的关系也是线性的。核函数完全等价于用 $\phi(\boldsymbol{x})$ 预处理所有的输入，然后在新的转换空间学习线性模型。

在某些情况下，$\phi(\boldsymbol{x})$ 甚至可以是无限维的，对于普通的显示方法而言，这将是无限的计算代价。在很多情况下，即使 $\phi(\boldsymbol{x})$ 是难算的，$k(\boldsymbol{x},\boldsymbol{x}')$ 却会是一个关于 $\boldsymbol{x}$ 的数。

提示：在几何中，超平面是一个空间的子空间，它是维度比所在空间小一维的空间。如果数据空间本身是三维的，则其超平面是二维平面；如果数据空间本身是二维的，则其超平面是一维的直线。

2. 函数方程描述

假设给定数据集 $D=\{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}$，$y_i\in\{-1,+1\}$，那么，样本空间中任意点 $\boldsymbol{x}$ 到超平面的距离可以写成

$$r=\frac{|\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}+b|}{\|\boldsymbol{w}\|}$$

由此也可以求得两个不同标签的支持向量到超平面的距离之和，也就是间隔，可以表示为

$$\gamma=\frac{2}{\|\boldsymbol{w}\|}$$

目标就是找到最大间隔的超平面，也就是要满足如下约束的参数 $\boldsymbol{w}$ 和 b，使得 γ 最大，即

$$\max_{\boldsymbol{w},b}\frac{2}{\|\boldsymbol{w}\|}$$

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

显然，最大化间隔 γ 只需要最小化 $\|\boldsymbol{w}\|$ 即可，所以约束变成

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

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

3. 参数求解

求解带有约束的最优化问题，一般常用的就是引入拉格朗日乘子 λ 构造拉格朗日函数。这属于多元函数的条件极值问题。

1）拉格朗日乘数

要找函数 $z=f(x,y)$ 在附加条件 $\varphi(x,y)=0$ 下的可能极值点，可以先作拉格朗日函数：

$$L(x,y)=f(x,y)+\lambda\varphi(x,y)$$

其中，λ 为参数，求其对 x、y 和 λ 的一阶偏导数，并使之为零，然后联立方程：

$$\begin{cases}f_x(x,y)+\lambda\varphi_x(x,y)=0\\ f_y(x,y)+\lambda\varphi_y(x,y)=0\\ \varphi_y(x,y)=0\end{cases}$$

由方程组解出 x、y 和 λ，这样求得的 (x,y) 就是函数 $f(x,y)$ 在附加条件 $\varphi(x,y)=0$ 下的可能极值点。

如果函数的自变量多于两个，且附加条件多于一个，如，要求函数

$$u=f(x,y,z,t)$$

在附加条件

$$\varphi(x,y,z,t)=0$$

$$\psi(x,y,z,t)=0$$

下的极值，可以先作拉格朗日函数：

$$L(x,y,z,t)=f(x,y,z,t)+\lambda f(x,y,z,t)+\mu f(x,y,z,t)$$

其中，λ、μ 为参数，求其对 x、y、z、t、λ 和 μ 的一阶偏导数，并使之为零，然后联立方程求解出 (x,y,z,t)。

2）拉格朗日对偶函数

函数本身是二次的（quadratic），函数的约束条件在其参数下是线性的，这样的函数称为凸优化问题。

首先构造支持向量机的拉格朗日函数，也就是损失函数：

$$L(\boldsymbol{w},b,\boldsymbol{\alpha})=\frac{1}{2}\|\boldsymbol{w}\|^2+\sum_{i=1}^{m}\alpha_i(1-y_i(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x}_i+b)),\alpha_i\geqslant 0$$

其中，$\boldsymbol{\alpha}=(\alpha_1,\alpha_2,\cdots,\alpha_n)^{\mathrm{T}}$。

可以看出，拉格朗日函数分为两部分：第一部分和原始的损失函数一样；第二部分表达的是约束条件。我们希望，构造的损失函数不仅能够代表原有的损失函数和约束条件，最好还能够表示最小化损失函数来求解 $\boldsymbol{w}$ 和 b 的意图，所以要先以 $\boldsymbol{\alpha}$ 为参数，求解 $L(\boldsymbol{w},b,\boldsymbol{\alpha})$ 的最大值，再以 $\boldsymbol{w}$ 和 b 为参数，求解 $L(\boldsymbol{w},b,\boldsymbol{\alpha})$ 的最小值。因此，目标可以写作如下形式：

$$\min_{\boldsymbol{w},b}\ \max_{\alpha_i\geqslant 0} L(\boldsymbol{w},b,\boldsymbol{\alpha}),\alpha_i\geqslant 0$$

4. 支持向量机的实现

支持向量机适合用于变量越多越好的问题，因此在神经网络之前，它对于文本和图片领域都算效果还不错的方法。支持向量机可以分类也可以回归，但一般用于分类问题更好。

【例 3-10】 支持向量机二分类 Python 实现。

具体的实现步骤为：

（1）采用垃圾邮件的数据集，导入包读取数据。

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold, StratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import plot_confusion_matrix

from sklearn.svm import SVC
from sklearn.svm import SVR
#from sklearn.svm import LinearSVC
from sklearn.datasets import load_boston
from sklearn.datasets import load_digits
from sklearn.datasets import make_blobs
from mlxtend.plotting import plot_decision_regions

spam = pd.read_csv('spam.csv')
spam.shape
spam.head(2)        #显示前 2 行结果，如图 3-14 所示
```

	A.1	A.2	A.3	A.4	A.5	A.6	A.7	A.8	A.9	A.10	...	A.49	A.50	A.51	A.52	A.53	A.54	A.55	A.56	A.57	spam
0	0.00	0.64	0.64	0.0	0.32	0.00	0.00	0.00	0.00	0.00	...	0.00	0.000	0.0	0.778	0.000	0.000	3.756	61	278	spam
1	0.21	0.28	0.50	0.0	0.14	0.28	0.21	0.07	0.00	0.94	...	0.00	0.132	0.0	0.372	0.180	0.048	5.114	101	1028	spam

图 3-14　显示前 2 行结果

（2）提取 X 和 y，划分训练测试集，将数据标准化。

```
X = spam.iloc[:, :-1]
y = spam.iloc[:, -1]
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 1000, stratify = y,
random_state = 0)
scaler = StandardScaler()
scaler.fit(X_train)
X_train_s = scaler.transform(X_train)
X_test_s = scaler.transform(X_test)
```

(3) 分别采用不同的核函数进行支持向量机的估计。

```
#线性核函数
model = SVC(kernel = "linear", random_state = 123)
model.fit(X_train_s, y_train)
model.score(X_test_s, y_test)
#二次多项式核
model = SVC(kernel = "poly", degree = 2, random_state = 123)
model.fit(X_train_s, y_train)
model.score(X_test_s, y_test)
#三次多项式
model = SVC(kernel = "poly", degree = 3, random_state = 123)
model.fit(X_train_s, y_train)
model.score(X_test_s, y_test)
#径向核
model = SVC(kernel = "rbf", random_state = 123)
model.fit(X_train_s, y_train)
model.score(X_test_s, y_test)
#S 核
model = SVC(kernel = "sigmoid",random_state = 123)
model.fit(X_train_s, y_train)
model.score(X_test_s, y_test)
```

(4) 正常情况下,径向核效果比较好,网格化搜索最优超参数。

```
param_grid = {'C': [0.1, 1, 10], 'gamma': [0.01, 0.1, 1]}
kfold = StratifiedKFold(n_splits = 10, shuffle = True, random_state = 1)
model = GridSearchCV(SVC(kernel = "rbf", random_state = 123), param_grid, cv = kfold)
model.fit(X_train_s, y_train)

model.best_params_
model.score(X_test_s, y_test)
pred = model.predict(X_test)
pd.crosstab(y_test, pred, rownames = ['Actual'], colnames = ['Predicted'])
```

运行程序,预测得到混淆矩阵如图 3-15 所示。

Predicted	email	spam
Actual		
email	476	130
spam	387	7

图 3-15　预测得到的混淆矩阵

【例 3-11】 支持向量机多分类 Python 实现。

分析:使用 sklearn 库自带的手写数字的案例,采用支持向量机分类。具体的实现步骤为:

（1）导入手写数字。

```
digits = load_digits()
dir(digits)
#图片数据(三维)
digits.images.shape
#拉成二维
digits.data.shape
#y的形状
digits.target.shape
#查看第15个图片,如图3-16所示
plt.imshow(digits.images[15], cmap = plt.cm.gray_r)
```

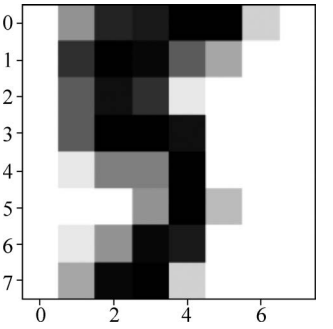

图 3-16　手写数字 5

（2）打印数字为 8 的图片展示。

```
images_8 = digits.images[digits.target == 8]

for i in range(1, 10):
      plt.subplot(3, 3, i)
      plt.imshow(images_8[i - 1], cmap = plt.cm.gray_r)
plt.tight_layout()
```

运行程序，效果如图 3-17 所示。

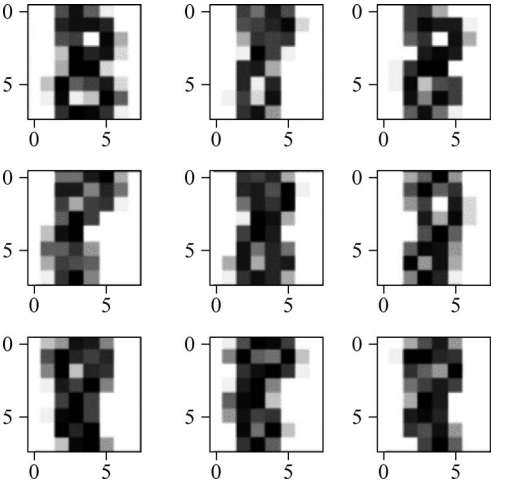

图 3-17　显示数字 8

(3) 提取 X 和 y,进行支持向量机分类。

```
X = digits.data
y = digits.target
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify = y, test_size = 0.2,
random_state = 0)
#核函数为线性函数
model = SVC(kernel = "linear", random_state = 123)
model.fit(X_train, y_train)
model.score(X_test, y_test)
#核函数为多项式,等级为 2
model = SVC(kernel = "poly", degree = 2, random_state = 123)
model.fit(X_train, y_train)
model.score(X_test, y_test)
#核函数为多项式,等级为 3
model = SVC(kernel = "poly", degree = 3, random_state = 123)
model.fit(X_train, y_train)
model.score(X_test, y_test)
#核函数 rbf
model = SVC(kernel = 'rbf', random_state = 123)
model.fit(X_train, y_train)
model.score(X_test, y_test)
##核函数 sigmoid
model = SVC(kernel = "sigmoid",random_state = 123)
model.fit(X_train, y_train)
model.score(X_test, y_test)
```

(4) 网格化搜索最优超参数,预测得到混淆矩阵。

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10], 'gamma': [0.001, 0.01, 0.1, 1, 10]}
kfold = StratifiedKFold(n_splits = 10, shuffle = True, random_state = 1)
model = GridSearchCV(SVC(kernel = 'rbf',random_state = 123), param_grid, cv = kfold)
model.fit(X_train, y_train)

model.best_params_
model.score(X_test, y_test)

pred = model.predict(X_test)
pd.crosstab(y_test, pred, rownames = ['Actual'], colnames = ['Predicted'])
```

运行程序,得到混淆矩阵如图 3-18 所示。

(5) 画热力图。

```
plt.rcParams['font.sans-serif'] = ['SimHei']          #设置中文字体
plot_confusion_matrix(model, X_test, y_test,cmap = 'Blues')
plt.tight_layout()
```

运行程序,效果如图 3-19 所示。

3.3.4　Adaboost 算法

自适应增强(Adaboost)是一种迭代算法,在每一轮中加入一个新的弱分类器,直到

Predicted	0	1	2	3	4	5	6	7	8	9
Actual										
0	36	0	0	0	0	0	0	0	0	0
1	0	36	0	0	0	0	0	0	0	0
2	0	0	35	0	0	0	0	0	0	0
3	0	0	0	36	0	0	0	1	0	0
4	0	0	0	0	35	0	0	0	1	0
5	0	0	0	0	0	37	0	0	0	0
6	0	0	0	0	0	0	36	0	0	0
7	0	0	0	0	0	0	0	36	0	0
8	0	2	0	0	0	0	0	0	33	0
9	0	0	0	0	0	0	0	0	0	36

图 3-18　混淆矩阵

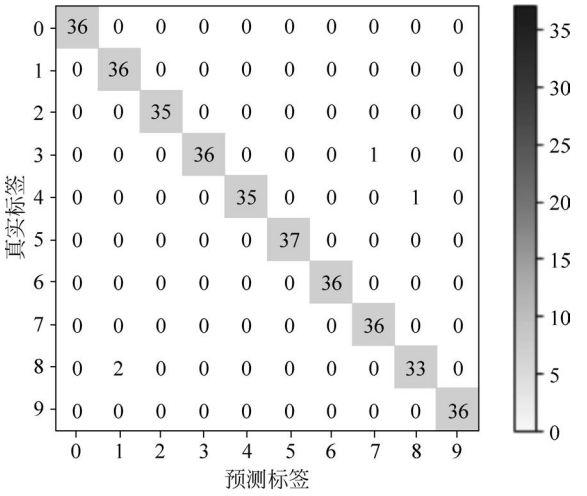

图 3-19　热力图

达到某个预定的足够小的错误率。其核心思想是针对同一个训练集训练不同的分类器(弱分类器),然后把这些弱分类器集合起来,构成一个更强的最终分类器(强分类器)。

Adaboost 算法根据每次训练集中每个样本的分类是否正确,以及上次的总体分类的准确率,来确定每个样本的权值。将修改过权值的新数据集送给下层分类器进行训练,最后将每次训练得到的分类器最后融合起来,作为最后的决策分类器。

1. Adaboost 算法的流程

Adaboost 算法的流程主要表现在:

(1) 初始化训练数据的权值分布 D_1。每一个训练样本最开始时都被赋予相同的权重:$1/m$。

$$D_1 = (w_{11}, w_{12}, \cdots, w_{1m}), w_{1i} = 1/m, i = 1, 2, \cdots, N$$

如果某个样本点已经被准确地分类,那么在构造下一个训练集中,它被选中的概率就被降低;相反,如果某个样本点没有被准确地分类,那么它的权重就得到提高。

(2) $m=1,2,\cdots,M$。

① 使用具有权值分布 D_m 的训练数据集学习，得到基本的分类器：

$$G_m(x):\chi\rightarrow\{-1,+1\}$$

② 求 $G_m(x)$在训练集上的分类误差率：

$$e_m=\sum_{i=1}^{N}P(G_m(x_i)\neq y_i)=\sum_{i=1}^{N}w_{mi}I(G_m(x_i)\neq y_i)$$

③ 计算 $G_m(x)$的系数：

$$\alpha_m=\frac{1}{2}\log\frac{1-e_m}{e_m}$$

④ 更新训练集的权值分布：

$$w_{m+1,i}=\frac{w_{mi}}{Z_m}\exp(-\alpha_m y_i G_m(x_i))$$

其中，Z_m 是规范化因子，$Z_m=\sum_{i=1}^{N}w_{mi}\exp(-\alpha_m y_i G_m(x_i))$。

(3) 构建基本分类器的线性组合。

$$f(x)=\sum_{m=1}^{N}\alpha_m G_m(x)$$

(4) 最终分类器：

$$G(x)=\mathrm{sign}(f(x))=\mathrm{sign}\Big(\sum_{m=1}^{N}\alpha_m G_m(x)\Big)$$

下面对这几个步骤进行说明。

第(1)步假设训练数据具有均匀的权值分布，即每个训练样本在基本分类器的学习中作用相同，这一假设保证第(1)步能够在原始数据上学习基本分类器 $G_1(x)$。

第(2)步 Adaboost 反复学习基本分类器，在每一轮 $m=1,2,\cdots,M$ 顺序执行下列操作。

① 使用当前分布 D_m 加权的训练数据集，学习基本分类器 $G_m(x)$。

② 计算基本分类器 $G_m(x)$在加权训练集上的误分类误差：

$$e_m=\sum_{i=1}^{N}w_{mi}I(G_m(x_i)\neq y_i)=\sum_{G_m(x_i)\neq y_i}w_{mi}$$

误差率是指被 $G_m(x)$误分类样本的权值之和。

③ 计算基本分类器 $G_m(x)$的系数 α_m。由

$$\alpha_m=\frac{1}{2}\log\frac{1-e_m}{e_m}$$

可知，当 $e_m\leqslant\frac{1}{2}$时，$\alpha_m\geqslant 0$，且 α_m 随着 e_m 的减小而增大，所以分类误差率越小的基本分类器在最终分类器中的作用越大。

④ 更新训练数据的权值分布为下一轮做准备。在式 $w_{m+1,i}=\frac{w_{mi}}{Z_m}\exp(-\alpha_m y_i G_m(x_i))$中：

- $y_i = G_m(x_i)$时，$y_i G_m(x_i) = 1$。
- $y_i \neq G_m(x_i)$时，$y_i G_m(x_i) = -1$。

所以有

$$w_{m+1,i} = \begin{cases} \dfrac{w_{mi}}{Z_m} e^{-\alpha_m}, y_i = G_m(x_i) \\ \dfrac{w_{mi}}{Z_m} e^{\alpha_m}, y_i \neq G_m(x_i) \end{cases}$$

由此可知，被基本分类器$G_m(x)$误分类的样本权值得以扩大，而被正确分类的样本权值缩小。误分类样本权值被放大$e^{2\alpha_m} = \dfrac{1-e_m}{e_m}$倍。

第(3)步，实现M个基本分类器的加权表决。

2. Adaboost 算法的 Python 实现

在前面已对 Adaboost 算法的基本思想，算法步骤及算法步骤的说明进行了介绍，下面直接演示 Adaboost 算法的 Python 实现。

【例 3-12】 Adaboost 实现二分类。

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.datasets import load_iris
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import LinearSVC
from sklearn import metrics
from sklearn.metrics import roc_auc_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

plt.rcParams['font.sans-serif'] = ['SimHei']          #设置中文字体
'''导入数据'''
cancer = load_breast_cancer()
x = cancer.data
y = cancer.target
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.333, random_state
=0)                                                   # 分训练集和验证集
# Adaboost 分类器,使用 SVM 为弱分类器
model = AdaBoostClassifier(LinearSVC(C=1), n_estimators=40, learning_rate=0.9,
algorithm='SAMME')                                    #使用 SVM 弱分类器
model.fit(x_train, y_train)
#对测试集进行预测
y_pred = model.predict(x_test)
predictions = [round(value) for value in y_pred]
#计算准确率
accuracy = accuracy_score(y_test, predictions)
print("Accuracy: %.2f%%" % (accuracy * 100.0))
print(f"\nAdaboost 模型混淆矩阵为: \n{metrics.confusion_matrix(y_test,y_pred)}")
```

```
'''绘制 ROC 曲线'''
fpr,tpr,threshold = roc_curve(y_test,y_pred)      # 计算 ROC 曲线,即真阳率、假阳率
roc_auc = auc(fpr, tpr)                           # 计算 AUC 值
lw = 2
plt.figure(figsize = (8, 5))
plt.plot(fpr, tpr, color = 'darkorange',
lw = lw, label = 'ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color = 'navy', lw = lw, linestyle = '--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('假正例率')
plt.ylabel('正例率')
plt.title('Adaboost ROC')
plt.legend(loc = "lower right")
plt.show()
print(f"\nAdaboost 模型 AUC 值为: \n{roc_auc_score(y_test,y_pred)}")
```

运行程序,输出如下,效果如图 3-20 所示。

```
精度: 95.79 %
Adaboost 模型混淆矩阵为:
[[ 63   5]
 [  3 119]]
Adaboost 模型 AUC 值为:
0.9509402121504339
```

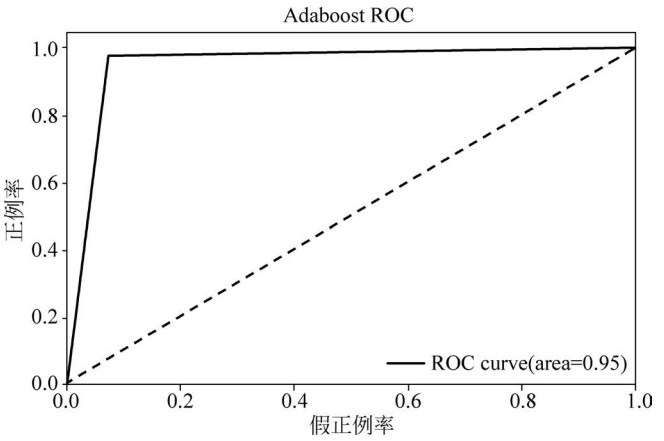

图 3-20　Adaboost ROC 曲线

3.3.5　决策树

决策树(decision tree)是一种基本的分类与回归方法,其每个非叶结点表示一个特征属性上的测试,每个分支代表这个特征属性在某个值域上的输出,而每个叶结点存放一个类别。使用决策树进行决策的过程就是从根结点开始,测试待分类项中相应的特征属性,并按照其值选择输出分支,直到到达叶结点,将叶结点存放的类别作为决策结果。

决策树通常有三个步骤：特征选择、决策树的生成、决策树的修剪。

1. 特征选择

决策树学习的算法通常是一个递归地选择最优特征，并根据该特征对训练数据进行分割，使得各个子数据集有一个最好的分类的过程。这一过程对应着对特征空间的划分，也对应着决策树的构建。

(1) 开始：构建根结点，将所有训练数据都放在根结点，选择一个最优特征，按照这一特征将训练数据集分割成子集，使得各个子集有一个在当前条件下最好的分类。

(2) 如果这些子集已经能够被基本正确分类，那么构建叶结点，并将这些子集分到所对应的叶结点。

(3) 如果还有子集不能够被正确地分类，那么就对这些子集选择新的最优特征，继续对其进行分割，构建相应的结点，如果递归进行，直至所有训练数据子集被基本正确地分类，或者没有合适的特征为止。

(4) 每个子集都被分到叶结点上，即都有了明确的类，这样就生成了一棵决策树。

使用决策树做预测的过程为：收集数据→准备数据→分析数据→测试数据→使用算法。

本节使用 ID3 算法来划分数据集，该算法处理如何划分数据集，何时停止划分数据集。

1) 信息增益

划分数据集的大原则是将无序数据变得更加有序，但是各种方法都有各自的优缺点，信息论是量化处理信息的分支科学，在划分数据集前后信息发生的变化称为信息增益，获得信息增益最高的特征就是最好的选择，所以必须先学习如何计算信息增益。知道如何计算信息增益，就可以计算每个特征值划分数据集得到的信息增益。

熵定义为信息的期望值，如果待分类的事物可能划分在多个类之中，则符号 x_i 的信息定义为

$$l(x_i) = -\log_2 p(x_i)$$

其中，$p(x_i)$是选择该分类的概率。

为了计算熵，需要计算所有类别所有可能值所包含的信息期望值，可通过下式得到：

$$H = -\sum_{i=1}^{n} p(x_i)\log_2 p(x_i)$$

其中，n 为分类数目。熵越大，随机变量的不确定性就越大。

当熵中的概率由数据估计(特别是最大似然估计)得到时，所对应的熵称为经验熵(empirical entropy)。

(1) 条件熵。

信息增益表示得知特征 X 的信息而使得类 Y 的信息不确定性减少的程度。

条件熵 $H(X|Y)$表示在已知随机变量 X 的条件下随机变量 Y 的不确定性。随机变量 X 给定条件下随机变量 Y 的条件熵(conditional entropy)$H(Y|X)$，定义为 X 给定条件下 Y 的条件概率分布的熵对 X 的数学期望：

$$H(Y \mid X)=\sum_{i=1}^{n} p_i H(Y \mid X=x_i)$$

其中，$p_i=P(X=x_i)$。

当熵和条件熵中的概率由数据估计（特别是极大似然估计）得到时，所对应的分别为经验熵和经验条件熵，此时如果有 0 概率，令 $0\log_2 0=0$。

（2）信息增益。

信息增益是相对于特征而言的。所以，特征 A 对训练数据集 D 的信息增益 $g(D,A)$，定义为集合 D 的经验熵 $H(D)$ 与特征 A 给定条件下 D 的经验条件熵 $H(D|A)$ 之差，即

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

一般地，熵 $H(D)$ 与条件熵 $H(D|A)$ 之差成为互信息（mutual information）。决策树学习中的信息增益等价于训练数据集中类与特征的互信息。

（3）信息增益比。

特征 A 对训练数据集 D 的信息增益比 $g_R(D,A)$，定义为其信息增益 $g(D,A)$ 与训练数据集 D 的经验熵之比：

$$g_R(D,A)=\frac{g(D,A)}{H(D)}$$

2）编写代码计算经验熵

表 3-1 列出了贷款申请样本数据情况。

表 3-1　贷款申请样本数据

ID	年龄	是否有工作	是否有房子	信贷情况	类别（是否放贷）
1	青年	无	无	一般	否
2	青年	无	无	好	否
3	青年	有	无	好	是
4	青年	有	有	一般	是
5	青年	无	无	一般	否
6	中年	无	无	一般	否
7	中年	无	无	好	否
8	中年	有	有	好	是
9	中年	无	有	非常好	是
10	中年	无	有	非常好	是
11	老年	无	有	非常好	是
12	老年	无	有	好	是
13	老年	有	无	好	是
14	老年	有	无	非常好	是
15	老年	无	无	好	否

在编写代码之前，先对数据集进行属性标注。

- 年龄：0 代表青年，1 代表中年，2 代表老年。
- 是否有工作：0 代表无，1 代表有。

- 是否有自己的房子：0 代表无，1 代表有。
- 信贷情况：0 代表一般，1 代表好，2 代表非常好。
- 类别(是否放贷)：no 代表否，yes 代表是。

创建数据集，计算经验熵的代码为：

```
from math import log

"""创建测试数据集"""
def creatDataSet():
    #数据集
    dataSet = [[0, 0, 0, 0, 'no'],
               [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']]
    #分类属性
    labels = ['年龄','是否有工作','是否有自己的房子','信贷情况']
    #返回数据集和分类属性
    return dataSet,labels

"""计算给定数据集的经验熵(香农熵)"""
def calcShannonEnt(dataSet):
    #返回数据集行数
    numEntries = len(dataSet)
    #保存每个标签(label)出现次数的字典
    labelCounts = {}
    #对每组特征向量进行统计
    for featVec in dataSet:
        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                  #label 计数
    shannonEnt = 0.0                                    #经验熵
    #计算经验熵
    for key in labelCounts:
        prob = float(labelCounts[key])/numEntries       #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
```

```
#main 函数
if __name__ == '__main__':
    dataSet,features = creatDataSet()
    print('数据集的经验熵: ',calcShannonEnt(dataSet))
```

运行程序,输出如下:

```
数据集的经验熵: 0.9709505944546686
```

3) 利用代码计算信息增益

以下代码实现计算信息增益。

```
from math import log
"""创建测试数据集"""
def creatDataSet():
    #数据集
    dataSet = [[0, 0, 0, 0, 'no'],
               [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']]
    #分类属性
    labels = ['年龄','是否有工作','是否有自己的房子','信贷情况']
    #返回数据集和分类属性
    return dataSet,labels

"""计算给定数据集的经验熵(香农熵)"""
def calcShannonEnt(dataSet):
    #返回数据集行数
    numEntries = len(dataSet)
    #保存每个标签(label)出现次数的字典
    labelCounts = {}
    #对每组特征向量进行统计
    for featVec in dataSet:
        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                  #label 计数

    shannonEnt = 0.0                                    #经验熵
```

```
    #计算经验熵
    for key in labelCounts:
        prob = float(labelCounts[key])/numEntries          #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

"""计算给定数据集的经验熵(香农熵)"""
def chooseBestFeatureToSplit(dataSet):
    #特征数量
    numFeatures = len(dataSet[0]) - 1
    #计算数据集的香农熵
    baseEntropy = calcShannonEnt(dataSet)
    #信息增益
    bestInfoGain = 0.0
    #最优特征的索引值
    bestFeature = -1
    #遍历所有特征
    for i in range(numFeatures):
        #获取 dataSet 的第 i 个所有特征
        featList = [example[i] for example in dataSet]
        #创建集合,元素不可重复
        uniqueVals = set(featList)
        #经验条件熵
        newEntropy = 0.0
        #计算信息增益
        for value in uniqueVals:
            #subDataSet 划分后的子集
            subDataSet = splitDataSet(dataSet, i, value)
            #计算子集的概率
            prob = len(subDataSet) / float(len(dataSet))
            #根据公式计算经验条件熵
            newEntropy += prob * calcShannonEnt((subDataSet))
        #信息增益
        infoGain = baseEntropy - newEntropy
        #打印每个特征的信息增益
        print("第 %d 个特征的增益为 %.3f" % (i, infoGain))
        #计算信息增益
        if(infoGain > bestInfoGain):
            #更新信息增益,找到最大的信息增益
            bestInfoGain = infoGain
            #记录信息增益最大特征的索引值
            bestFeature = i
            #返回信息增益最大特征的索引值
    return bestFeature

"""按照给定特征划分数据集"""
def splitDataSet(dataSet,axis,value):
    retDataSet = []
    for featVec in dataSet:
        if featVec[axis] == value:
            reducedFeatVec = featVec[:axis]
            reducedFeatVec.extend(featVec[axis + 1:])
```

```
                retDataSet.append(reducedFeatVec)
        return retDataSet

"""main 函数"""
if __name__ == '__main__':
        dataSet,features = creatDataSet()
        print("最优索引值: " + str(chooseBestFeatureToSplit(dataSet)))
```

运行程序，输出如下：

```
第 0 个特征的增益为 0.083
第 1 个特征的增益为 0.324
第 2 个特征的增益为 0.420
第 3 个特征的增益为 0.363
最优索引值: 2
```

从结果可看出，最优特征的索引值为 3，也就是特征 A3。

2. 决策树的生成和修剪

构建决策树的算法有很多，如 ID3、C4.5 和 CART，这些算法在运行时并不总是在每次划分数据分组时都会消耗特征。决策树生成算法递归地产生决策树，直到不能继续下去为止。这样产生的树往往对训练数据的分类很准确，但对未知的测试数据的分类却没有那么准确，即出现过拟合现象。

过拟合的原因在于学习时过多地考虑如何提高对训练数据的正确分类，从而构建出过于复杂的决策树。解决这个问题的办法是考虑决策树的复杂度，对已生成的决策树进行简化。

决策树的构建可用 ID3 算法构建，也可用 C4.5 算法构建。

(1) ID3 算法。

ID3 算法的核心是在决策树各个结点上对应信息增益准则选择特征，递归地构建决策树。具体方法是：

① 从根结点开始，对结点计算所有可能的信息增益，选择信息增益最大的特征作为结点的特征。

② 由该特征的不同取值建立子结点，再对子结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。

③ 得到一个决策树。

ID3 算法相当于用极大似然法进行概率模型的选择。该算法步骤为：

输入：训练数据集 D，特征集 A，阈值 ε；

输出：决策树 T。

① 如果 D 中所有的实例属于同一类 C_k，则 T 为单结点，并将类 C_k 作为该结点的类标记，返回 T；

② 如果 $A \neq \phi$，则 T 为单结点树，并将 D 中实例数最大的类 C_k 作为该结点的类标记，返回 T；

③ 否则，计算 A 中各特征对 D 的信息增益，选择信息增益最大的特征 A_g；

④ 如果 A_g 的信息增益小于阈值 ε，则置 T 为单结点树，并将 D 中实例数最大的类 C_k 作为该结点的类标记，返回 T；

⑤ 否则，对 A_g 的每一个可能值 α_i，依 $A_g=\alpha_i$ 将 D 分割为若干非空子集 D_i，将 D_i 中实例数最大的类作为标记，构建子结点，由结点及其子结点构成树 T，返回 T；

⑥ 对第 i 个子结点，以 D_i 为训练集，以 $A-\{A_g\}$ 为特征集，递归地调用步①～⑤，得到子树 T_i，返回 T_i。

(2) C4.5 算法。

C4.5 算法与 ID3 算法相似，但是做了改进，将信息增益比作为选择特征的标准。

① 递归构建决策树。

从数据集构造决策树算法所需的子功能模块工作原理如下：得到原始数据集，然后基于最好的属性值划分数据集，由于特征值可能多于两个，因此可能存在大于两个分支的数据集划分，第一次划分之后，数据将被向下传递到树分支的下一个结点，在此结点再次划分数据，因此可以使用递归的原则处理数据集。

② 递归结束的条件。

递归结束的条件：程序完全遍历所有划分数据集的属性，或者每个分支下的所有实例都具有相同的分类，如果所有实例都具有相同的分类，则得到一个叶结点或者终止块，任何到达叶结点的数据必然属于叶结点的分类。

③ 编写 ID3 算法的代码。

```
from math import log
import operator

"""计算给定数据集的经验熵(香农熵)"""
def calcShannonEnt(dataSet):
    #返回数据集行数
    numEntries = len(dataSet)
    #保存每个标签(label)出现次数的字典
    labelCounts = {}
    #对每组特征向量进行统计
    for featVec in dataSet:
        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,添
                                                   #加进去
            labelCounts[currentLabel] = 0
        labelCounts[currentLabel] += 1                #label 计数
    shannonEnt = 0.0                                  #经验熵
    #计算经验熵
    for key in labelCounts:
        prob = float(labelCounts[key])/numEntries     #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

"""创建测试数据集"""
def createDataSet():
    #数据集
```

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

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

"""计算给定数据集的经验熵(香农熵)"""
def chooseBestFeatureToSplit(dataSet):
    #特征数量
    numFeatures = len(dataSet[0]) - 1
    #计数数据集的香农熵
    baseEntropy = calcShannonEnt(dataSet)
    #信息增益
    bestInfoGain = 0.0
    #最优特征的索引值
    bestFeature = -1
    #遍历所有特征
    for i in range(numFeatures):
        #获取 dataSet 的第 i 个所有特征
        featList = [example[i] for example in dataSet]
        #创建集合,元素不可重复
        uniqueVals = set(featList)
        #经验条件熵
```

```
            newEntropy = 0.0
            #计算信息增益
            for value in uniqueVals:
                #subDataSet 划分后的子集
                subDataSet = splitDataSet(dataSet, i, value)
                #计算子集的概率
                prob = len(subDataSet) / float(len(dataSet))
                #根据公式计算经验条件熵
                newEntropy += prob * calcShannonEnt((subDataSet))
            #信息增益
            infoGain = baseEntropy - newEntropy
            #打印每个特征的信息增益
            print("第%d个特征的增益为%.3f" % (i, infoGain))
            #计算信息增益
            if (infoGain > bestInfoGain):
                 #更新信息增益,找到最大的信息增益
                 bestInfoGain = infoGain
                 #记录信息增益最大特征的索引值
                 bestFeature = i
                 #返回信息增益最大特征的索引值
    return bestFeature

"""统计 classList 中出现次数最多的元素(类标签)"""
def majorityCnt(classList):
    classCount = {}
    #统计 classList 中每个元素出现的次数
    for vote in classList:
        if vote not in classCount.keys():
            classCount[vote] = 0
            classCount[vote] += 1
        #根据字典的值降序排列

         sortedClassCount = sorted(classCount.items(), key = operator.itemgetter(1),
reverse = True)
        return sortedClassCount[0][0]

"""创建决策树"""
def createTree(dataSet,labels,featLabels):
    #取分类标签(是否放贷: yes or no)
    classList = [example[-1] for example in dataSet]
    #如果类别完全相同,则停止继续划分
    if classList.count(classList[0]) == len(classList):
         return classList[0]
    #遍历完所有特征时返回出现次数最多的类标签
    if len(dataSet[0]) == 1:
         return majorityCnt(classList)
    #选择最优特征
    bestFeat = chooseBestFeatureToSplit(dataSet)
    #最优特征的标签
    bestFeatLabel = labels[bestFeat]
    featLabels.append(bestFeatLabel)
    #根据最优特征的标签生成树
    myTree = {bestFeatLabel: {}}
```

```
        #删除已经使用的特征标签
        del(labels[bestFeat])
        #得到训练集中所有最优特征的属性值
        featValues = [example[bestFeat] for example in dataSet]
        #去掉重复的属性值
        uniqueVls = set(featValues)
        #遍历特征,创建决策树
        for value in uniqueVls:
            myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet,bestFeat,value),
                                                      labels,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:

```
第 0 个特征的增益为 0.083
第 1 个特征的增益为 0.324
第 2 个特征的增益为 0.420
第 3 个特征的增益为 0.363
第 0 个特征的增益为 0.252
第 1 个特征的增益为 0.918
第 2 个特征的增益为 0.474
{'是否有自己的房子': {0: {'是否有工作': {0: 'no', 1: 'yes'}}, 1: 'yes'}}
```

3. 决策树的剪枝

决策树生成算法递归产生的树往往对训练数据的分类很准确,但对未知测试数据的分类却没有那么精确,即会出现过拟合现象。解决方法是考虑决策树的复杂度,对已经生成的树进行简化(剪枝,pruning)。

决策树学习的损失函数定义为

$$C_\alpha(T)=\sum_{t=1}^{|T|}N_tH_t(T)+\alpha\mid T\mid$$

其中,T 表示这棵子树的叶结点;$H_t(T)$表示第 t 个叶子的熵;N_t 表示该叶子所含的训练样例的个数;α 为惩罚系数;$|T|$表示子树的叶结点的个数。

又因为经验熵为

$$H_t(T)=-\sum_k\frac{N_{tk}}{N_t}\log\frac{N_{tk}}{N_t}$$

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

所以有 $C_\alpha(T)=C(T)+\alpha|T|$。

其中,$C(T)$表示模型对训练数据的预测误差,即模型与训练数据的拟合程度;$|T|$表示

模型复杂度；参数 $\alpha \geqslant 0$ 控制两者之间的影响，较大的 α 促使选择较简单的模型(树)，较小的 α 促使选择较复杂的模型(树)，$\alpha=0$ 只考虑模型与训练数据的拟合程度，不考虑模型的复杂度。

剪枝就是当 α 确定时，选择损失函数最小的模型，即损失函数最小的子树。

(1) 当 α 值确定时，子树越大，往往与训练数据的拟合越好，但是模型的复杂度越高；

(2) 子树越小，模型的复杂度就越低，但是往往与训练数据的拟合不好；

(3) 损失函数正好表示了对两者的平衡。

如果一棵子树的损失函数值越大，则说明这棵子树越差，因此希望让每一棵子树的损失函数值尽可能地小，损失函数最小化就是用正则化的极大似然估计进行模型选择的过程。

决策树的剪枝过程(泛化过程)就是从叶结点开始递归，记其父结点将所有子结点回缩后的子树为 T_b(分类值取类别比例最大的特征值)，未回缩的子树为 T_α，如果 $C_\alpha(T_\alpha) \geqslant C_\alpha(T_b)$ 说明回缩后使得损失函数减小了，那么应该使这棵子树回缩，递归直到无法回缩为止，这样使用“贪心”的思想进行剪枝可以降低损失函数，也使决策树得到泛化。

【例 3-13】 使用已训练好的决策树做分类。

提示：只需要提供这个人是否有房子、是否有工作这两个信息即可，无须提供冗余的信息。

```
from math import log
import operator

"""计算给定数据集的经验熵(香农熵)"""
def calcShannonEnt(dataSet):
    #返回数据集行数
    numEntries = len(dataSet)
    #保存每个标签(label)出现次数的字典
    labelCounts = {}
    #对每组特征向量进行统计
    for featVec in dataSet:
        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,添加进去
            labelCounts[currentLabel] = 0
        labelCounts[currentLabel] += 1                      #label 计数

    shannonEnt = 0.0                                        #经验熵
    #计算经验熵
    for key in labelCounts:
        prob = float(labelCounts[key])/numEntries           #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

"""创建测试数据集"""
def createDataSet():
    #数据集
```

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

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

"""计算给定数据集的经验熵(香农熵)"""
def chooseBestFeatureToSplit(dataSet):
    #特征数量
    numFeatures = len(dataSet[0]) - 1
    #计算数据集的香农熵
    baseEntropy = calcShannonEnt(dataSet)
    #信息增益
    bestInfoGain = 0.0
    #最优特征的索引值
    bestFeature = -1
    #遍历所有特征
    for i in range(numFeatures):
        # 获取 dataSet 的第 i 个所有特征
        featList = [example[i] for example in dataSet]
        #创建集合,元素不可重复
        uniqueVals = set(featList)
        #经验条件熵
        newEntropy = 0.0
        #计算信息增益
        for value in uniqueVals:
```

```
                #subDataSet 划分后的子集
                subDataSet = splitDataSet(dataSet, i, value)
                #计算子集的概率
                prob = len(subDataSet) / float(len(dataSet))
                #根据公式计算经验条件熵
                newEntropy += prob * calcShannonEnt((subDataSet))
            #信息增益
            infoGain = baseEntropy - newEntropy
            #打印每个特征的信息增益
            print("第%d个特征的增益为%.3f" % (i, infoGain))
            #计算信息增益
            if (infoGain > bestInfoGain):
                #更新信息增益,找到最大的信息增益
                bestInfoGain = infoGain
                #记录信息增益最大特征的索引值
                bestFeature = i
                #返回信息增益最大特征的索引值
    return bestFeature

"""统计 classList 中出现次数最多的元素(类标签)"""
def majorityCnt(classList):
    classCount = {}
    #统计 classList 中每个元素出现的次数
    for vote in classList:
        if vote not in classCount.keys():
            classCount[vote] = 0
            classCount[vote] += 1
        #根据字典的值降序排列

         sortedClassCount = sorted(classCount.items(), key = operator.itemgetter(1),
reverse = True)
        return sortedClassCount[0][0]

"""创建决策树"""
def createTree(dataSet,labels,featLabels):
    #取分类标签(是否放贷: yes or no)
    classList = [example[-1] for example in dataSet]
    #如果类别完全相同,则停止继续划分
    if classList.count(classList[0]) == len(classList):
        return classList[0]
    #遍历完所有特征时返回出现次数最多的类标签
    if len(dataSet[0]) == 1:
        return majorityCnt(classList)
    #选择最优特征
    bestFeat = chooseBestFeatureToSplit(dataSet)
    #最优特征的标签
    bestFeatLabel = labels[bestFeat]
    featLabels.append(bestFeatLabel)
    #根据最优特征的标签生成树
    myTree = {bestFeatLabel: {}}
    #删除已经使用的特征标签
    del(labels[bestFeat])
    #得到训练集中所有最优特征的属性值
```

```
        featValues = [example[bestFeat] for example in dataSet]
        #去掉重复的属性值
        uniqueVls = set(featValues)
        #遍历特征,创建决策树
        for value in uniqueVls:
            myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet,bestFeat,value),
                                                      labels,featLabels)
        return myTree

"""使用决策树进行分类"""
def classify(inputTree,featLabels,testVec):
    #获取决策树结点
    firstStr = next(iter(inputTree))
    #下一个字典
    secondDict = inputTree[firstStr]
    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)
    #测试数据,0表示青年,1表示有工作
    testVec = [0,1]
    result = classify(myTree,featLabels,testVec)
    #分类是否放贷
    if result == 'yes':
        print('有保障,放贷')
    if result == 'no':
        print('无保障,不放贷')
```

运行程序,输出如下:

```
第0个特征的增益为0.083
第1个特征的增益为0.324
第2个特征的增益为0.420
第3个特征的增益为0.363
第0个特征的增益为0.252
第1个特征的增益为0.918
第2个特征的增益为0.474
有保障,放贷
```

3.3.6　随机森林

集成学习通过训练学习出多个估计器,当需要预测时通过结合器将多个估计器的结

果整合起来当作最后的结果输出。图 3-21 展示了集成学习的基本流程。

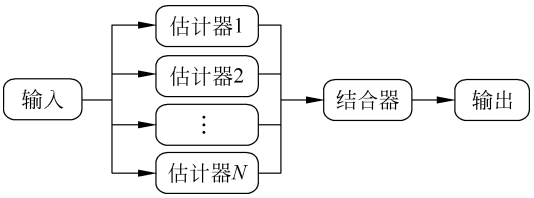

图 3-21　集成算法流程

集成学习的优势是提升了单个估计器的通用性与健壮性，比单个估计器拥有更好的预测性能。集成学习的另一个特点是能方便地进行并行化操作。

1. Bagging 算法

Bagging 算法是一种集成学习算法，其全称为自助聚集算法（Bootstrap aggregating），该算法由 Bootstrap 与 Aggregating 两部分组成。

图 3-22 展示了 Bagging 算法使用自助取样（Bootstrapping）生成多个子数据的实例。

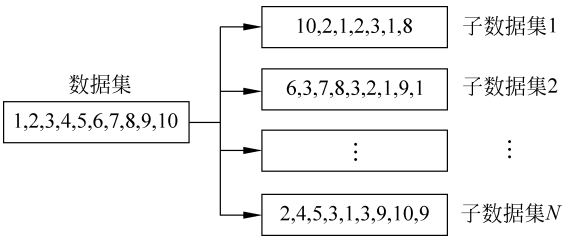

图 3-22　自助取样法

算法的步骤：假设有一个大小为 N 的训练数据集，每次从该数据集中有放回地选取出大小为 M 的子数据集，一共选 K 次，根据这 K 个子数据集，训练学习出 K 个模型。当要预测时，使用这 K 个模型进行预测，再通过取平均值或者多数分类的方式，得到最后的预测结果。

2. 随机森林算法

将多个决策树结合在一起，每次数据集是随机有放回地选出，同时随机选出部分特征作为输入，所以该算法被称为随机森林算法。可以看到随机森林算法是以决策树为估计器的 Bagging 算法。

图 3-23 展示了随机森林算法的具体流程。其中，结合器在分类问题中，选择多数分类结果作为最后的结果；在回归问题中，对多个回归结果取平均值作为最后的结果。

使用 Bagging 算法能降低过拟合的情况，从而带来了更好的性能。单个决策树对训练集的噪声非常敏感，但通过 Bagging 算法降低了训练出的多棵决策树之间关联性，有效缓解了上述问题。

1）算法步骤

假设训练集 T 的大小为 N，特征数目为 M，随机森林的大小为 K，随机森林算法的实现步骤为：

遍历随机森林的大小 K 次：

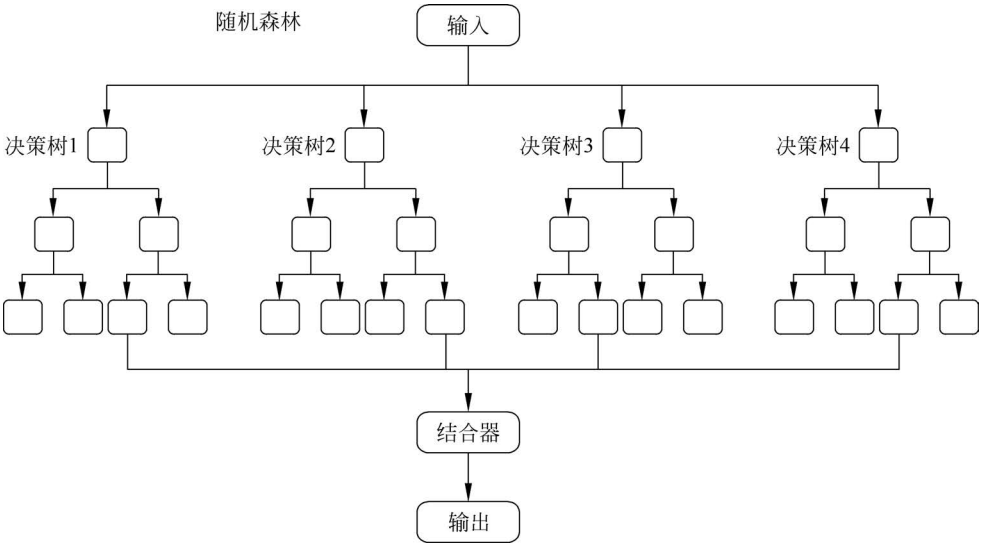

图 3-23　随机森林算法的具体流程

- 从训练集 T 中有放回抽样的方式，取样 N 次形成一个新子训练集 D；
- 随机选择 m 特征，其中 $m<M$；
- 使用新的训练集 D 和 m 个特征，学习出一个完整的决策树；
- 得到随机森林。

上面算法中 m 的选择：对于分类问题，可以在每次划分时使用 $\sqrt{M}$ 个特征，对于回归问题，选择$\frac{M}{3}$但不少于 5 个特征。

2）优点

根据随机森林的算法特点，它的优点主要表现为：

（1）对于很多种资料，可以产生高准确度的分类器；

（2）可以处理大量的输入变量；

（3）可以在决定类别时，评估变量的重要性；

（4）在建造森林时，可以在内部对于一般化后的误差产生不偏差的估计；

（5）包含一个好方法可以估计丢失的资料，并且如果有很大一部分的资料丢失，仍可以维持准确度；

（6）对于不平衡的分类资料集来说，可以平衡误差；

（7）可被延伸应用在未标记的资料上；

（8）学习过程很快速。

3）缺点

与其他算法一样，它同时存在缺点，主要有：

（1）牺牲了决策树的可解释性；

（2）在某些噪声较大的分类或回归问题上会过拟合；

（3）在多个分类变量的问题中，随机森林可能无法提高基学习器的准确性。

3. 随机森林的 Python 实现

前面对随机森林算法及概念进行了简单的介绍，下面直接通过例子来演示 Python 的实现。

【例 3-14】 利用 Python 的两个模块，分别为 pandas 和 scikit-learn 来实现随机森林。

```
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
import pandas as pd
import numpy as np

iris = load_iris()
df = pd.DataFrame(iris.data, columns = iris.feature_names)
df['is_train'] = np.random.uniform(0, 1, len(df)) <= 0.75
#在新版本的 pandas 中,Factor 被替换为 Categorical,因此,pd.Factor 改为 pd.Categorical.
#from_codes
df['species'] = pd.Categorical.from_codes(iris.target, iris.target_names)
df.head()
train, test = df[df['is_train'] == True], df[df['is_train'] == False]

features = df.columns[:4]
clf = RandomForestClassifier(n_jobs = 2)
y, _ = pd.factorize(train['species'])
clf.fit(train[features], y)
preds = iris.target_names[clf.predict(test[features])]
pd.crosstab(test['species'], preds, rownames = ['actual'], colnames = ['preds'])
```

运行程序，得到分类结果如图 3-24 所示。

preds actual	setosa	versicolor	virginica
setosa	12	0	0
versicolor	0	13	1
virginica	0	1	13

图 3-24 分类结果

【例 3-15】 随机森林与其他机器学习分类算法进行对比。

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
```

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as QDA

plt.rcParams['font.sans-serif'] = ['SimHei']                    #设置中文字体
h = .02                                                         #网格中的步

names = ["最近邻", "线性 SVM", "高斯核 SVM", "决策树",
         "随机森林", "Adaboost", "贝叶斯", "LDA", "QDA"]
classifiers = [
     KNeighborsClassifier(3),
     SVC(kernel="linear", C=0.025),
     SVC(gamma=2, C=1),
     DecisionTreeClassifier(max_depth=5),
     RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1),
     AdaBoostClassifier(),
     GaussianNB(),
     LDA(),
     QDA()]

X, y = make_classification(n_features=2, n_redundant=0, n_informative=2,
                           random_state=1, n_clusters_per_class=1)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

datasets = [make_moons(noise=0.3, random_state=0),
            make_circles(noise=0.2, factor=0.5, random_state=1),
            linearly_separable
            ]
figure = plt.figure(figsize=(27, 9))
i = 1
#在数据集上迭代
for ds in datasets:
     #预处理数据集,分为训练部分和测试部分
     X, y = ds
     X = StandardScaler().fit_transform(X)
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4)

     x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
     y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
     xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                          np.arange(y_min, y_max, h))

     #只要先绘制数据集即可
     cm = plt.cm.RdBu
     cm_bright = ListedColormap(['#FF0000', '#0000FF'])
     ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
     #绘制训练数据点
     ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright)
     #绘制测试点
     ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.6)
     ax.set_xlim(xx.min(), xx.max())
```

```
        ax.set_ylim(yy.min(), yy.max())
        ax.set_xticks(())
        ax.set_yticks(())
        i += 1

        #在分类器上迭代
        for name, clf in zip(names, classifiers):
            ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
            clf.fit(X_train, y_train)
            score = clf.score(X_test, y_test)

            #绘制决策边界,并为网格中[x_min,m_max]x[y_min,y_max]的每个点分配一种颜色
                if hasattr(clf, "decision_function"):
                Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
            else:
                Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

            # Put the result into a color plot
            Z = Z.reshape(xx.shape)
            ax.contourf(xx, yy, Z, cmap=cm, alpha=0.8)

            #将结果放到一个颜色绘图中
            ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright)
            #同时绘制训练点
            ax.sand testing pointsX_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
                        alpha=0.6)

            ax.set_xlim(xx.min(), xx.max())
            ax.set_ylim(yy.min(), yy.max())
            ax.set_xticks(())
            ax.set_yticks(())
            ax.set_title(name)
            ax.text(xx.max() - 0.3, yy.min() + 0.3, ('%.2f' % score).lstrip('0'),
                      size=15, horizontalalignment='right')
            i += 1

    figure.subplots_adjust(left=0.02, right=0.98)
    plt.show()
```

运行程序,效果如图 3-25 所示。

这里随机生成了三个样本集,分割面近似为月形、圆形和线形。可以重点对比决策树和随机森林对样本空间的分割。

(1) 从准确率上可以看出,随机森林在这 3 个测试集上都要优于单棵决策树,90%>85%,82%>80%,95%=95%。

(2) 从特征空间上直观地可以看出,随机森林比决策树拥有更强的分割能力(非线性拟合能力)。

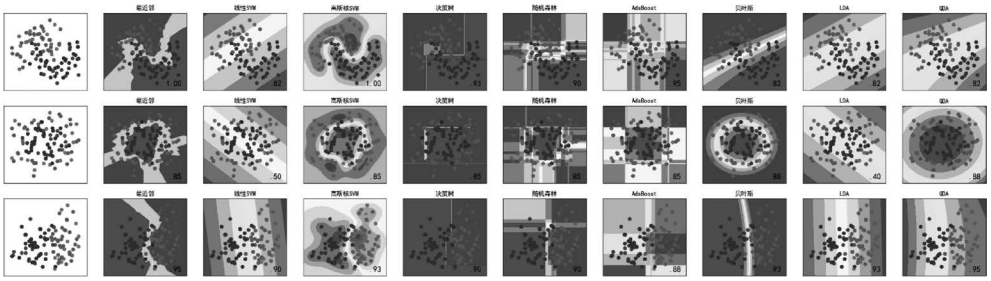

图 3-25　各方法的分类对比情况

3.4　数据预处理

常见的不规则数据主要有缺失数据、重复数据、异常数据几种，在开始正式的数据分析之前，我们需要先把这些不太规整的数据处理掉，做数据预处理。

3.4.1　数据预处理概述

数据预处理常遇到数据存在噪声、冗余、关联性、不完整性等。常见处理方法如下。

（1）数据清理：补充缺失值、消除噪声数据、识别或删除离群点（异常值）并解决不一致性。其目标是为了数据格式标准化、异常数据清除、重复数据清除、错误纠正。

（2）数据集成：将多个数据源中的数据进行整合并统一存储。

（3）数据变换：通过平滑数据、数据概率化及规范化等方式将数据转换为适用数据挖掘的形式。

（4）数据归约：针对在数据挖掘时，数据量非常大的问题，数据归约技术对数据集进行归约或简化，不仅保持原数据的完整性，且数据归约后的结果与归约前的结果相同或几乎相同。

3.4.2　数据清理

1. 异常数据处理

异常数据也称离群点，指采集的数据中个别数据值明显偏离其余观测值。应对这些数据采用一定的方法消除，否则对结果产生影响。

1）异常数据分析

异常数据分析的原则主要有：

（1）统计量判断：最大值、最小值、均值等，检查数据是否超出合理范围。

（2）3σ 原则：根据正态分布定义，出现距离均值 3σ（3 倍标准差）以上的数值属于小概率事件，此时，数据和均值的偏差超过 3σ（3 倍标准差）的视为异常值。

（3）箱型图判断：箱型图反应数据分布。如果数据超出箱型图上界或下界视为异常数据。

2）异常数据处理方法

异常数据处理方法主要有：

（1）删除：直接把存在的异常数据删除，不进行考虑。

（2）视为缺失值：按照缺失值处理方法进行操作。

（3）不处理：看作正常数据处理、

（4）平均值修正：使用前后两个观测值的均值代替，或使用整个数据集的均值。

2. 缺失值处理

经常使用数据补插值方法替代缺失值，如以下几种。

（1）最近邻插值：使用缺失值附近样本或其他样本代替，或前后数据均值代替。

（2）回归方法：建立拟合模型，用该模型预测缺失值。

（3）插值法：类似回归法，利用已知数据建立插值函数，用该函数计算近似值代替缺失值。常见插值函数有拉格朗日插值法、牛顿插值法、分段插值法、样条插值法、Hermite插值法。

3. 噪声数据处理

噪声数据处理方法主要有分箱法、聚类法、回归法等。

1）分箱法

把待处理数据（某列属性值）按一定规则放进一些箱子（区间），考查每一个区间里的数据，然后采用某方法分别对各个区间数据处理。需考虑2个问题：如何分箱、如何对每个箱子中数据进行平滑处理。分箱法如下几种。

（1）等深分箱法（统一权重法）：将数据集按记录（行数）分箱，每箱具有相同的记录数（元素个数）。每箱记录数称为箱子深度（权重）。

（2）等宽分箱法（统一区间法）：使数据集在整个属性值的区间上平均分布，即每个箱的区间范围（箱子宽度）是一个常量。

（3）用户自定义区间：当用户明确希望观察某些区间范围内的数据时，可根据需要自定义区间。

分箱后对每个箱子中数据进行平滑处理。常见数据平滑方法如下。

（1）按均值平滑：对同一箱子中数据求均值，用均值代替该箱子中所有数据。

（2）按边界值平滑：用距离较小的边界值代替每一个数据。

（3）按中值平滑：取箱子中数据的中值，代替箱子中所有数据。

2）聚类法

聚类法是将物理或抽象对象的集合分组为由类似对象组成的多个类，然后展出并清除那些落在簇之外的值（孤立点），这些孤立点称为噪声数据。

3）回归法

回归法即试图发现两个变量之间的相关变化模式，找到这个函数来平滑数据，即通过建立一个数学模型来预测下一个数值，一般包括线性回归和非线性回归。

3.4.3 数据集成

数据集成是将多文件或数据库中的异构数据进行合并，然后存放在一个统一数据库

中存储。需考虑的问题：

(1) 实体识别：数据来源不同，其中概念定义不同。

- 同名异义：数据源 A 某个数据特征名称与数据源 B 某个数据特征名称相同，但表示内容不同。
- 异名同义：数据源 A 某个数据特征名称与数据源 B 某个数据特征名称不同，但表示内容相同。
- 单位不统一：不同的数据源记录单位不同。如身高同时用米和厘米表述。

(2) 冗余属性。

- 同一属性多个数据源均有记录。
- 同一属性命名不一致引起数据重复。

(3) 数据不一致：编码使用不一致、数据表示不一致，如日期。

3.4.4　数据变换

数据变换是将数据转换为适合机器学习的形式。

1. 使用简单的数学函数对数据进行变换

如果数据较大，可对数据取对数、开方等，使数据压缩变小；如果数据较小，对数据进行平方处理，扩大数据。如果数据为时间序列，可对序列进行对数变换或差分运算，使非平稳序列转换为平稳序列。

2. 归一化(数据规范化)

如果比较工资收入，有人每月工资上万元，有人每月才几百元，归一化可消除数据间这种量纲影响。

(1) 最小最大归一化(离散标准化)。

对原始数据进行线性变换，使其映射到[0,1]，转换函数如下：

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

(2) Z-score 标准法(零-均值规范法)。

使用原始数据均值和标准差，对数据标准化。经过处理的数据符合标准正态分布，即均值为 0，标准差为 1。转换函数如下：

$$x' = \frac{x - \mu}{\sigma}$$

其中，$x-\mu$ 为离均差，σ 表示总体标准偏差。但值得注意的是，做归一化的原始数据近似为高斯分布(正态分布)，否则，归一效果不理想。

(3) 小数定标规范化。

通过移动数据的小数点位置进行规范化。

$$x' = \frac{x}{10^x}$$

3. 连续属性离散化

连续属性离散化本质上是将连续属性空间划分为若干区间，最后用不同的符号或整

数值代表每个子区间中的数据。离散化涉及两个子任务：确定分类和将连续属性值映射到这些分类中。

机器学习常用离散化方法如下。

(1) 等宽法：将数据划分为具有相同宽度的区间，将数据按照值分配到不同区间，每个区间用一个数值表示。

(2) 等频法：把数据划分为若干区间，按照其值分配到不同区间，每个区间内数据个数相等。

(3) 基于聚类分析的方法——典型算法 K-means 算法：

首先，从数据集中随机找出 K 个数据作为 K 个聚类的中心。

其次，根据其他数据相对于这些中心的欧氏距离，对所有对象聚类。如果数据点 x 距某个中心近，则将 x 划归该中心所代表的聚类。

最后，重新计算各区间中心，利用新中心重新聚类所有样本。循环直至所有区间中心不再随算法循环而改变。

3.4.5 数据归约

在尽可能保持数据原貌前提下，最大限度精简数据量。与非归约数据相比，在归约的数据上进行挖掘，所需时间和内存资源更少，挖掘更有效，产生几乎相同分析结果。常用维归约(特征规约)、数值归约等方法。

1. 维归约(特征归约)

维归约(特征归约)通过减少属性特征方式压缩数据量，移除不相关属性，提高模型效率。维归约方法常用的几种如下。

(1) AIC 准则：通过选择最优模型来选择属性。

(2) LASSO：通过一定约束条件选择变量。

(3) 分类树、随机森林：通过对分类效果的影响大小筛选属性。

(4) 小波变换、主成分分析：通过把原数据变换或投影到较小空间来降低维数。

2. 数值归约(样本归约)

数值归约(样本归约)从数据集中选出一个有代表性的样本子集。子集大小的确定需考虑计算成本、存储要求、估计量精度、其他与算法有关因素。

(1) 参数法中使用模型估计数据，可只存放模型参数代替存放实际数据。例如，回归模型、对数线性模型。

(2) 非参数方法可使用直方图、聚类、抽样、数据立方体聚焦等。

3.4.6 Python 的数据预处理函数

Pandas 使用浮点值 NaN 代表缺失数据，NumPy 使用 NaN 代表缺失值，Python 内置的 None 也会被当作 NA 处理。

```
from pandas import Series,DataFrame
from numpy import nan as NA
data = Series([12,None,34,NA,58])
```

```
print(data)
0    12.0
1     NaN
2    34.0
3     NaN
4    58.0
dtype: float64
```

还可使用 isnull 函数检测是否为缺失值，该函数返回一个布尔型数组，一般可用于布尔型索引。

```
print(data.isnull())
0    False
1     True
2    False
3     True
4    False
dtype: bool
```

当数据中存在缺失值时，常用以下方法处理。

(1) 数据过滤(dropna)。

数据过滤即直接将缺失值数据过滤掉，不再考虑。数据过滤对 Series 结构没太大问题；对 DataFrame 结构，如果过滤掉，至少丢掉包含缺失值所在的一行或一列。dropna 函数的语法格式为：

```
dropna(axis = 0,how = 'any',thresh = None)
```

其中，axis=0 表示行，axis=1 表示列；how 参数可选值为 all/any，all 表示丢掉全为 NA 的行；thresh 为整数类型，表示删除条件，如 thresh=3，表示一行中至少有 3 个。

提示：*数据为非 NA 值才将其保留。*

```
from pandas import Series,DataFrame
from numpy import nan as NA
data = Series([12,None,34,NA,58])
print(data.dropna())
0     12.0
2     34.0
4     58.0
dtype: float64
```

接下来看一个两维数据的情况：

```
from pandas import Series,DataFrame
from numpy import nan as NA
import numpy as np
data = DataFrame(np.random.randn(5,4))
data.loc[:2,1] = NA
```

```
data.loc[:3,2] = NA
print('--- 删除前的结果是 --- ')
print(data)

print('--- 删除后的结果是 --- ')
print(data.dropna(thresh = 2))
print(data.dropna(thresh = 3))
```

运行程序,输出如下:

```
--- 删除前的结果是 ---
          0             1             2             3
0 -0.754303           NaN           NaN    -0.039454
1  0.163656           NaN           NaN     0.213773
2 -0.340330           NaN           NaN     0.284442
3  1.411648      0.778669           NaN     1.150938
4 -1.470341      0.425240      0.133391    -1.358089
--- 删除后的结果是 ---
          0             1             2             3
0 -0.754303           NaN           NaN    -0.039454
1  0.163656           NaN           NaN     0.213773
2 -0.340330           NaN           NaN     0.284442
3  1.411648      0.778669           NaN     1.150938
4 -1.470341      0.425240      0.133391    -1.358089
          0             1             2             3
3  1.411648      0.778669           NaN     1.150938
4 -1.470341      0.425240      0.133391    -1.358089
```

(2) 数据填充(fillna)。

在 Pandas 中可以利用 fillna 对数据进行填充。函数的语法格式为:

```
fillna(value,method,axis)
```

其中,value 除了基本类型外,还可使用字典实现对不同列填充不同值; method 为采用填补数值的方法,默认 None。

下面代码用 0 代替所有 NaN:

```
print(data.fillna(0))
0    12.0
1     0.0
2    34.0
3     0.0
4    58.0
dtype: float64
```

使用字典填充:第 1 列缺失值用 11 代替,第 2 列缺失值用 22 代替。

```
print(data.fillna({1: 11,2: 22}))
0    12.0
1    11.0
```

```
2    34.0
3     NaN
4    58.0
dtype: float64
```

第 1 列缺失值用该列均值代替，第 2 列同理。

```
print(data.fillna({1: data[1].mean(),2: data[2].mean()}))
0    12.0
1     NaN
2    34.0
3     NaN
4    58.0
dtype: float64
```

（3）拉格朗日插值法。

```
from scipy.interpolate import lagrange                          #导入拉格朗日插值函数
import pandas as pd
from pandas import DataFrame
import numpy as np

df = DataFrame(np.random.randn(20,2),columns = ['first','second'])

df[(df['first']<-1.5)|(df['first']>1.5)] = None                 #将异常值变为空值
print(df)
```

运行程序，输出如下：

```
        first     second
 0  -0.264102   1.226255
 1  -0.284215   1.346911
 2  -1.175365   0.539050
 3  -0.211221   0.547658
 4  -1.305737  -0.043319
 5   0.479445  -0.487864
 6  -0.259229  -0.232353
 7   0.735482  -0.530868
 8   1.070011  -1.326933
 9  -0.074371  -1.345669
10        NaN        NaN
11  -1.458927   0.982368
12   0.936049   1.640186
13   0.311783  -1.091818
14   0.105756  -0.572328
15  -0.393785   0.158342
16  -0.832418   0.422542
17  -1.120591  -0.917236
18  -0.561338   0.293409
19   0.764413   1.232289
```

自定义列向量插值函数。*s* 为列向量，*n* 为被插值位置，*k* 为取前后的数据个数，默认为 5，代码如下：

```
from scipy.interpolate import lagrange                    #导入拉格朗日插值函数
import pandas as pd
from pandas import DataFrame
import numpy as np

df = DataFrame(np.random.randn(20,2),columns = ['first','second'])
df[(df['first']<-1.5)|(df['first']>1.5)] = None           #将异常值变为空值
print(df)
```

运行程序，输出如下：

```
        first      second
 0   0.273606 -0.920961
 1   0.923742 -0.327123
 2   0.743703 -1.147971
 3  -0.396132 -0.736615
 4   1.039964   0.127254
 5   0.013551 -0.122106
 6   1.129434 -0.687094
 7  -0.991091   1.103130
 8   0.115933   1.329474
 9   0.408046 -0.426811
10   1.176634   1.265182
11  -0.817599   0.216728
12  -0.361764   0.697742
13   1.464341 -2.171481
14   0.735621 -0.214796
15   1.306358 -0.494527
16  -0.174018   0.163075
17        NaN        NaN
18   1.056959   0.064894
19   0.227799   0.031293
```

(4) 检测和过滤异常值(outlier)。

```
from pandas import Series, DataFrame
import pandas as pd
from numpy import nan as NA

data = DataFrame(np.random.randn(10,4))
print(data.describe())
print('\n...找出某一列中绝对值大小超过 1 的项...\n')
data1 = data[2]                          #data 的第 2 列赋值给 data1
print(data1[np.abs(data1)>1])
data1[np.abs(data1)>1] = 100             #将绝对值大于 1 的数据修改为 100
print(data1)
```

运行程序，输出如下：

```
                0             1             2             3
count   10.000000     10.000000     10.000000     10.000000
mean    -0.655747      0.102771     -0.149770      0.176366
std      1.005762      1.216042      0.977085      1.074940
min     -2.267146     -2.683228     -1.890334     -1.406471
25%     -1.402271     -0.168816     -0.786397     -0.711641
50%     -0.567194     -0.098740     -0.075330      0.394950
75%      0.115545      0.835951      0.619358      0.904470
max      1.161726      1.704885      1.240517      1.888549

...找出某一列中绝对值大小超过1的项...

2       1.240517
6      -1.890334
Name: 2, dtype: float64
0       0.127491
1      -0.278152
2     100.000000
3       0.906773
4      -0.713616
5      -0.997429
6     100.000000
7       0.779863
8      -0.810658
9       0.137843
Name: 2, dtype: float64
```

(5) 移除重复数据。

在 Pandas 中使用 duplicated 方法发现重复值(返回 bool 型数组,F 表示非重复,T 表示重复),使用 drop_duplicates 方法移除重复值。

```
from pandas import Series, DataFrame
import pandas as pd
from numpy import nan as NA
import pandas as pd
import numpy as np

data = DataFrame({'name': ['zhang'] * 3 + ['wang'] * 4,'age': [18,18,19,19,20,20,21]})
print(data)
print('...重复的内容是...\n')
print(data.duplicated())
print('-- 删除重复的内容后 --- ')
print(data.drop_duplicates())
```

运行程序,输出如下:

```
    name  age
0  zhang  18
1  zhang  18
2  zhang  19
```

```
3   wang  19
4   wang  20
5   wang  20
6   wang  21
...重复的内容是...

0    False
1     True
2    False
3    False
4    False
5     True
6    False
dtype: bool
-- 删除重复的内容后 ---
    name  age
0  zhang   18
2  zhang   19
3   wang   19
4   wang   20
6   wang   21
```

duplicated 和 drop_duplicates 默认保留第一个出现值组合，可修改参数 keep＝'last' 保留最后一个。

```
print('-- 删除重复的内容后 --- ')
print(data.drop_duplicates(keep = 'last'))
```

运行程序，输出如下：

```
-- 删除重复的内容后 ---
    name  age
1  zhang   18
2  zhang   19
3   wang   19
5   wang   20
6   wang   21
```

(6) 数据规范化。

为消除数据之间量纲影响，需对数据进行规范化处理，将数据落入一个有限范围。常见归一化将数据范围调整到[0,1]或[－1,1]。一般使用方法：最小最大规范化、零均值规范化和小数规范化。

假设属性 income 的最小值和最大值分别是 5000 元和 58 000 元。利用 Min-Max 规范化的方法将属性的值映射到[0,1]范围内，那么属性 income 的 16 000 元将被转换为多少？

```
from sklearn import preprocessing
import numpy as np
```

```
x = np.array([[5000.],[58000.],[16000.]])
min_max_scaler = preprocessing.MinMaxScaler()
minmax_x = min_max_scaler.fit_transform(x)
print(minmax_x)
```

运行程序，输出如下：

```
[[0.        ]
 [1.        ]
 [0.20754717]]
```

（7）汇总和描述等统计量的计算。

汇总和描述等统计量的计算分别有最大值、最小值、方差、标准差（standard deviation）等。

```
import pandas as pd
import numpy as np
from pandas import Series,DataFrame

df = DataFrame(np.random.randn(4,3),index = list('abcd'),columns = ['first','second','third'])
print(df.describe())                                          #对数据统计量进行描述
```

运行程序，输出如下：

```
            first        second         third
count    4.000000      4.000000      4.000000
mean    -0.042684     -0.085264     -0.288354
std      1.301102      0.727818      1.490419
min     -1.552620     -0.926142     -1.950211
25%     -0.869810     -0.572914     -1.239619
50%      0.021231     -0.006652     -0.318090
75%      0.848358      0.480997      0.633175
max      1.339422      0.598391      1.432977
print('统计每列数据的和：',df.sum())                          #统计每列数据的和
print('统计每行(从左到右)数据和：',df.sum(axis = 1))          #统计每行(从左到右)数据和
print('统计每列最小数值所在的行：',df.idxmin())              #统计每列最小数值所在的行
print('统计每行最小数值所在的列：',df.idxmin(axis = 1))      #统计每行最小数值所在的列
print('计算相对于上一行的累计和：',df.cumsum())              #计算相对于上一行的累计和
print('方差：',df.var())                                     #计算方差
print('标准差：',df.std())                                   #计算标准差
print('百分数变化：',df.pct_change())                        #计算百分数变化
统计每列数据的和：first     -0.170735
```

运行程序，输出如下：

```
second    -0.341056
third     -1.153415
dtype: float64
统计每行(从左到右)数据和：a     2.147863
```

```
b     2.716037
c    -3.047589
d    -3.481517
dtype: float64
统计每列最小数值所在的行: first     d
second     d
third      c
dtype: object
统计每行最小数值所在的列: a       third
b     second
c      third
d      first
dtype: object
计算相对于上一行的累计和:         first    second     third
a     1.339422    0.441866     0.366574
b     2.024092    1.040257     1.799551
c     1.381885    0.585086    -0.150660
d    -0.170735   -0.341056    -1.153415
方差: first     1.692865
second          0.529719
third           2.221349
dtype: float64
标准差: first     1.301102
second            0.727818
third             1.490419
dtype: float64
百分数变化:          first    second     third
a         NaN          NaN         NaN
b   -0.488832     0.354234    2.909104
c   -1.937981    -1.760658   -2.360951
d    1.417632     1.034713   -0.485822
```