Python
机器学习与量化投资

何海群 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书采用生动活泼的语言，从入门者的角度，讲解了 Python 语言和 sklearn 模块库内置的各种经典机器学习算法；介绍了股市外汇、比特币等实盘交易数据在金融量化方面的具体分析与应用，包括对未来股票价格的预测、大盘指数趋势分析等。简单风趣的实际案例让广大读者能够快速掌握机器学习在量化分析方面的编程，为进一步学习金融科技奠定扎实的基础。

图书在版编目（CIP）数据

Python 机器学习与量化投资 / 何海群著. —北京：电子工业出版社，2018.12
（金融科技丛书）
ISBN 978-7-121-35210-2

Ⅰ. ①P… Ⅱ. ①何… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字（2018）第 238872 号

策划编辑：黄爱萍
责任编辑：张彦红
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：720×1000　1/16　印张：18.75　字数：270 千字
版　　次：2018 年 12 月第 1 版
印　　次：2018 年 12 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

前　言

本书特色

本书全程采用黑箱模式和 MBA 案例模式，结合大量经典案例，介绍 sklearn 机器学习模块库和常用的机器学习算法，懂 Excel 就能看懂本书；逆向式课件模式，结合大量案例、图表，层层剖析；三位一体的课件模式：图书+开发平台+成套的教学案例，系统讲解、逐步深入。

本书是《零起点 Python 机器学习快速入门》的后续之作，为了节省篇幅，省略了 Python 基础教程，以及 sklearn 等机器学习方面的入门内容，没有经验的读者，建议先阅读《零起点 Python 机器学习快速入门》，再阅读本书，这样会收到事半功倍的效果。

本书简单实用，书中配备大量的图表说明，本书特点如下。

- IT 零起点：无须任何电脑编程基础，只要会打字、会使用 Excel，就能看懂本书，利用本书配套的 Python 软件包，轻松学会如何利用 Python 对股票数据进行专业分析和量化投资分析。
- 投资零起点：无须购买任何专业软件，本书配套的 zwPython 软件包，采用开源模式，提供 100%全功能、全免费的工业级数据分析平台。

- 配置零起点：所有软件、数据全部采用 “开箱即用”模式，绿色版本，无须安装，解压缩后即可直接运行系统。
- 理财零起点：采用通俗易懂的语言，配合大量专业的图表和实盘操作案例，无须任何专业金融背景，轻松掌握各种量化投资策略。
- 数学零起点：全书没有任何复杂的数学公式，只有最基本的加、减、乘、除，轻轻松松就能看懂全书。

网络资源

本书的案例程序，已经做过优化处理，无须 GPU 显卡，全部支持单 CPU 平台，不过为避免版本冲突，请尽量使用 zwPython2017m6 版本运行本书的案例程序。

使用其他运行环境的读者，如 Linux、Mac 平台的用户，请尽量使用 Python 3 版本，自行安装其他所需的模块库，如 Numpy、Pandas、Tushare 等第三方模块库。

此外需要注意的是，大家运行本书案例得到的结果可能与书中略有差别；甚至多次运行同一案例，结果都有所差异。这属于正常情况，因为很多机器学习函数，内部使用了随机数作为种子数，用于系统变量初始化等操作，每次分析的起点或者中间参数会有所不同。

版本冲突是开源项目常见的问题，为了解决这个问题，本书的源码是独立保存的。此外，我们还特意设计了 zwPython 教学版。

建议初学者先使用 zwPython 教学版，有关的课件程序，已经经过版本兼容测试，并且集成了 zwDat 金融数据集。

本书的读者 QQ 互动群：QQ 1 群的群号是 124134140；QQ 2 群的群号是 650924099；QQ 3 群的群号是 450853713。

资源下载地址：TopQuant 极宽量化网站“资源中心”。

请浏览以下网站，获取最新的网络资源地址：

- TopQuant.vip 极宽量化社区
- ziwang.com 字王网站

目前两个网站的指向都是一样的。

另外还可以在博文视点网站下载：http://www.broadview.com.cn。

目录设置

为运行本书课件程序，用户需要下载以下三个软件，并设置好目录：

- zwPython，必须放在根目录，是 Python 开发平台，为避免版本冲突，请尽量使用 zwPython2017m6 版本。
- kb_demo，本书 sklearn 机器学习配套课件源码。
- pg_demo，本书 Python 入门学习配套课件源码。

以上软件、程序最好保存在固态硬盘，这样速度会快很多；目录名称不要使用中文名称，压缩文件当中的中文名称只是为了便于用户下载。

zwPython 开发平台必须放在根目录，课件程序可以放在其他自定义目录，建议放在 zwPython 目录下面，作为二级目录。

致谢

特别感谢电子工业出版社的黄爱萍和陈林编辑在选题策划和稿件整理方面所做的大量工作。

同时，在本书创作过程中，极宽开源量化团队和培训班的全体成员提出很多宝贵的意见，并对部分课件程序做了中文注解。

特别是吴娜、余勤、邢梦来、孙励、王硕几位成员，为 TOP 极宽开源量化文库和开源软件编写文档，以及在团队成员管理方面做了大量工作，对他们的付出表示感谢。

何海群（字王）

TOP 极宽量化开源组发起人

2018 年 10 月 1 日

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源**：本书如提供示例代码及资源文件，均可在 下载资源 处下载。
- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：*http://www.broadview.com.cn/35210*

目　　录

第 1 章

Python 与机器学习

目前神经网络、深度学习大热，谷歌、脸书、微软、IBM、亚马逊等企业巨头，纷纷投入巨资，各种深度学习的开发平台层出不穷：TensorFlow、PyTorch、CNTK、MXNet、Keras 等。

与此同时，Python 语言成为人工智能第一开发语言。

在传统的机器学习领域，或者说古典人工智能领域却波澜不惊，scikit-learn 始终居于王者般的统治地位。

图 1.1 是 scikit-learn 网站首页截图，在网站首页抬头右侧，写着：

Machine Learning in Python（Python 中的机器学习）

换句话说：

scikit-learn=Python 机器学习

图 1.1　scikit-learn 网站首页截图

1.1 scikit-learn模块库

scikit-learn，简称 sklearn，是用 Python 开发的机器学习模块库，其中包含大量机器学习算法、数据集。

scikit-learn 模块库优点很多：简单易用、API 接口完善、案例文档丰富等。内置大量经过筛选的、高质量的机器学习模型；模块库覆盖了大多数机器学习任务；系统可扩展至较大的数据规模。

在 scikit-learn.org 网站首页抬头，scikit-learn 官方自我总结的优点如下。

- 简单高效的数据挖掘和数据分析工具。
- 可在各种环境中重复使用。
- 建立在 NumPy、SciPy 和 Matplotlib 上。
- 开放源码，可免费商业使用 BSD license。

scikit-learn 模块库是老牌的开源 Python 机器学习算法框架，源于 2007 年谷歌公司的 Google Summer of Code 项目，最早由数据科学家 David Cournapeau 发起，是 Python 语言中专门针对机器学习应用而发展起来的一款开源框架。

scikit-learn 模块库是一个简洁、高效的算法库，提供一系列的监督学习和无监督学习的算法，以用于数据挖掘和数据分析。scikit-learn 几乎覆盖了机器学习的所有主流算法，这为其在 Python 开源世界中奠定了江湖地位。

scikit-learn 的算法库是建立在 SciPy（Scientific Python）之上的，这也是其命名的由来。

SciPy 模块库是 Python 语言的基础科学计算工具包，基于 SciPy，目前开发者们针对不同的应用领域，已经发展出了众多的分支版本，SciPy 的扩展和模块在传统上被命名为 Scikits，即 SciPy 工具包的意思。

和其他众多的开源项目一样，scikit-learn 目前主要由社区成员自发维护。可能是由于维护成本的限制，scikit-learn 相比其他项目要显得更为保守。

这种保守主要体现在两个方面。

- scikit-learn 从来不做除机器学习领域之外的其他扩展。
- scikit-learn 从来不采用未经广泛验证的算法。

1.1.1　scikit-learn 的缺点

虽然 scikit-learn 模块库功能强大，目前已经是机器学习最重要的模块库，但是，scikit-learn 也有缺点。

scikit-learn 模块库的缺点主要包括以下几个方面。

- 不支持深度学习和强化学习。
- 不支持 PyPy 加速，也不支持 GPU 加速。
- 不支持除 Python 之外的其他编程语言。

这些缺点主要是由历史原因造成的，scikit-learn 毕竟是 2007 年的作品，已经有超过十年的历史，其整体架构不适用于目前的 GPU 编程、神经网络和深度学习。不过这些缺点都不是大问题。

在深度学习、神经网络领域，目前有众多的优秀平台：TensorFlow、MXNet、CNTK、PyTorch 等，scikit-learn 模块库与这些平台配合，主要用于数据预处理和结果验证。

GPU 加速和 PyPy 优化属于 Python 底层优化，特别是 NumPy 基础科学计算库的优化，目前基于 GPU 的优化版本已经不断涌现，其中 MXNet、PyTorch 底层模块库，都是基于 NumPy 的 GPU 优化版本。

从工程角度而言，scikit-learn 的性能表现是非常不错的。

究其原因，一方面是因为其内部算法的实现十分高效，另一方面或许可以归功于 Cython 编译器：通过 Cython 在 scikit-learn 框架内部生成 C 语言代码的运行方式，scikit-learn 消除了大部分的性能瓶颈。

至于最后一个缺点，scikit-learn 模块库不支持除 Python 之外的其他编

程语言。这更加不是问题，目前 Python 已经是人工智能、机器学习领域的标准编程语言，强如 Facebook，坚持多年，最终还是把 Lua 语言开发的 Torch 项目，全部采用 Python 改写，并重新命名为 PyTorch 项目。

1.1.2 scikit-learn 算法模块

为了方便学习，scikit-learn 开发团队还提供了一个 scikit-learn 算法模块图，如图 1.2 所示。

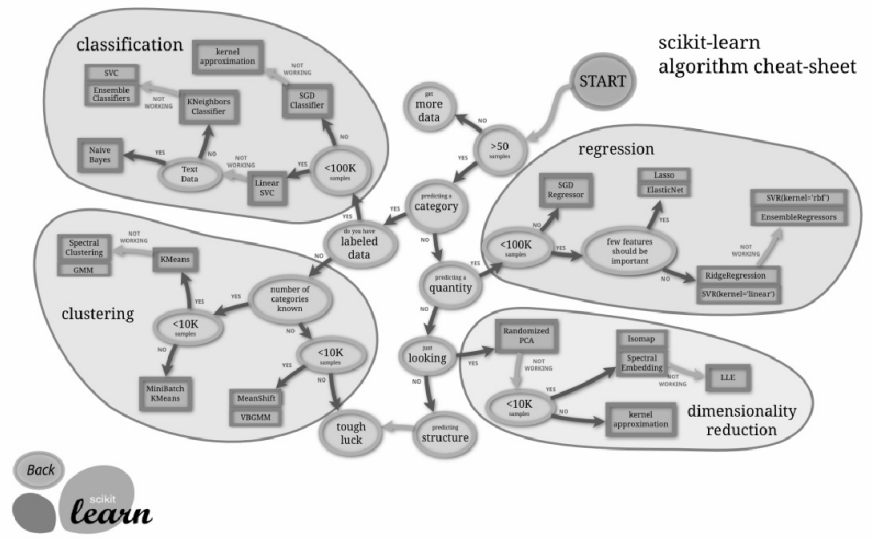


图 1.2　scikit-learn 算法模块图

图 1.3 是 scikit-learn 算法模块图的汉化版本。

scikit-learn 模块库实现了一整套用于数据降维、模型选择、特征提取和归一化的完整算法和模块，并且针对每个算法和模块，底层都进行了高度优化以提升速度，与此同时，模块库提供了丰富的参考案例和详细的说明文档。

这些案例程序，内容覆盖全面，讲解细致，并且很多案例都使用了真实的数据，绝大多数案例还配有 Matplotlib 绘制的数据图表。

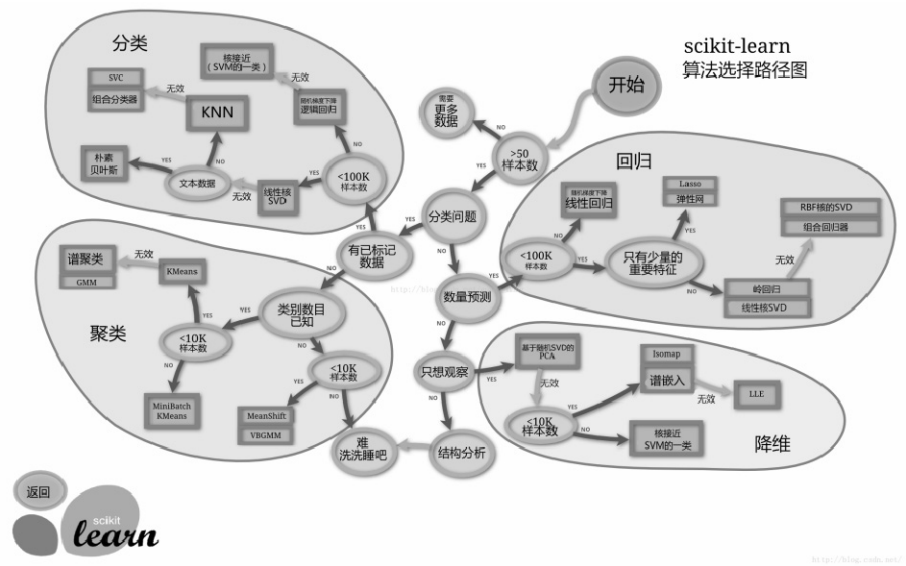


图 1.3　scikit-learn 算法模块图汉化版

据官方的统计，scikit-learn 模块库共提供了 200 多个参考案例程序，包括统计学习、监督学习、模型选择和无监督学习等若干部分。

1.1.3　scikit-learn 六大功能

scikit-learn 机器学习模块库，功能强大，模块繁多。模块库的主要功能包括以下六大类别：分类、回归、聚类、数据降维、模型选择和数据预处理。

需要说明的是，随着近年神经网络、深度学习的崛起，scikit-learn 机器学习模块库在最新版本的模型库中增加了 MLP 多层感知器等最基本的神经网络模型。

不过，scikit-learn 本身不支持深度学习，也不支持 GPU 加速，因此其内置的 MLP 等模型仅用于教学和小规模数据应用。

图 1.4 是 scikit-learn 模块功能图，也是 scikit-learn 网站首页截图，图 1.5 是 scikit-learn 模块功能图的汉化版本。

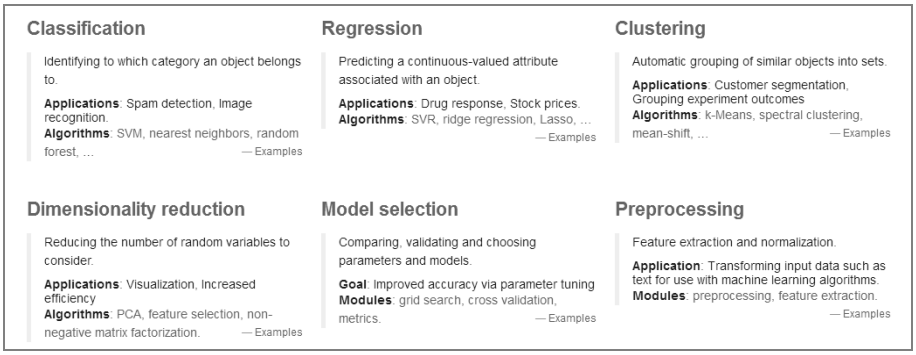


图 1.4　scikit-learn 模块功能图

图 1.5　scikit-learn 模块功能图汉化版

1．分类

分类是指识别某个对象属于哪个类别，属于监督学习的范畴，最常见的应用场景包括垃圾邮件检测和图像识别等。

目前 scikit-learn 已经实现的分类算法包括：SVM（支持向量机）、Nearest Neighbors（最近邻）、逻辑回归、Random Forest（随机森林）、决策树，以及多层感知器、MLP 神经网络等。

2．回归

回归是指预测与对象相关联的连续值属性，最常见的应用场景包括预测药物反应和股票价格等。

目前 scikit-learn 已经实现的回归算法包括：SVR（支持向量回归）、Ridge Regression（脊回归）、Lasso 回归、Elastic Net（弹性网络）、LARS（最小

角回归）、贝叶斯回归，以及各种不同的鲁棒回归算法等。

可以看到，scikit-learn 内置的回归算法几乎涵盖了所有开发者的需求，而且更重要的是，scikit-learn 针对每种算法都进行了底层优化加速，并且提供了简单明了的参考案例。

3．聚类

聚类是指将相似对象自动分组，也就是自动识别具有相似属性的给定对象，并将其分组为集合，属于无监督学习的范畴，最常见的应用场景包括顾客细分和试验结果分组。

目前 scikit-learn 已经实现的聚类算法包括：k-Means 聚类、Spectral Clustering（谱聚类）、Mean-shift（均值偏移）、分层聚类、DBSCAN 聚类等。

4．降维

数据降维是指减少要考虑的随机变量的个数，其主要应用场景包括可视化处理和效率提升。

目前 scikit-learn 已经实现的降维算法包括：PCA（主成分分析）、NMF（非负矩阵分解）、Feature Selection（特征选择）等降维技术。

5．模型选择

模型选择是指比较、验证、选择参数和模型。对于给定参数和模型进行比较、验证和选择，其主要目的是通过参数调整来提升精度。

目前 scikit-learn 实现的模型选择算法包括：Grid Search（网格搜索）、Cross Validation（交叉验证）和各种针对预测误差评估的 Metrics（度量）函数。

6．预处理

数据预处理是指数据的特征提取和归一化，是机器学习过程中的第一个环节，也是最重要的一个环节。

这里归一化是指将输入数据转换为具有零均值和单位权方差的新变

量，但因为大多数时候都做不到精确等于零，因此会设置一个可接受的范围，一般都要求落在 0～1 之间。而特征提取是指将文本或图像数据转换为可用于机器学习的数字变量。

需要注意的是，数据预处理环节的特征提取与数据降维中提到的特征选择完全不同。

这里的特征选择是指通过去除不变、协变或其他统计上不重要的特征量，来改进机器学习的一种方法。

目前 scikit-learn 实现的预处理算法包括预处理和特征提取。

1.2 开发环境搭建

1.2.1 AI 领域的标准编程语言：Python

Python 是最适合初学者的编程语言，也是目前 IT 行业入门简单、功能强大的工业级开发平台的语言。

Python 已经是机器学习、神经网络等人工智能开发领域的工业标准编程语言。

1．入门简单

任何熟悉 JavaScript 脚本、Visual Basic、C 语言、Delphi 的用户，通常一天即可学会 Python。

即使是不会编程的设计师、打字员，一周内也能熟练掌握 Python，学习难度绝对不会高于 Photoshop、五笔输入法，至少笔者和许多程序员一样，也不会使用五笔输入法。

2．功能强大

海量级的 Python 模块库，提供了 IT 行业最前沿的开发功能。

- 大数据：pandas 已经逐步超越 R 语言。

- CUDA：高性能计算，Python、C（C++）与 Fortran 是 NVIDIA 官方认可的 3 种编程语言，也是目前唯一适合 PC 平台的 CUDA 编程工具。
- 机器学习：scikit-learn、Theano、Pattern 是国际上热门的机器学习平台。
- 自然语言：NLTK，全球首选的自然语言处理平台；spaCy，工业级的 NLP 平台。
- 人脸识别：OpenCV，光流算法、图像匹配、人脸算法，简单优雅。
- 游戏开发：Pygame 提供图像、音频、视频、手柄、AI 等全套游戏开发模块库。
- 字体设计：Fontforge，是唯一商业级的字体设计开源软件，内置的脚本语言和底层核心的 Fonttools，都是基于 Python 而开发的。
- 电脑设计：blend、GIMP、Inkscape、MAYA、3DS 都内置或扩展了 Python 语言支持。

既然 Python 如此美好，而且是 100%免费的开源软件，学习 Python 的人越来越多，为什么 Python 始终还只是一种小众语言呢？

笔者认为，Python 的“大众化”之路，存在以下两个瓶颈。

- 配置：软件行业有一句俗话“搞懂了软件配置，就学会了一半”。对于 Python 和 Linux 等许多开源项目而言，80%的问题，都出现在配置方面，尤其是模块库的配置。
- OOP（面向对象程序设计）：大部分人都认为 Python 是一种“面向对象”的编程语言，而业界公认 OOP 的编程风格比较繁杂。

如果能够解决好以上两个问题，Python 的学习难度可以降低 90%，在应用领域可以瞬间提升数十倍效能，而且这种提升是零成本的。

1.2.2　zwPython：难度降低 90%，性能提高 10 倍

笔者在 WinPython 软件包的基础上，推出了“zwPython”——字王集

成式 Python 开发平台。

- 提出“零配置、零对象”研发理念，绿色软件封装模式，类似 Mac 开箱即用风格，无须安装，解压即可直接使用，还可以放入 U 盘，支持 Mob-App 移动式开发编程。
- 具有“外挂”式“核弹”级开发功能，内置很多功能强大的开发模块库，例如 OpenCV 视觉、人脸识别、CUDA 高性能 GPU 并行计算（OpenCL）、pandas 大数据分析、机器学习、NLTK 自然语言处理。
- 便于扩展，用户可以轻松增删相关模块库，全程智能配置，无须用户干预，就好像拷贝文件一样简单，而且支持 U 盘移动便携模式，真正实现了“一次安装，随处可用”。
- 针对中文开发文档缺乏、零散的不足，内置多部中文版 OpenCV、Fontforge 和 Python 入门教材。
- 内置多款中文开源 TrueType 字库。
- 大量示例脚本源码，涵盖 OpenCV、CUDA、OpenCL、Pygame 等。

如此种种只是为了便于 IT 行业外的用户能够“零起步”，快速入门，并且短时间内应用到生产环节当中去。

zwPython 前身是 zw2015sdk：字王智能字模设计平台，原设计目标是向广大设计师提供一款统一的、可编程的字体设计平台，以便于大家交流。

- 设计师、美工都是文艺青年、IT 小白，所以，简单是必需的，开箱即用也必须是标配。
- 由于设计师做设计，所以图像处理 PIL、Matplotlib 模块是必需的。
- 集成了 OpenCV 作为图像处理、匹配模块，自然也提供了机器学习功能。
- 字模处理数据量很大，属于大数据范畴，须集成 SciPy、NumPy 和 pandas 数据分析模块。
- 由于原生 Python 速度慢，所以增加了 PyCUDA、OpenCL 高性能 GPU 计算模块。

如此一而再，再而三地扩充，发现 zwPython 已经基本覆盖了目前 Python 和 IT 编程 90%的应用领域，因此又增加了部分模块，将 zwPython 扩展成为一个通用的、集成式 Python 开发平台。

1.2.3 “零对象”编程模式

虽然，很多人认为 Python 是一种“面向对象”的编程语言。

但对于初学者而言，把 Python 视为一种 BASIC 风格的、过程式入门语言，学习难度可以降低 90%，基本上学习一小时即可动手编写学习代码。

有人说，“面向对象”最大的好处是方便把人脑子搅乱。

Windows、Linux、UNIX、Mac OS X 内核都是 C 语言编写的。有一种系统是 C++写的内核，就是诺基亚的塞班系统，据说代码量很大，连他们自己的程序员都无法维护，最后就死掉了。

简而言之，“面向对象”风格的代码：一个字“繁”、两个字“繁繁”、三个字“繁繁繁”。

“零对象编程模式，用 BASIC 的方式学习 Python”，是笔者向 Python 编程语言的入门用户提出的一种全新的学习理论，一家之言，仅供参考。

Talk is cheap, show me the code！大家还是多多动手。

大家很容易理解“零配置”，下面关于“零对象”再补充几点。

- 不写“面向对象”风格的代码不等于不能使用，对于各种采用“对象”模式开发的模块库，我们仍然可以直接调用。
- 将 Python 视为非“面向对象”语言并非大逆不道，事实上，许多人认为，Python 也是一种类似 LISP 的“函数”编程语言。
- 笔者从事编程十多年，从未用过“面向对象”模式编写过一行“class”（类对象）代码，依然可以应对各种编程工作。
- 目前“面向对象”编程理论在业界仍然争论不休，入门者功力不够，最好避开强者间的火力杀伤。

- “面向对象”过于复杂，与“人生苦短，我用 Python”的优雅风格天生不合。

1.2.4 开发平台搭建

本节主要讲解 Python 开发环境和数据包的配置、应用流程方面的知识。

本书所有案例程序均采用纯 Python 语言开发，除特别指明外，均默认使用 Python 3 语法，且经过 zwPython 平台测试。

zwPython 是极宽公司推出的一个 Python 集成版本，功能强大，属于免费开源软件。系统内置了数百种专业的 Python 模块库，无须安装，解压即用。

本书所有案例程序可用于 zwPython 平台，以及各种支持 Python 3 的设备平台，包括 Linux 操作系统、Mac 苹果电脑，以及安卓系统、树莓派。

其他非 zwPython 用户运行本书程序，如果出现问题，通常是缺少有关的 Python 模块库，可以根据调试信息安装相关的 Python 模块库，再运行相关程序。

限于篇幅，关于 Python 语言和 pandas 数据分析软件的基本操作，请读者查看有关图书。

zwPython 下载地址，请参见 TOP 极宽量化网站“资源中心”：

http://www.topquant.vip，或 http://www.ziwang.com。

1.2.5 程序目录结构

本书配套程序的工作目录是 zwPython\kb_demo，这个目录也是默认的工作目录，凡是没有标注目录的脚本文件，一般都位于该目录。有关的程序会定时在读者群发布更新，请读者及时下载。

kb_demo 目录收录的是本书课件配套代码和所需数据，kb_demo 目录也可以复制到其他目录，建议放到 zwPython 根目录下。

zwPython 目录结构中的其他子目录如下。

- \zwPython\doc\：用户文档中心，包括用户手册和部分中文版的模块库资料。
- \zwPython\py36\：Python 3.6 版本系统目录，除增加、删除模块库外，一般不需要改动本目录下的文件，以免出错。另外，如果日后 Python 版本升级，这个目录也会变化，如 Python 3.7，会采用 py37 目录。
- \zwPython\demo\：示例脚本源码。
- \zwPython\zwrk\：zw 工作目录，用户编写的脚本代码文件建议放在本目录下。
- \zwPython\TopQuant\：极宽量化系统源码。

案例 1-1：重点模块版本测试

案例 1-1 文件名是 kb101_ver.py，主要用于重点模块版本号的测试，因为 Python 发展很快，许多配套的第三方软件模块也更新频繁，有时会出现 API 接口变化，从而造成版本冲突。

这种版本测试，是工程一线特别是团队项目起步阶段规范开发环境常用的一种手段。

案例 1-1 源码很简单，需要注意的是源码头部的 import 模块库导入语句。

```
import tensorflow as tf
import tensorlayer as tl
import keras as ks
import nltk

import pandas as pd
import tushare as ts
import matplotlib as mpl
import plotly
```

```
import arrow

#import tflearn
tflearn = tf.contrib.learn
```

import 导入常用的模块和模块的缩写。前面的 import 模块语句都很正常，请注意最后两行语句。

```
#import tflearn
tflearn = tf.contrib.learn
```

tflearn 模块有两种导入方式。严格说来，第二种方式不是标准的模块导入命令，而是一种别名，是长变量名称的缩写。这是因为，tflearn 本身是 TensorFlow 系统内置的第三方模块，目前也有独立版本的 tflearn 模块，不过两者的源码并非完全同步，笔者在测试 TF-GPU 1.2 和 tflearn 0.31 时，就发现有版本冲突问题。

为了程序的兼容性，通常采用以下的模式。

```
tflearn = tf.contrib.learn
```

案例 1-1 的核心代码也很简单，就是输出相关模块的版本号。

```
#1
print('\n#1 tensorflow.ver:',tf.__version__)

#2
print('\n#2 tensorlayer.ver:',tl.__version__)

#3
print('\n#3 keras.ver:',ks.__version__)

#4
print('\n#4 nltk.ver:',nltk.__version__)

#5
print('\n#5 pandas.ver:',pd.__version__)

#6
print('\n#6 tushare.ver:',ts.__version__)
```

```
#7
print('\n#7 matplotlib.ver:',mpl.__version__)

#8
print('\n#8 plotly.ver:', plotly.__version__)

#9
print('\n#9 arrow.ver:',arrow.__version__)

#10
print('\n#10 tflearn.ver:')
```

以上代码中，需要注意的是使用的版本显示命令。

```
.__version__
```

这个虽然不是标准的 Python 语法，但已经是一种约定俗成，有时，也被称为魔法语句。

对应的输出信息如下。

```
Using TensorFlow backend.

#1 tensorflow.ver: 1.7.0
#2 tensorlayer.ver: 1.8.4
#3 keras.ver: 2.1.6
#4 nltk.ver: 3.2.5
#5 pandas.ver: 0.22.0
#6 tushare.ver: 1.1.7
#7 matplotlib.ver: 2.2.2
#8 plotly.ver: 2.5.1
#9 arrow.ver: 0.12.1
#10 tflearn.ver:
```

以上输出信息中，需要注意的是开头的输出信息。

```
Using TensorFlow backend.
```

以上输出信息，表示使用 TensorFlow 作为后端程序，import 导入 Keras 模块库时自动输出。这条信息只在第一次运行程序时出现，大家可以多运

行几次，看看效果。

此外，需要注意的还有最后一条输出信息。

```
#10 tflearn.ver:
```

tflearn 没有版本号，这是因为 tflearn 模块不支持相关的版本命令。

这可能是因为 tflearn 原本是 TensorFlow 内置的模块，所以没有加入相关的版本语句，即使我们采用 import 导入独立的 tfearn 模块，也无法显示相关的版本号。

目前，我们使用的 tflearn 版本号是 0.31，为了避免版本冲突，还是希望 tflearn 开发团队在未来的更新当中增加相关的版本号信息，

案例 1-1 介绍了机器学习结合金融工程、量化回溯、数据分析等项目时常用的 Python 模块。

- TensorFlow 是神经网络、深度学习开发平台。
- tflearn、Keras、TensorLayer 是 TensorFlow 的简化接口。
- NLTK 是语义分析模块。
- pandas 是新一代数据分析工具。
- Plotly 是新一代互动型数据可视化绘图工具。
- Arrow 是新一代优雅、简洁的时间模块。
- Matplotlib 是经典的绘图模块。
- Tushare 是国内股票数据的采集模块。

以上模块和对应的版本，我们均已集成在最新版本的 zwPython 当中。使用其他平台，如 Mac、Linux，或者其他 Python 开发环境的用户，请自行升级安装以上模块，并注意相关的版本号。

升级安装相关模块库时，请注意相关的依赖模块软件，特别是手动升级独立的安装包时。

笔者在升级以上模块库时，就遇到提示说，新版本的 pandas 缺乏某种属性，无法使用。后来在 Stack Overflow 这个全球最大的 IT 答问网站才找到答案。原来是配套的 Dash 模块也必须升级到最新的 Dash 0.14.3 版本。

1.3　机器学习：从忘却开始

广大初学者面对人工智能、机器学习这些“高大上”的概念，一方面迫切希望能够掌握相关的知识，另外一方面，面对各种层出不穷的概念，往往会眼花缭乱，无所适从。

目前 Python 已经是人工智能、机器学习的行业标准语言，TensorFlow、Torch 在 2015 年、2016 年先后开放了 Python 语言接口。

scikit-learn 是 Python 语言最重要的人工智能模块库，目前已经收入 scikit 套件，通常简称为 sklearn。

还记得本书开头的那张 scikit-learn 算法模块图吗？不知道大家的第一感觉是什么，反正笔者看到此图的第一反应是吓呆了，也明白为什么这么多初学者对于人工智能望而生畏了。

笔者始终认为自己是一名软件工程师，或者叫作程序员，也就是网络上常说的“码农”。

对于程序员而言：Talk is cheap, show me the code！

再多的理论，也比不上几件成功的软件作品，笔者虽然谈不上有很多成功的软件作品，但也写过不少专业的程序，例如原生的 OCR 识别程序和英汉翻译程序。

特别是英语翻译软件，笔者当时（1997 年）的语料库直接使用的是美剧字幕，几千万条的语料库，多种语言素材，全部都有时间戳同步，基本无须任何成本，翻译效果也还不错。而当时同期国家项目组的语料库，耗资巨大，也不过几十万条的数据量。

所谓原生程序，就是直接采用 C 语言、Delphi 语言的标准函数库，没有任何第三方 AI 架构库（如 sklearn、TensorFlow 等 AI 模块库）直接编程的程序。其间的难度，做过系统的程序员，或者阅读过 Linux、安卓系统、TensorFlow 系统源码的程序员就会明白。

这种基于原生的开发，最大的好处就是无论采用何种编程语言，何种理论算法，最终的底层代码结构都是差不多的，就像电脑里面的 CPU，算得再快，也不过是个加法器。

明白了这一点，再看 sklearn 的知识图谱，虽然表面看起来非常烦琐，但仔细梳理，核心还是只有一个：分类。

分好了类，其他的匹配、识别都是简单的问题，只是贴上标签，加上备注而已。

问题最终又回归“一生二、二生三”当中。

国外也有专家认为，所有的人工智能、机器学习，本质上都是二元一次方程的寻优算法。

笔者在博客当中曾经也说过：

简单来说，可以把人工智能、机器学习看成一个巨大的字符串查找算法，只不过这个算法当中的关键词与被查找的字符串的大小非常庞大，趋于无限。

很多初学者面对人工智能、机器学习，往往连系统配置都玩不转，最基本的“Hello 程序”都无法运行。

在 zwPython 用户手册里面，笔者曾经说过，虽然很多人认为 Python 是面向对象的语言，但实践表明，忘记 OOP 对象编程的概念，采用传统的 BASIC 语言（面向过程）模式，学习效率可以提高 10 倍以上。

所以，学习人工智能，笔者的建议就是从忘却开始，忘却各种乱七八糟的概念。

忘记这些概念之后，大家会发现所谓的人工智能，不过是传统的 Python 函数调用，而且只有为数不多的几个函数，最简单的案例当中，只需要 2～3 个函数。

需要说明的是，对于初学者而言，不要左顾右盼，MATLAB、R 语言、Torch、TensorFlow、NLTK 都想学习，反而什么都学不好。

对于初学者而言，不妨在入门阶段就学 sklearn。

sklearn 模块库本身就是人工智能、机器学习的行业标准，该有的人工智能、机器学习经典算法全部都有，其他的模块库无非是在局部进行了某些优化。

至于人工智能的进阶课程，大家不要急，在完成 sklearn 的机器学习课程后，再看 TensorFlow、Torch、MXNet 就不会有看天书的感觉了。

黑箱大法

在机器学习领域许多概念非常抽象拗口，如果大家无法理解，也属于正常情况。

大部分初学者，即使克服种种困难，独自完成了人工智能、机器学习开发平台的配置，面对这些拗口的算法名称，也会有崩溃的感觉。

这很正常，因为这些算法、名称的背后都有非常专业的理论和模型，其学术价值和专业难度都很高。

不过，正如笔者前面所说，初学者对于人工智能、机器学习，最好从忘却开始。

同样，面对这些眼花缭乱的专业名称术语，我们还是采用忘却的模式，采用黑箱大法，大家无须纠结各种算法背后的理论，只将其看作一个个黑箱函数即可。

输入数据 →【黑箱分析】→ 获得结果

函数调用是 Python 语言的基本功能，能够看到这里的读者，想必对函数的调用、编程都非常熟了。

市场经济，讲究的是结果导向，对于大部分学习者而言，需要的也只是最终的结果数据。

采用这种黑箱模式，有了结果数据，再学习理论和算法，就有了具体的数据支持和更多的感性认识，学习过程也会事半功倍。

1.4 学习路线图

要更好地学习本书，掌握相关的配套程序，最好具备以下基础。

- Python 编程基础，不懂 Python 语言的，先花 1 周时间学习一些 Python 的基本知识。
- 花几天时间学习 pandas 数据分析软件的基础操作。
- 要根据代码，学习画流程图，有流程图，就可以把握程序逻辑，特别是策略逻辑。
- 学习过程当中，一定要多提问，特别是在论坛提问，大家都会有收益。
- 若学习中碰到问题难点，就多上网搜索学习。

第 2 章

机器学习编程入门

本章我们通过几个经典的、完整的机器学习小型案例，介绍机器学习应用的完整流程，以及机器学习在金融数据、量化分析过程当中的应用场景。

2.1 经典机器学习算法

目前，机器学习虽然光彩夺目，但还处于启蒙阶段。在本书的案例中，涉及以下经典的机器学习算法。

- 线性回归，函数名：LinearRegression。
- 朴素贝叶斯算法，函数名：Multinomialnb。
- KNN 最近邻算法，函数名：KNeighborsClassifier。
- 逻辑回归算法，函数名：LogisticRegression。
- 随机森林算法，函数名：RandomForestClassifier。
- 决策树算法，函数名：tree.DecisionTreeClassifier。
- 迭代决策树算法，又叫 MART（Multiple Additive Regression Tree），函数名：GradientBoostingClassifier。
- SVM 算法，函数名：SVC。

以上算法中的函数名，均为 sklearn 模块库内置的函数名称，所以，笔者说 sklearn 是初学者学习机器学习的不二选择。

2.2 经典爱丽丝

Iris（爱丽丝）数据集是机器学习最经典的数据集，全称是安德森鸢尾花卉数据集，是统计学习的必备数据集，图 2.1 是爱丽丝数据集的特征分类图。

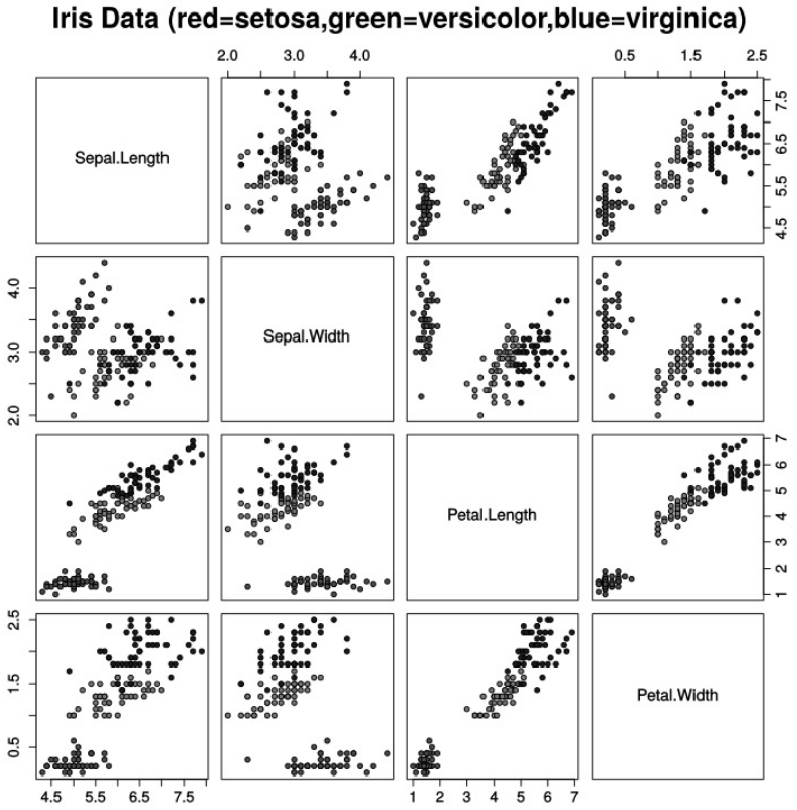


图 2.1 爱丽丝数据集的特征分类图

维基百科有专门的词条。

安德森鸢尾花卉数据集（Anderson's Iris data set），也称鸢尾花卉数据

集（Iris flower data set）或费雪鸢尾花卉数据集（Fisher's Iris data set），是一类多重变量分析的数据集。它最初是埃德加·安德森从加拿大加斯帕半岛上的鸢尾属花朵中提取的地理变异数据，后由罗纳德·费雪作为判别分析的一个例子，并运用到统计学中。其数据集包含了 50 个样本，都属于鸢尾属下的三个亚属，分别是山鸢尾、变色鸢尾和维吉尼亚鸢尾。其 4 个特征被用作样本的定量分析，即花萼和花瓣的长度和宽度。基于这 4 个特征的集合，费雪发展了线性判别分析以确定其属种。

我们的目的就是通过编程，对这 3 种不同种类的爱丽丝植物的数据，采用专业的数据分析手段和人工智能算法，让程序自动判断植物的种类。

由于 sklearn 发布很早，当时还没有 pandas 等新一代数据分析软件，所以 sklearn 为了保证运行速度，直接使用 NumPy 模块库的 ndarray 多维数组作为数据源和内部的数据格式。

近几年发布的 TensorFlow、MXNet 等神经网络、深度学习平台，为了提高效率，在数据接口部分也直接使用 NumPy 模块库的 ndarray 多维数组格式。

不过 NumPy 模块库的 ndarray 多维数组是侧重性能的数据格式，在应用方面非常烦琐。

目前，随着 pandas 等新一代数据分析软件的普及和推广，以及一线开发人员对效率的考虑，越来越多的项目开始使用 pandas 的 DataFrame 数据框架作为数据保存格式，从而方便调试程序，交换数据。

本书的各个案例都是基于 pandas 优先的模式，尽量直接使用 DataFrame 数据框架作为程序中间数据，只是在必须使用 NumPy 模块库的 ndarray 多维数组格式时，才转换为相应的格式。

有关数据转换的细节，我们会在案例当中具体说明。全程采用 pandas 学习 sklearn 人工智能，方便初学者把握数据内部的结构和细节。

传统的 sklearn 人工智能文档，大部分直接采用 NumPy 数组模块，而 NumPy 是为了追求极限性能设计的模块库，很多算法函数非常复杂，不亚

于汇编。

从某种程度上讲，绝大部分初学者的人工智能学习之路，在起步阶段就被 NumPy 这个模块库给吓退了。

本书全部采用现有的 pandas 命令，从数据源对 sklearn 进行整合，无须学习额外的语法，更加方便初学者入门。

案例 2-1：经典爱丽丝

案例 2-1 的文件名是 kb201_iris01.py，程序很简单，秉持着 pandas 优先的原则，只读取爱丽丝数据集当中的数据，并查看其中数据字段的内容。

第 1 组代码如下：

```
#1
fss='data/iris.csv'
df=pd.read_csv(fss,index_col=False)
print('\n#1 df')
print(df.tail())
print(df.describe())
```

第 1 组代码，运行结果如下：

```
     x1   x2   x3   x4       xname
145  6.7  3.0  5.2  2.3  virginica
146  6.3  2.5  5.0  1.9  virginica
147  6.5  3.0  5.2  2.0  virginica
148  6.2  3.4  5.4  2.3  virginica
149  5.9  3.0  5.1  1.8  virginica
             x1          x2          x3          x4
count  150.000000  150.000000  150.000000  150.000000
mean     5.843333    3.054000    3.758667    1.198667
std      0.828066    0.433594    1.764420    0.763161
min      4.300000    2.000000    1.000000    0.100000
25%      5.100000    2.800000    1.600000    0.300000
50%      5.800000    3.000000    4.350000    1.300000
```

```
75%      6.400000    3.300000    5.100000    1.800000
max      7.900000    4.400000    6.900000    2.500000
```

案例 2-1 看起来简单，其实做了很多小优化和修改，比如取消了原始的数据列名称，用 x1～x4 代替。一方面更加通用，另一方面，更加符合人工智能、机器学习的本质。

此外，从输出信息来看，统计命令如下：

```
print(df.describe())
```

在对应的输出信息当中没有 xname 的数据，因为 xname 字段是字符串，无法统计分析，需要先对其进行数字化处理，也就是常说的文字信息的矢量化运算。

第 2 组代码如下：

```
#2
d10=df['xname'].value_counts()
print('\n#2 xname')
print(d10)
```

第 2 组代码当中的函数如下：

```
d10=df['xname'].value_counts()
```

这是调用 pandas 的 value_counts 统计函数，用来查看 xname 数据的具体数据分布情况。

对于植物种类进行了简单的分类统计，共有 3 种，对应的输出信息如下：

```
#2 xname
versicolor    50
setosa        50
virginica     50
Name: xname, dtype: int64
```

由输出信息可以看出，3 种植物名称分别是：山鸢尾（Iris setosa）、变色鸢尾（Iris versicolor）和维吉尼亚鸢尾（Iris virginica）。

案例 2-2：爱丽丝进化与文本矢量化

案例 2-2 的文件名是 kb202_iris02.py，将根据 xname 的植物名称，设置一个新的数据字段 xid，来完成这个文本信息的矢量化工作。

下面我们分组逐一讲解。

第 1 组代码，读取 Iris 数据文件，并保存到 df 变量：

```
#1
fss='data/iris.csv'
df=pd.read_csv(fss,index_col=False)
```

第 2 组代码，根据 xname 字段，按 1、2、3 分别设置 xid 字段，完成读取爱丽丝数据名称的矢量化操作。xid 格式设置为 int 整数格式，并保存到文件 iris2.csv 中。

```
#2
df.loc[df['xname']=='virginica', 'xid'] = 1
df.loc[df['xname']=='setosa', 'xid'] = 2
df.loc[df['xname']=='versicolor', 'xid'] = 3
df['xid']=df['xid'].astype(int)
df.to_csv('tmp/iris2.csv',index=False)
```

我们已经将 iris2.csv 文件复制到 dat 目录下，在后面的案例中，大家可以直接使用这个文件作为数据源：data/iris2.csv。

第 3 组代码，输出修改后的 df 数据信息：

```
#3
print('\n3#df')
print(df.tail())
print(df.describe())
```

对应的输出信息是：

```
      x1   x2   x3   x4      xname  xid
145  6.7  3.0  5.2  2.3  virginica    1
146  6.3  2.5  5.0  1.9  virginica    1
147  6.5  3.0  5.2  2.0  virginica    1
```

```
148  6.2  3.4  5.4  2.3  virginica    1
149  5.9  3.0  5.1  1.8  virginica    1
             x1         x2         x3         x4         xid
count 150.000000 150.000000 150.000000 150.000000 150.000000
mean    5.843333   3.054000   3.758667   1.198667   2.000000
std     0.828066   0.433594   1.764420   0.763161   0.819232
min     4.300000   2.000000   1.000000   0.100000   1.000000
25%     5.100000   2.800000   1.600000   0.300000   1.000000
50%     5.800000   3.000000   4.350000   1.300000   2.000000
75%     6.400000   3.300000   5.100000   1.800000   3.000000
max     7.900000   4.400000   6.900000   2.500000   3.000000
```

第 4 组代码，输出 xname 方面的分类统计信息：

```
d10=df['xname'].value_counts()
print('\n4#xname')
print(d10)
```

对应的输出信息是：

```
4#xname
versicolor    50
setosa        50
virginica     50
Name: xname, dtype: int64
```

第 5 组代码，输出 xid 方面的分类统计信息：

```
d10=df['xid'].value_counts()
print('\n5#xid')
print(d10)
```

对应的输出信息是：

```
5#xid
3    50
2    50
1    50
Name: xid, dtype: int64
```

2.3 机器学习算法流程

通常机器学习的算法流程如下：

- 选择模型函数 mx_fun。
- 把训练用的特征数据集 x_train 和对应的特征（结果）数据集 y_train，输入模型函数 mx_fun。
- 系统内置的机器学习函数会自动分析特征数据与结果数据之间的关系，这样的一个过程就是机器学习的过程，也是算法建模的过程。
- 通过对训练数据的机器学习和数据分析，系统会生成一个 AI 机器学习的模型，我们将其保存到变量 mx。
- 把测试数据 x _test 输入到 mx 模型变量中，mx 会调用内置的分析函数 predict，生成最终的分析结果 y_pred。
- 如果是实盘，我们输入最新的数据，比如今天的股市数据或正在销售的足彩比赛赔率数据，系统会自动生成相关的预测数据，比如每天或未来几天股市走势数据或比赛输赢结果预测数据。

在进行实盘运行前，我们会对模型产生的预测数据 y_pred 和正确结果数据 y_test 进行对比，判断模型的准确度，并通过一些优化措施和参数调整进行迭代运算，或者采用其他的模型提高最终结果的准确度。

2.4 机器学习数据集

机器学习通常使用两组数据，一组作为训练数据，一组作为测试数据。

在每组数据中都包含一组多维的参数数据集作为特征数据集，以及一组一维的数组作为结果分类数据，从而形成 4 个数据集合，通常这 4 组数据变量的名称如下。

- x_train，训练数据，多维参数数据集。
- y_train，训练数据，一维结果数据集。

- x_test，测试数据，多维参数数据集。
- y_test，测试数据，一维参数数据集。

习惯上，train 数据集用于训练，test 数据集用于测试。此外，通过对数据集 y_train 的分析，会生成一个新的预测结果数据集 y_pred，这个数据集也是一维的结果数据集。

通过对结果数据集 y_pred 与实际的测试数据集 y_test 的对比，就可以检测模型的准确度。

案例 2-3：爱丽丝分解

案例 2-3 的文件名是 kb203_iris03.py，具体讲解如何分割相关的数据，下面分组进行介绍。

第 1 组代码，读取 Iris 数据文件，并保存到 df 变量：

```
#1
fss='data/iris2.csv'
df=pd.read_csv(fss,index_col=False)
```

请注意，这里使用的是我们修改后增加了 xid 的爱丽丝数据源文件。

第 2 组代码，输出 df 数据信息：

```
#2
print('\n2#df')
print(df.tail())
```

对应的输出信息是：

```
     x1   x2   x3   x4     xname  xid
145  6.7  3.0  5.2  2.3  virginica    1
146  6.3  2.5  5.0  1.9  virginica    1
147  6.5  3.0  5.2  2.0  virginica    1
148  6.2  3.4  5.4  2.3  virginica    1
149  5.9  3.0  5.1  1.8  virginica    1
```

第 3 组代码，根据机器学习算法要求，设置总的数据源 x、y：

```
    #3
    xlst,ysgn=['x1','x2','x3','x4'],'xid'
    x,y= df[xlst],df[ysgn]
    #
    print('\n3# xlst,',xlst)
    print('ysgn,',ysgn)
    print('x')
    print(x.tail())
    print('y')
    print(y.tail())
```

对应的输出信息是：

```
    3# xlst, ['x1', 'x2', 'x3', 'x4']
    ysgn, xid
    x
         x1   x2   x3   x4
    145  6.7  3.0  5.2  2.3
    146  6.3  2.5  5.0  1.9
    147  6.5  3.0  5.2  2.0
    148  6.2  3.4  5.4  2.3
    149  5.9  3.0  5.1  1.8
    y
    145   1
    146   1
    147   1
    148   1
    149   1
    Name: xid, dtype: int64
```

第 4 组代码，生成 x_train、x_test、y_train、y_test 数据，并输出相关数据的数据格式信息：

```
    #4
    x_train, x_test, y_train, y_test = train_test_split(x, y,
random_state=1)
    x_test.index.name,y_test.index.name='xid','xid'
    print('\n4# type')
```

```
print('type(x_train),',type(x_train))
print('type(x_test),',type(x_test))
print('type(y_train),',type(y_train))
print('type(y_test),',type(y_test))
```

对应的输出信息是：

```
4# type
type(x_train), <class 'pandas.core.frame.DataFrame'>
type(x_test), <class 'pandas.core.frame.DataFrame'>
type(y_train), <class 'pandas.core.series.Series'>
type(y_test), <class 'pandas.core.series.Series'>
```

第 5 组代码，保存相关的数据：

```
#5
fs0='tmp/iris_'
print('\n5# fs0,',fs0)
x_train.to_csv(fs0+'xtrain.csv',index=False);
x_test.to_csv(fs0+'xtest.csv',index=False)
y_train.to_csv(fs0+'ytrain.csv',index=False,header=True)
y_test.to_csv(fs0+'ytest.csv',index=False,header=True)
```

需要注意的是以下代码：

```
y_test.to_csv(fs0+'ytest.csv',index=False,header=True)
```

其中的 header 参数我们很少使用，强制保存 xid 字段头信息，不然 y 数据集会缺少字段头 xid，少一行数据，与 x 数据集尺寸不匹配。大家可以自己测试一下，看看 header 值为 False 的结果。

对应的输出信息：

```
5# fs0, tmp/iris_
```

需要说明的是，以上数据我们均会复制到 dat 目录当中，以用于稍后的案例中。

第 6 组代码，输出 x 数据集：

```
#6
print('\n6# x_train')
print(x_train.tail())
print('\nx_test')
```

```
print(x_test.tail())
```

对应的输出信息如下：

```
6# x_train
     x1   x2   x3   x4
133  6.3  2.8  5.1  1.5
137  6.4  3.1  5.5  1.8
72   6.3  2.5  4.9  1.5
140  6.7  3.1  5.6  2.4
37   4.9  3.1  1.5  0.1

x_test
     x1   x2   x3   x4
xid
128  6.4  2.8  5.6  2.1
114  5.8  2.8  5.1  2.4
48   5.3  3.7  1.5  0.2
53   5.5  2.3  4.0  1.3
28   5.2  3.4  1.4  0.2
```

第 7 组代码，输出 y 数据集：

```
#7
print('\n7# y_train')
print(y_train.tail())
print('\ny_test')
print(y_test.tail())
```

对应的输出信息如下：

```
7# y_train
133    1
137    1
72     3
140    1
37     2
Name: xid, dtype: int64

y_test
```

```
    xid
    128    1
    114    1
    48     2
    53     3
    28     2
    Name: xid, dtype: int64
```

2.5　数据切割函数

在本书中，涉及一个 sklearn 模块库的专业函数：

```
    train_test_split
```

train_test_split 函数只是对数据进行切割，属于数据预处理函数，并非正式的机器学习函数。

我们也可以通过其他函数或者自行编写的函数完成类似的功能。在前面的案例中，数据文件切割也有类似的功能。

对于普通小型数据集合而言，sklear 内置的 train_test_split 函数使用更加方便。

train_test_split 数据分割函数位于 sklearn 的 cross_validation 子模块中，功能是从样本中按比例随机选取 train data 和 test data。

需要注意的是，在未来新版本的 sklearn 模块库中，cross_validation 子模块会被废弃，train_test_split 数据分割函数会采用全新的函数接口。

目前，该函数的调用格式形式为：

```
    x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.4, random_state=0)
```

其中：

- x 是训练参数的数据集合。
- y 是训练参数 x 对应的结果数据集合。

- test_size 是样本占比，如果是整数，就是样本的数量。
- random_state 是随机数的种子。

2.6 线性回归算法

有了合适的训练数据和测试数据，人工智能就变得很简单，只要两三个函数，就可完成相关的编程。

线性回归算法是最简单的机器学习算法之一，如图 2.2 所示。

百度百科对应的“线性回归算法”词条是：

线性回归是利用数理统计中的回归分析，来确定两种或两种以上变量间相互依赖的定量关系的一种统计分析方法，运用十分广泛。其表达形式为 $y = w'x+e$，e 为误差，服从均值为 0 的正态分布。

线性回归算法是最简单、最经典、最古老的人工智能算法，其背后的理论非常复杂，在此，我们采用前面所说的黑箱模式，不予深入讨论，有兴趣的读者请自行参考相关资料。

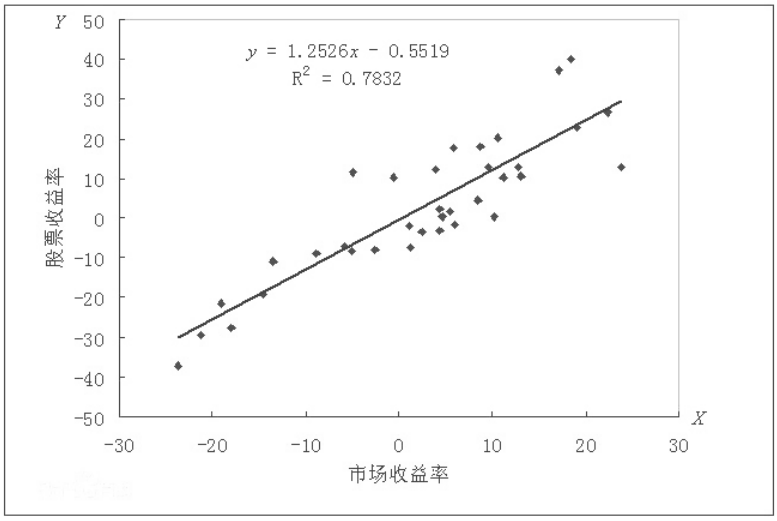


图 2.2 线性回归算法例子

案例 2-4：爱丽丝回归

案例 2-4 的文件名是 kb204_iris04.py，是一个百分之百的机器学习程序，通过对输入数据的学习，可以自动对测试数据进行分类。

下面，我们逐一对程序代码进行讲解。

第 1 组代码，读取训练数据，并保存到相关变量：

```
#1
fs0='data/iris_'
print('\n1# fs0,',fs0)
x_train=pd.read_csv(fs0+'xtrain.csv',index_col=False);
y_train=pd.read_csv(fs0+'ytrain.csv',index_col=False);
```

第 2 组代码，输出部分训练数据尾部：

```
#2
print('\n2# train')
print(x_train.tail())
print(y_train.tail())
```

对应的输出信息如下：

```
2# train
     x1   x2   x3   x4
107  6.3  2.8  5.1  1.5
108  6.4  3.1  5.5  1.8
109  6.3  2.5  4.9  1.5
110  6.7  3.1  5.6  2.4
111  4.9  3.1  1.5  0.1
    xid
107   1
108   1
109   3
110   1
111   2
```

第 3 组代码，调用机器学习函数，通过对输入数据的分析、学习，建立机器学习模型，并保存到变量 mx：

```
    #3
    print('\n3# 建模')
    mx =zai.mx_line(x_train.values,y_train.values)
```

本案例当中第 3 组代码使用的是线性回归算法建立机器学习模型，但是并没有直接调用 sklearn 模块库中的 LinearRegression 线性回归建模函数，而是通过 ztools_ai 极宽智能模块库的 mx_line 函数接口间接进行调用的，对应的函数代码是：

```
    def mx_line(train_x, train_y):
        mx = LinearRegression()
        mx.fit(train_x, train_y)
        #print('\nlinreg.intercept_')
        #print (mx.intercept_);print (mx.coef_)

        return mx
```

线性回归函数，位于 sklearn.linear_model 模块中，函数接口是：

```
    LinearRegression(fit_intercept=True, normalize=False,
copy_X=True, n_jobs=1)
```

mx_line 函数代码很简单，调用基于最小二乘法的 LinearRegression 线性回归函数，生成模型变量 mx，运行内置的 fit 命令，分析学习训练数据：train_x（训练数据）、train_y（训练数据对应的答案）。

sklearn 模块库中的各种机器学习函数，基本上都采用 fit 命令自动学习、建立模型。

采用 ztools_ai 极宽智能模块库的 mx_xxx 系列函数，对 sklearn 模块中的各种智能算法函数进行二次封装，其优点如下。

- 统一调用接口，规范函数 API 调用模式。
- 无须直接面对底层的机器学习函数，如 LinearRegression，无须了解相关的理论知识即可直接使用。

如果不考虑各种复杂的机器学习函数名称，只要记住 mx_xxx 系列函数，那么复杂的机器学习建模过程，就基本简化成一个最简单的 fit 内置函数了。

第 4 组代码，读入测试数据并输出相关信息：

```
#4
x_test=pd.read_csv(fs0+'xtest.csv',index_col=False)
df9=x_test.copy()
print('\n4# x_test')
print(x_test.tail())
```

对应的输出信息是：

```
4# x_test
    x1   x2   x3   x4
33  6.4  2.8  5.6  2.1
34  5.8  2.8  5.1  2.4
35  5.3  3.7  1.5  0.2
36  5.5  2.3  4.0  1.3
37  5.2  3.4  1.4  0.2
```

第 5 组代码，运行机器学习变量 mx 的内置函数 predict，生成结果数据：

```
#5
print('\n5# 预测')
y_pred = mx.predict(x_test.values)
df9['y_predsr']=y_pred
```

第 6 组代码，读入训练数据的正确答案，并保存到变量 y_test：

```
#6
y_test=pd.read_csv(fs0+'ytest.csv',index_col=False)
print('\n6# y_test')
print(y_test.tail())
```

对应的输出信息是：

```
6# y_test
    xid
33    1
34    1
35    2
36    3
37    2
```

需要注意的是，本案例为了强调学习，特意把结果数据放在 predict 函

数之后，在生成结果或者预测数据之后再读入，以强调两者之间是完全独立的。

第 7 组代码，整理结果数据变量 df9，并保存到文件中：

```
#7
df9['y_test'],df9['y_pred']=y_test,y_pred
df9['y_pred']=round(df9['y_predsr']).astype(int)
df9.to_csv('tmp/iris_9.csv',index=False)
print('\n7# df9')
print(df9.tail())
```

对应的输出信息是：

```
7# df9
    x1   x2   x3   x4  y_predsr  y_test  y_pred
33  6.4  2.8  5.6  2.1  1.551677       1       2
34  5.8  2.8  5.1  2.4  1.209887       1       1
35  5.3  3.7  1.5  0.2  2.093058       2       2
36  5.5  2.3  4.0  1.3  2.317451       3       2
37  5.2  3.4  1.4  0.2  2.300976       2       2
```

大家可以自己打开结果文件，查看相关的结果，其中 y_test 是正确结果，y_pred 是程序生成的结果数据。

第 8 组代码，检验测试结果：

```
#8
dacc,df9x=zai.ai_acc_xed2x(df9['y_test'],df9['y_pred'],1,False)
print('\n8.1# mx:mx_sum,kok:{0:.2f}%'.format(dacc))
#
print('\n8.2# df9x')
print(df9x.tail())
```

在本案例中，测试结果数据使用的是 ztools_ai 极宽智能模块库当中的预测数据验证函数：zai.ai_acc_xed2x。

第 8 组代码，对应的输出信息是：

```
8.1# mx:mx_sum,kok:44.74%
```

由结果数据可以看出，线性回归的准确率有些低，只有 44.74%，不过

这个案例是三选一，相比 33%的随机概率还是提高了不少。

随后的输出信息是：

```
8.2#df9x
    y_true  y_pred  y_diff  y_true2     y_kdif
33       1       2       1       1.0  100.000000
34       1       1       0       1.0    0.000000
35       2       2       0       2.0    0.000000
36       3       2       1       3.0   33.333333
37       2       2       0       2.0    0.000000
```

预测结果 df9['y_pred']数据字段和真实数据 df9['y_test']字段是经过对比分析后的内部结果数据。

至此，一个完整的机器学习程序就完成了。虽然有些简单，但毕竟只是一个开始。众所周知，从 0 到 1 是一个艰难而又漫长的过程，也是一个质变的过程。至此，大家迈出了机器学习的第一步。

第 3 章

金融数据的预处理

在现实世界中，金融股票数据是相对比较规范的数据源，但还是存在许多不完整、不一致的“脏数据”。这些“脏数据”无法直接进行数据分析和机器学习，还会影响最终的结果。

为了提高数据分析、机器学习的质量，数据预处理技术应运而生。

数据预处理，英文名称是：Data Preprocessing，是指在数据分析、机器学习前，对所收集的数据进行分类或分组时，所做的审核、筛选、排序等必要的处理过程。

数据预处理有多种方法：数据清理、数据集成、数据变换、数据归约等，其核心都是数据的“归一化”处理。

各种数据预处理技术，大大提高了数据分析模式的质量，降低了金融量化分析所需的时间。

3.1 至简归一法

数据归一化是数据预处理当中最简单也最常见的处理手段。即将有量纲的表达式经过变换化为无量纲的表达式，成为标量。在很多计算中都经常用到这种方法。

数据归一化方法有两种形式。

- 把数变为(0, 1)之间的小数。
- 把有量纲表达式变为无量纲表达式。

数据归一化主要是为了方便数据处理而提出来的，把数据映射在(0, 1)的范围之内来处理，可以更加便捷快速。数据归一化是一种无量纲处理手段，可以使数据源的数值绝对值变成某种相对值的关系，是一种简化计算、缩小量值的有效办法。而且在各种运算都结束后，再经过反归一化处理，一切数据都可以复原。

案例 3-1：麻烦的外汇数据

尽管在 zwDAT 金融数据集当中已经收录了各种外汇交易对的历史交易数据，但都是 2016 年以前的，笔者决定采用最新的外汇数据作为案例数据。

常用的 Tushare 只有国内的 A 股数据，很多大数据网站的 API 只提供 3～5 天的数据，而且需要注册。和讯等财经网站也是股票历史数据，基本上没有外汇历史数据，最终还是在 pandas 上找到了解决方案。

pandas 在其独立出来的 pandas_datareader 模块库中集成了多个著名的金融数据网站，一站式提供了股票期货、外汇黄金等多种金融数据。

pandas_datareader 模块库，内置的金融数据网站包括：Yahoo、Enigma、FRED、Google、MOEX、Morningstar、Quandl、Stooq、NASDAQ、IEX 等。

pandas_datareader 模块库源码在 GitHub 开源项目网站上：https://pydata.github.io/pandas-datareader/stable/。

案例 3-1 的文件名是 kb301_xdown.py，本案例介绍如何使用 pandas_datareader 模块库，并下载最新的外汇交易数据。

程序文件导入模块库代码如下：

```
import pandas_datareader.data as web
```

采用 web 作为 pandas_datareader 模块库的缩写是为了和老版本的 pandas 保持兼容。

pandas_datareader 模块库在作为独立项目以前，是 pandas 系统中子模块 io 的组成部分，原来的导入代码如下：

```
import pandas.io.data as web
```

因此，为了兼容和遵循习惯，我们在这里还是用 web 作为模块库的缩写。也有部分程序用 pdr 作为 pandas_datareader 模块库的缩写，这是选择 pandas、data、reader 三个单词首字母的模式，也是一种常用的编程习惯。

下面，我们分组对程序代码进行说明。

第 1 组代码如下：

```
#1
jpy = web.DataReader('USDJPY','stooq')
jpy.to_csv('tmp/usdjpy2018.csv')
zt.prDF('#1 jpy',jpy)
```

调用 pandas_datareader 模块库的 DataReader 函数，读取 USDJPY（美元/日元交易对）的交易数据。

DataReader 函数是一个一站式数据读取、产生的函数，其中的名词含义如下。

- USDJPY，外汇交易对代码。
- stooq，数据源名称，这里表示 stooq.com 金融数据网站。

DataReader 函数功能很强，提供了起始时间、结束时间等参数。此外，pandas_datareader 模块库还针对各种不同的金融数据网站，提供了独立的下载程序，这些都属于专业的领域，有兴趣研究金融数据抓取的读者可以自行查看相关的源码。

第 1 组代码，对应的输出信息如下：

```
#1 jpy
              Open    High     Low   Close
Date
```

```
2018-04-26  109.44  109.46  109.07  109.31
2018-04-25  108.82  109.45  108.79  109.40
2018-04-24  108.72  109.20  108.54  108.82
2018-04-23  107.77  108.75  107.66  108.72
2018-04-20  107.40  107.86  107.38  107.65
2018-04-19  107.22  107.52  107.18  107.39
2018-04-18  107.02  107.38  107.01  107.23
2018-04-17  107.14  107.21  106.89  107.02
2018-04-16  107.50  107.60  107.04  107.14
2018-04-13  107.30  107.78  107.20  107.38

            Open    High    Low   Close
Date
1971-01-15  358.40  358.40  358.40  358.40
1971-01-14  358.39  358.39  358.39  358.39
1971-01-13  358.46  358.46  358.46  358.46
1971-01-12  358.04  358.04  358.04  358.04
1971-01-11  357.97  357.97  357.97  357.97
1971-01-08  357.83  357.83  357.83  357.83
1971-01-07  357.87  357.87  357.87  357.87
1971-01-06  357.87  357.87  357.87  357.87
1971-01-05  357.81  357.81  357.81  357.81
1971-01-04  357.73  357.73  357.73  357.73
```

从输出信息可以看出，pandas_datareader 模块库在 Stooq 网站可以抓取自 1971 年开始的外汇交易历史数据。时间区间长达近 50 年，这是比较难得的。

第 2 组代码与第 1 组代码类似，读取 USDEUR（美元/欧元交易对）历史数据，有关的输出信息省略，代码如下：

```
#2
eur = web.DataReader('USDEUR','stooq')
eur.to_csv('tmp/USDEUR2018.csv')
zt.prDF('#2 eur',eur)
```

有趣的是 pandas_datareader 模块库在 Stooq 网站还可以抓取 BTC（比

特币）的交易数据。

第 3 组的代码如下：

```
#3
btc = web.DataReader('BTCUSD','stooq')
btc.to_csv('tmp/BTCUSD2018.csv')
zt.prDF('#3 BTC',btc)
```

由于 BTC 目前汇率太高，将近 1 万美元，所以我们采用的是 BTCUSD（比特币/美元交易对）模式，与前面两个交易对相反。

第 3 组代码对应的输出信息如下：

```
#3 BTC
              Open     High      Low    Close
Date
2018-04-26  9165.90  9174.31  8658.01  9090.91
2018-04-25  9469.70  9746.59  8733.62  9165.90
2018-04-24  8936.55  9487.67  8912.66  9469.70
2018-04-23  8912.66  8984.73  8764.24  8936.55
2018-04-20  8271.30  8554.32  8223.68  8517.89
2018-04-19  8190.01  8291.87  8103.73  8271.30
2018-04-18  7911.39  8210.18  7867.82  8190.01
2018-04-17  8000.00  8149.96  7836.99  7911.39
2018-04-16  8305.65  8410.43  7911.39  7993.61
2018-04-13  7830.85  8216.93  7757.95  8097.17

           Open  High   Low  Close
Date
2010-07-30  0.07  0.07  0.06   0.06
2010-07-29  0.06  0.07  0.06   0.07
2010-07-28  0.06  0.06  0.05   0.06
2010-07-27  0.06  0.06  0.05   0.06
2010-07-26  0.05  0.06  0.05   0.06
2010-07-23  0.05  0.07  0.05   0.06
2010-07-22  0.08  0.08  0.05   0.05
2010-07-21  0.07  0.08  0.07   0.08
```

```
2010-07-20  0.08  0.08  0.07   0.07
2010-07-19  0.09  0.09  0.08   0.08
```

从输出信息可以看出，起始时间是 2010 年 7 月 19 日，比外汇数据少很多，这是因为 BTC 属于数字货币，最早的记录就是从 2010 年开始的。

从以上输出信息可以看出，2010 年 7 月比特币的价格才 0.09 美元，而到了 2018 年 4 月已经是 9165 美元，8 年时间上涨了约 10 万倍。

案例 3-2：尴尬的日元

“广场协议”后，日本经济进入了“失去的二十年”，接着又是“失去的二十年”，如今已经开始进入第三次“失去的二十年”了。

不过，在国际金融市场日本一直是个重量级玩家，日元也是重要的国际贸易货币。

图 3.1 是 USDJPY（美元/日元交易对）、USDEUR（美元/欧元交易对）的汇率曲线对比图。

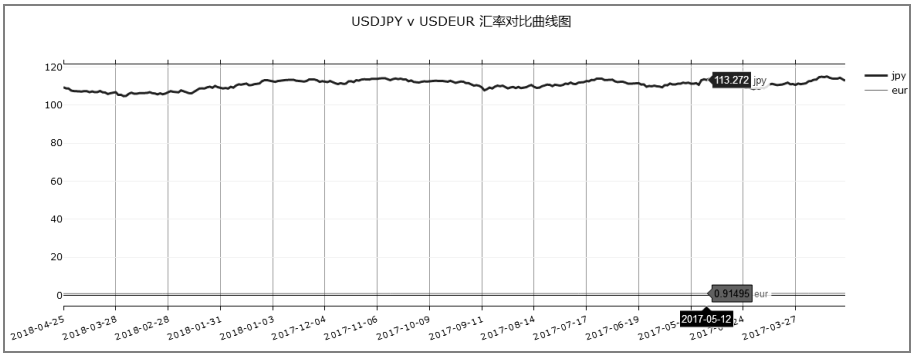


图 3.1　美元/日元交易对、美元/欧元交易对的汇率曲线对比图

尽管我们采用的是最先进的互动型 Plotly 绘图模块，可以交互式提供对比数据，但是图 3.1 的美元/欧元汇率曲线像一根直线，看不到任何波动线条。这是因为日元当中没有分、角的概念，只有元一种单位，相当于把元的币值缩小了，一日元相当于其他币种的一分。而欧元、美元本身就是

高汇率货币，两者对比，汇率数值差距有上百倍，直接绘图就会出现图 3.1 的尴尬情况。

案例 3-2 的文件名是 kb301_jp.py，本案例介绍如何提供最简单的编程手段，并通过类似归一化的数据处理方法来解决问题。

第 1 组代码如下：

```
#1
fjpy='data/USDJPY2018.csv'
feur='data/USDEUR2018.csv'
print('\n1# f,',fjpy,feur)
jpy=pd.read_csv(fjpy,index_col=0)
eur=pd.read_csv(feur,index_col=0)
#
zt.prDF('#1.1 @jpy',jpy)
#
zt.prDF('#1.2 @eur',eur)
```

读取 USDJPY（美元/日元交易对）、USDEUR（美元/欧元交易对）的数据文件，文件保存在 data 目录中。

对应的输出信息如下：

```
1# f, data/USDJPY2018.csv data/USDEUR2018.csv
```

随后是对应的 USDJPY 美元/日元交易对、USDEUR 美元/欧元交易对汇率数据：

```
    #1.1 @jpy
                Open    High
Low   Close
    Date
    2018-04-25  108.82  109.45
108.79  109.40
    2018-04-24  108.72  109.20
108.54  108.82
    2018-04-23  107.77  108.75
107.66  108.72
    2018-04-20  107.40  107.86
```

```
    #1.2 @eur
                Open  High   Low
Close
    Date
    2018-04-25  0.82  0.82  0.82
0.82
    2018-04-24  0.82  0.82  0.82
0.82
    2018-04-23  0.81  0.82  0.81
0.82
    2018-04-20  0.81  0.82  0.81
```

```
107.38  107.65
    2018-04-19  107.22  107.52
107.18  107.39
    2018-04-18  107.02  107.38
107.01  107.23
    2018-04-17  107.14  107.21
106.89  107.02
    2018-04-16  107.50  107.60
107.04  107.14
    2018-04-13  107.30  107.78
107.20  107.38
    2018-04-12  106.80  107.43
106.70  107.31

                Open    High
Low   Close
    Date
    1971-01-15  358.40  358.40
358.40  358.40
    1971-01-14  358.39  358.39
358.39  358.39
    1971-01-13  358.46  358.46
358.46  358.46
    1971-01-12  358.04  358.04
358.04  358.04
    1971-01-11  357.97  357.97
357.97  357.97
    1971-01-08  357.83  357.83
357.83  357.83
    1971-01-07  357.87  357.87
357.87  357.87
    1971-01-06  357.87  357.87
357.87  357.87
    1971-01-05  357.81  357.81
357.81  357.81
    1971-01-04  357.73  357.73
```

```
0.81
    2018-04-19  0.81  0.81  0.81
0.81
    2018-04-18  0.81  0.81  0.81
0.81
    2018-04-17  0.81  0.81  0.81
0.81
    2018-04-16  0.81  0.81  0.81
0.81
    2018-04-13  0.81  0.81  0.81
0.81
    2018-04-12  0.81  0.81  0.81
0.81

                Open  High   Low
Close
    Date
    1971-01-15  1.87  1.87  1.87
1.87
    1971-01-14  1.87  1.87  1.87
1.87
    1971-01-13  1.87  1.87  1.87
1.87
    1971-01-12  1.87  1.87  1.87
1.87
    1971-01-11  1.87  1.87  1.87
1.87
    1971-01-08  1.87  1.87  1.87
1.87
    1971-01-07  1.87  1.87  1.87
1.87
    1971-01-06  1.87  1.87  1.87
1.87
    1971-01-05  1.87  1.87  1.87
1.87
    1971-01-04  1.87  1.87  1.87
```

```
357.73  357.73

    len-DF: 12095
```

```
1.87

    len-DF: 12093
```

第 2 组代码如下：

```
    #2
    #2.1
    print('\n2# train')
    df2=pd.DataFrame()

    df2['jpy']=jpy['Close']
    df2['eur']=eur['Close']
    zt.prDF('#2.1 @df2',df2)
    #
    #2.2
    df2['jpy']=df2['jpy']/100
    zt.prDF('#2.2 @df2.x',df2)
```

把 USDJPY（美元/日元交易对）、USDEUR（美元/欧元交易对）两组不同的汇率数据，合并到一个数据变量 df2 中以便于比较。

注意 2.2 段的代码：

```
    df2['jpy']=df2['jpy']/100
```

把日元的价格缩小以便于两者比较，这里的 100 是经验参数，不是正规的数据归一化模式。

第 3 组代码如下：

```
    #3
    df3=df2.head(300)
    df3=df3.sort_index()
    zdr.drm_line(df3,'USDJPY v USDEUR 汇率对比曲线图
',xlst=['jpy','eur'])
```

绘制汇率曲线图，这段码使用了几个小技巧。

技巧1，数据截取：

```
    df3=df2.head(300)
```

截取最新 300 组交易数据用于绘图，不然近 50 年的交易数据有上万条，很容易出现混乱。

因为 df2 采用的是逆向排序，最新的数据在前面，所以我们使用的是 head 函数；如果最新的数据在记录后面，一般使用 tail 函数。

技巧2，排序整理：

```
df3=df3.sort_index()
```

因为 df2 采用的是逆向排序，最新的数据在前面，如果不进行处理，绘制汇率曲线图时最新的数据就会位于图形最左边，和正常操作相反。

数据级 df3 截取自 df2，所以也需要进行排序处理。

图 3.2 是日元/欧元价格对比曲线图。

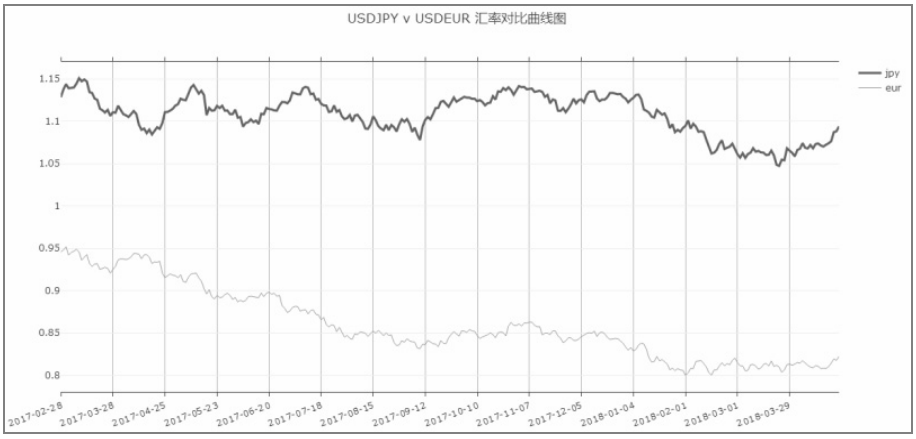


图 3.2　日元欧元价格对比曲线图

从图 3.2 中我们可以看出，经过修正后价格曲线图效果好了很多。

案例 3-3：凶残的比特币

案例 3-3 的文件名是 kb303_btc.py，本案例介绍如何通过数据归一化绘制价格相差巨大的币种的价格曲线图。

尽管比特币的汇率高达近万美元，可还是有很多其他数字货币的价格很低，特别是用于微支付领域的数字货币，比如 DOG（狗狗币）就是其中的一种，其汇率只有 0.2 美分。

案例 3-3 与案例 3-2 类似，我们只对重点部分进行讲解，全部源码如下：

```
#1
fbtc='data/btc2018.csv'
fdog='data/doge2018.csv'
print('\n1# f,',fbtc,fdog)
btc=pd.read_csv(fbtc,index_col=0)
dog=pd.read_csv(fdog,index_col=0)
zt.prDF('#1.1 @btc',btc)
#
zt.prDF('#1.2 @dog',dog,nfloat=3)
#--
#2
print('\n2# train')
df2=pd.DataFrame()
df2['btc']=btc['close']
df2['dog']=dog['close']
df2['btc1k']=df2['btc']/1000
df2['dog1k']=df2['dog']*1000
zt.prDF('#2 @df2.x',df2,nfloat=3)

#3
df3=df2.head(300)
df3=df3.sort_index()
zdr.drm_line(df3,'BTC v DOG 价格曲线图',xlst=['btc1k','dog1k'])
```

代码的重点是第 2 组程序当中的以下语句：

```
df2['btc1k']=df2['btc']/1000
df2['dog1k']=df2['dog']*1000
```

我们定义了两个新的变量：

- btc1k，相当于比特币的千分之一。

- dog1k，相当于 DOG（狗狗币）的一千倍。

这两个变量单位也是笔者在分析数据货币实盘项目中导入的新的中间变量单位。

BTC（比特币）与 DOG（狗狗币）的价格相差十万倍，比欧元、日元的差距还大，所以我们采用了这种双向修正的方法，对原始数据进行修正后再进行对比分析。

图 3.3 是程序输出的 BTC-DOG 价格对比曲线图。从图 3.3 可以看出，我们基本达到了预计的效果。

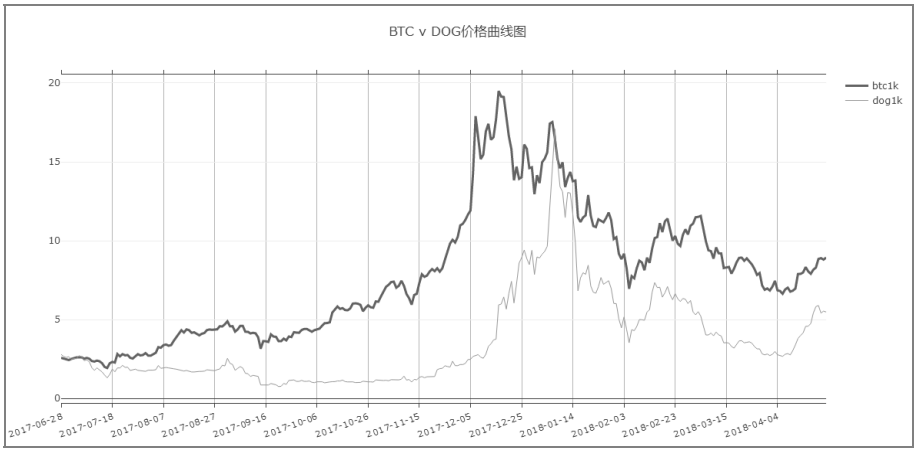


图 3.3　BTC-DOG 价格对比曲线图

3.2　股票池与Rebase

3.2.1　股票池

股票池，英文名称是：Stock Pool，是业内对投资对象的通俗称谓。

股票池体现的是一种股票投资组合，也就是通常说的“不要将鸡蛋放在一个篮子里”。将一笔钱分散投资到几只股票中有利于降低投资风险。

通常股票池内几只股票的选择是有讲究的，比如不同行业的搭配、高

收益（同时意味着高风险）与低风险（同时意味着低收益）的搭配、长短期的搭配等。

笔者常说，传统金融投资与量化投资的一个最大不同点就是股票池的应用。

传统金融投资往往喜欢挑选绩优股、黑马股、潜力股，偏重于对个股的分析，这类似于足球比赛，人们喜欢关注巴西、德国等传统强队。

量化投资不是关注几只强股的表现，而是关注“池子”的整体收益。一个股票池通常有 50～100 只不同风格、类型的股票，哪怕其中有 30%～40%的股票处于亏损状态，只要整个“池子”（pools）的收益是正收益的，就比传统投资更加稳健。

此外，量化投资中股票池的定义与传统金融投资也略有不同。传统投资领域中股票池更多的是备选库的意思。

而量化投资中所谓的股票池是已经进行投资的对象。对于一位量化投资者而言，假设他投资了 50 只股票，他的股票池就是这 50 只股票，而整个沪深两市的几千只股票，都是他的备选投资对象。

量化投资与传统金融投资对于股票池的应用也有很大区别。

传统投资者一般投资的股票品种很少，而且更换的频率很低。而在量化交易当中，股票池一般在 50～100 只股票之间，单只股票的资金额在 50 万元左右，和市场最低交易成本高度相关。此外，股票池的更换非常频繁，而且是周期性的，通常 1～2 周更换其中的部分股票，按 5%～10%的数目或股票收益率、亏损率的百分比进行更换。

在量化交易中更换股票池的通用原则是把近期收益最高的一批股票剔除，这个原则虽然不太符合传统金融投资者的思维，不过却有严格的统计学的数据支持。

3.2.2 Rebase 与归一化

Rebase 的中文直译是换底、改变基数，采用新的基数，对数据进行归

一化处理。

无论是传统金融投资，还是量化交易，在做投资回报分析时，经常需要把不同的股票按投资回报率、收益率归纳在一张图表当中再进行对比分析，从中找出最佳的或最差的投资产品。

一般而言，进行单一股票的收益率分析还算比较简单，把最终的资产总额除以投资额即可。不过，对于量化交易而言，股票池当中有数十只不同的股票，股价都不相同，进行对比分析会非常麻烦，需要预先进行烦琐的数据归一化处理。

不过有了 Rebase 变基归一化函数，这种分析就非常简单了。

需要注意的是，Rebase 变基归一化函数并不是 Python 和 pandas 的原生函数，而是 ffn 金融模块库对于 pandas 的扩充函数。

具体细节，请参看相关网址：http://pmorissette.github.io/ffn/。

案例 3-4：股票池 Rebase 归一化

案例 3-4 的文件名是 kb304_rebase.py，本案例介绍如何读取股票池数据，并调用 Rebase 变基归一化函数进行处理。

程序核心源码如下，我们会逐一进行解释。

第 1 组代码如下：

```
    #1
    print('#1,rd data')
    rss='data/'
    stklst=['000020','000528','000978', '300020', '300035',
'600201', '600211']
    stkPools=zdat.pools_frd4lst(rss,stklst)
```

调用极宽 zdat 模块库的 pools_frd4lst 函数，读取各只股票的 OHLC 交易数据，并保存到股票池变量 stkPools 中。函数输入参数是股票池当中各只股票的代码，采用列表格式。

需要注意的是，股票池变量 stkPools 的数据格式是字典格式。

第 2 组代码如下：

```
#2
print('#2,edit data')
df9=zdat.pools_lnk4lst(stkPools,stklst,'close')
zt.prDF('df9',df9)
```

调用极宽 zdat 模块库的 pools_lnk4lst 函数，把股票池当中多只股票的 close 收盘价全部汇合在单个 DataFrame 变量 df9 当中，便于后期分析。

本组代码对应的输出信息如下：

```
df9
            000020  000528  000978  300020  300035  600201  600211
date
2016-04-08   23.33   6.88   11.53   17.46   12.58   30.41   43.50
2016-04-07   24.00   7.01   11.63   17.91   12.71   30.89   44.88
2016-04-06   25.01   7.06   11.95   18.02   12.70   32.00   45.50
2016-04-05   23.99   7.11   11.85   18.69   13.02   31.37   44.88
2016-04-01   23.60   6.77   11.55   17.95   12.85   30.55   43.54
2016-03-31   24.16   6.83   11.70   18.25   13.00   30.94   44.49
2016-03-30   24.60   6.74   11.70   18.68   13.21   31.11   44.01
2016-03-29   23.38   6.52   11.24   17.11   12.01   30.30   42.98
2016-03-28   23.85   6.76   11.69   17.30   12.40   31.28   44.55
2016-03-25   24.45   6.82   11.46   17.60   12.58   31.34   45.41

            000020  000528  000978  300020  300035  600201  600211
date
1994-01-14    3.28    1.57     NaN     NaN     NaN     NaN     NaN
1994-01-13    3.37    1.62     NaN     NaN     NaN     NaN     NaN
1994-01-12    3.42    1.62     NaN     NaN     NaN     NaN     NaN
1994-01-11    3.46    1.62     NaN     NaN     NaN     NaN     NaN
1994-01-10    3.58    1.66     NaN     NaN     NaN     NaN     NaN
1994-01-07    3.51    1.66     NaN     NaN     NaN     NaN     NaN
1994-01-06    3.53    1.67     NaN     NaN     NaN     NaN     NaN
1994-01-05    3.30    1.58     NaN     NaN     NaN     NaN     NaN
```

```
1994-01-04    3.39    1.59     NaN     NaN     NaN     NaN     NaN
1994-01-03    3.42    1.62     NaN     NaN     NaN     NaN     NaN
```

从以上输出信息可以看出，后半部的输出信息有大量的 NaN 字段，没有交易数据。这是因为案例中股票池里面的很多股票是 1994 年以后才上市的。

第 3 组代码如下：

```
#3
print('#3,cut data time')
tim0str,tim9str='2010-01-01','2015-12-31'
df2=zdat.df_kcut8tim(df9,'',tim0str,tim9str)
df2=df2.sort_index()
zt.prDF('df2',df2)
df2.plot()
```

我们按时间截取 2010—2015 年的交易数据进行对比分析，还对股票数据按时间先后顺序进行重新排序。

本组代码对应的输出信息如下：

```
df2
            000020  000528  000978  300020  300035  600201  600211
date
2010-01-04   11.12   12.68  8.69    5.88   12.06    4.74   13.25
2010-01-05   11.36   12.68  8.93    5.89   11.82    4.73   13.84
2010-01-06   11.00   12.96  8.82    5.65   11.80    4.78   13.42
2010-01-07   10.62   12.58  8.49    5.59   11.79    4.61   13.05
2010-01-08   10.36   12.64  8.65    5.41   11.27    4.62   13.09
2010-01-11   10.39   12.35  8.96    5.49   11.10    4.67   13.88
2010-01-12   11.43   12.90  9.20    5.68   11.29    4.87   13.81
2010-01-13   11.16   12.61  9.20    5.67   11.19    4.84   13.63
2010-01-14   11.24   13.00  9.35    6.07   11.54    5.04   14.05
2010-01-15   11.24   13.22  9.60    6.05   11.58    5.01   14.04

            000020  000528  000978  300020  300035  600201  600211
date
2015-12-18   28.99  8.24   13.66   18.66   16.49   33.75   41.20
```

```
2015-12-21   29.19  8.45   14.50   20.53   16.57   35.56   44.12
2015-12-22   32.11  8.47   15.28   22.58   16.60   34.98   43.19
2015-12-23   34.41  8.44   15.04   24.84   15.76   36.67   42.38
2015-12-24   33.35  8.38   14.85   24.65   16.12   37.43   43.90
2015-12-25   31.95  8.57   14.67   24.09   16.25   37.71   44.00
2015-12-28   31.78  8.26   13.63   24.10   15.67   36.32   42.20
2015-12-29   32.05  8.34   14.40   24.54   17.03   36.52   42.96
2015-12-30   31.08  8.44   14.44   23.99   16.68   35.41   43.40
2015-12-31   29.92  8.29   14.17   24.52   16.10   36.97   43.78

len-DF: 1240
```

由以上输出信息可以看出，数据经过整理后，已经没有数据缺失，而且也是按正常的时间先后顺序进行排列的。

本案例第 1 组到第 3 组程序都是标准的数据预处理步骤，在后面的案例当中，我们会经常重复调用这些步骤，请大家注意。

第 4 组代码如下：

```
#4
print('#4,rebase')
df3=df2.rebase()
zt.prDF('df3',df3)
df3.plot()
```

调用 ffn 金融模块库的 Rebase 变基归一化函数，对数据进行归一化处理。

本组代码对应的输出信息如下：

```
#4,rebase

 df3
            000020  000528  000978  300020  300035  600201  600211
date
2010-01-04 100.00 100.00  100.00 100.00 100.00 100.00 100.00
2010-01-05 102.16 100.00  102.76 100.17  98.01   99.79  104.45
2010-01-06 98.92  102.21  101.50  96.09   97.84  100.84  101.28
2010-01-07 95.50  99.21   97.70   95.07   97.76   97.26   98.49
```

```
2010-01-08 93.17  99.68   99.54   92.01   93.45   97.47   98.79
2010-01-11 93.44  97.40  103.11   93.37   92.04   98.52  104.75
2010-01-12 102.79 101.74  105.87  96.60  93.62  102.74  104.23
2010-01-13 100.36 99.45  105.87  96.43  92.79  102.11  102.87
2010-01-14 101.08 102.52  107.59 103.23  95.69  106.33  106.04
2010-01-15 101.08 104.26  110.47 102.89   96.02  105.70  105.96

          000020  000528  000978  300020  300035  600201  600211
date
2015-12-18 260.70  64.98 157.19  317.35  136.73  712.03  310.94
2015-12-21 262.50  66.64 166.86  349.15  137.40  750.21  332.98
2015-12-22 288.76  66.80 175.83  384.01  137.65  737.97  325.96
2015-12-23 309.44  66.56 173.07  422.45  130.68  773.63  319.85
2015-12-24 299.91  66.09 170.89  419.22  133.67  789.66  331.32
2015-12-25 287.32  67.59 168.81  409.69  134.74  795.57  332.08
2015-12-28 285.79  65.14 156.85  409.86  129.93  766.24  318.49
2015-12-29 288.22  65.77 165.71  417.35  141.21  770.46  324.23
2015-12-30 279.50  66.56 166.17  407.99  138.31  747.05  327.55
2015-12-31 269.06  65.38 163.06  417.01  133.50  779.96  330.42
```

从以上输出信息可以看出，经过 Rebase 变基归一化函数处理后，所有股票的起始数据都变成 100，不仅可以进行对比分析，而且可以看到各只股票的单一收益率。

最后一行输出信息如下：

```
2015-12-31 269.06 65.38 163.06 417.01 133.50 779.96 330.42
```

我们可以看到，在 2010—2015 年期间，案例中股票组合“600201”代码的股票收益率最高，高达 779.96%。

3.3　金融数据切割

前面我们说过，机器学习通常使用两组数据，一组作为训练数据，另一组作为测试数据。而在每组数据当中，都包含一组多维的参数数据集作为特征数据集，以及一组一维的数组作为结果分类数据，从而形成 4 个数

据集合，通常这 4 组数据变量的名称如下。

- x_train，训练数据多维参数数据集。
- y_train，训练数据一维结果数据集。
- x_test，测试数据多维参数数据集。
- y_test，测试数据一维参数数据集。

习惯上将 x 数据集用于训练，y 数据集用于测试。此外通过对数据集 y_train 的分析，会生成一个新的预测结果数据集 y_pred，这个数据集也是一维的结果数据集。

案例 3-5：当上证遇到机器学习

案例 3-5 的文件名是 kb305_tds01.py，本案例将具体介绍如何按机器学习的要求把上证指数分为训练数据集和测试数据集。

第 1 组代码如下：

```
#1
print('#1,rd data')
fss='data/sh2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
zt.prDF('xdf',xdf)
```

读取上证指数数据，数据截至 2018 年 5 月月底，并按时间先后顺序进行排序。

对应的输出信息如下：

```
    open    high   close     low     volume      amount
date
1994-01-03 837.70  840.65  833.90  831.66  101005600 1048326000
1994-01-04 835.97  836.97  832.69  829.89   65274300 692748000
1994-01-05 829.30  847.05  846.98  823.10   89412100 975053000
1994-01-06 850.78  869.33  869.33  850.78  184511700 1970032000
```

```
1994-01-07 875.18  883.99  879.64  873.01  168688400 1752262000
1994-01-10 891.99  900.30  900.30  889.73  187595100 1896704000
1994-01-11 903.54  907.09  891.79  884.00  136850000 1590901000
1994-01-12 891.83  900.23  888.04  885.88  127429900 1306063000
1994-01-13 889.02  899.14  897.46  888.80   80792100 838602000
1994-01-14 900.13  900.99  849.23  842.62  242925400 2505558000

            open    high   close    low      volume     amount
date
2018-05-11 3179.80 3180.76 3163.26 3162.21 13065974900 167364290366
2018-05-14 3167.04 3183.82 3174.03 3163.48 12932735300 172410691054
2018-05-15 3180.42 3192.81 3192.12 3164.52 12454905100 162990790010
2018-05-16 3180.23 3191.95 3169.56 3166.81 13052496800 174590979834
2018-05-17 3170.01 3172.77 3154.28 3148.62 11399556700 150598842185
2018-05-18 3151.08 3193.45 3193.30 3144.78 13651691800 168038057477
2018-05-21 3206.18 3219.74 3213.84 3203.34 16445941300 202663464515
2018-05-22 3211.25 3214.59 3214.35 3192.23 14429268400 185721667752
2018-05-23 3205.44 3205.44 3168.96 3168.96 15780764800 199358101015
2018-05-24 3167.94 3173.53 3154.65 3152.07 12408580000 160658185502
```

第 2 组代码如下：

```
#2
print('#2,xed data')
xdf['y']=xdf['close'].shift(-1)
zt.prDF('xdf#1',xdf)
xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf#2',xdf)
```

机器学习与神经网络在金融量化的主要应用就是预测未来的股价，我们以第 2 天的股价作为预测目标数据。

按照惯例，预测数据字段名采用变量 y 命名，调用 pandas 的 shift 函数，即可产生对应的数据。

第 2.1 组代码对应的输出信息如下：

```
 xdf#2.1
            open    high   close    low      volume     amount     y
```

```
    date
    1994-01-03  837.70  840.65  833.90  831.66  101005600
1048326000  832.69
    1994-01-04  835.97  836.97  832.69  829.89   65274300
692748000  846.98
    1994-01-05  829.30  847.05  846.98  823.10   89412100
975053000  869.33
    1994-01-06  850.78  869.33  869.33  850.78  184511700
1970032000  879.64
    1994-01-07  875.18  883.99  879.64  873.01  168688400
1752262000  900.30
    1994-01-10  891.99  900.30  900.30  889.73  187595100
1896704000  891.79
    1994-01-11  903.54  907.09  891.79  884.00  136850000
1590901000  888.04
    1994-01-12  891.83  900.23  888.04  885.88  127429900
1306063000  897.46
    1994-01-13  889.02  899.14  897.46  888.80   80792100
838602000  849.23
    1994-01-14  900.13  900.99  849.23  842.62  242925400
2505558000  859.28

            open    high    close    low    volume    amount    y
    date
    2018-05-11  3179.80  3180.76  3163.26  3162.21  13065974900
167364290366  3174.03
    2018-05-14  3167.04  3183.82  3174.03  3163.48  12932735300
172410691054  3192.12
    2018-05-15  3180.42  3192.81  3192.12  3164.52  12454905100
162990790010  3169.56
    2018-05-16  3180.23  3191.95  3169.56  3166.81  13052496800
174590979834  3154.28
    2018-05-17  3170.01  3172.77  3154.28  3148.62  11399556700
150598842185  3193.30
    2018-05-18  3151.08  3193.45  3193.30  3144.78  13651691800
168038057477  3213.84
```

```
    2018-05-21  3206.18  3219.74  3213.84  3203.34  16445941300
202663464515  3214.35
    2018-05-22  3211.25  3214.59  3214.35  3192.23  14429268400
185721667752  3168.96
    2018-05-23  3205.44  3205.44  3168.96  3168.96  15780764800
199358101015  3154.65
    2018-05-24  3167.94  3173.53  3154.65  3152.07  12408580000
160658185502     NaN
```

注意以上输出信息的最后一行：

```
    2018-05-24  3167.94  3173.53  3154.65  3152.07  12408580000
160658185502     NaN
```

字段 y 是 NaN 空值，这是因为字段 y 保存的是第二天的预测数据，最后一行，只有当天的 OHLC 数据，没有第二天的数据。

在数据分析过程中，经常会把缺失字段的数据行删除，不过此处的数据属于有用数据，为防止数据清洗过程的错误删除，需要进行填充处理。

pandas 当中对于缺失数据通常采用数值 0，或者用上列数据进行填充。在本案例程序中也是采用上列数据进行填充。

```
    xdf.fillna(method='pad',inplace=True)
```

填充后，对应的输出数据如下：

```
     xdf#2.2
              open    high   close    low    volume    amount    y
    date
    1994-01-03  837.70  840.65  833.90  831.66  101005600
1048326000  832.69
    1994-01-04  835.97  836.97  832.69  829.89   65274300
692748000  846.98
    1994-01-05  829.30  847.05  846.98  823.10   89412100
975053000  869.33
    1994-01-06  850.78  869.33  869.33  850.78  184511700
1970032000  879.64
    1994-01-07  875.18  883.99  879.64  873.01  168688400
1752262000  900.30
    1994-01-10  891.99  900.30  900.30  889.73  187595100
```

```
1896704000  891.79
    1994-01-11  903.54  907.09  891.79  884.00  136850000
1590901000  888.04
    1994-01-12  891.83  900.23  888.04  885.88  127429900
1306063000  897.46
    1994-01-13  889.02  899.14  897.46  888.80   80792100
838602000  849.23
    1994-01-14  900.13  900.99  849.23  842.62  242925400
2505558000  859.28

              open    high    close     low     volume     amount  y
    date
    2018-05-11  3179.80  3180.76  3163.26  3162.21  13065974900
167364290366  3174.03
    2018-05-14  3167.04  3183.82  3174.03  3163.48  12932735300
172410691054  3192.12
    2018-05-15  3180.42  3192.81  3192.12  3164.52  12454905100
162990790010  3169.56
    2018-05-16  3180.23  3191.95  3169.56  3166.81  13052496800
174590979834  3154.28
    2018-05-17  3170.01  3172.77  3154.28  3148.62  11399556700
150598842185  3193.30
    2018-05-18  3151.08  3193.45  3193.30  3144.78  13651691800
168038057477  3213.84
    2018-05-21  3206.18  3219.74  3213.84  3203.34  16445941300
202663464515  3214.35
    2018-05-22  3211.25  3214.59  3214.35  3192.23  14429268400
185721667752  3168.96
    2018-05-23  3205.44  3205.44  3168.96  3168.96  15780764800
199358101015  3154.65
    2018-05-24  3167.94  3173.53  3154.65  3152.07  12408580000
160658185502  3154.65
```

第 3 组代码包括两个部分，把数据分为训练数据集和测试数据集，第 3.1 组代码如下：

```
    #3.1
```

```
print('#3,cut data')
tim0str,tim9str='2010-10-10','2017-12-31'
df_train=zdat.df_kcut8tim(xdf,'',tim0str,tim9str)
zt.prDF('#3.1 df_train\n',df_train)
```

以上采用 2010 年至 2017 年的数据作为训练数据集。

第 3.2 组代码如下：

```
#3.2
df_test=xdf[xdf.index>'2018'] #.tail(100)
zt.prDF('#3.2 df_test\n',df_test)
```

以上代码的输出信息略。

需要注意的是 sklearn 使用的数据格式是 NumPy 的 ndarray 数组格式。训练数据集和测试数据集是 DataFrame 格式，不能直接使用，需要使用变量中的 values 属性调用。

调用前，还需要定义一个 ohlc 列表变量：

```
clst=['open','high','low','close']
```

具体调用时，方法如下。

- x_train=df_train[clst].values，训练数据多维参数数据集。
- y_train=df_train[‘y’].values，训练数据一维结果数据集。
- x_test=df_test[clst].values，测试数据多维参数数据集。
- y_test= df_test[‘y’].values，测试数据一维参数数据集。

具体的使用步骤，我们在稍后的案例中再具体进行讲解。

3.4　preprocessing模块

sklearn 在数据预处理方面做了大量工作，专门集成了一个用于数据预处理的子模块：preprocessing。

事实上，目前的深度学习、神经网络平台，例如 TensorFlow、MXNet 也在直接使用 sklearn 的 preprocessing（数据预处理）模块作为数据清理的工具。

preprocessing 数据预处理模块，全称是预处理和规范化（Preprocessing and Normalization）

sklearn.preprocessing 模块包括数据缩放、归一化、二值化和插补方面的函数，具体如下。

- Binarizer：根据阈值对数据进行二值化。
- FunctionTransformer：构造变换函数。
- Imputer：用于完成缺失值的插补变换函数。
- KernelCenterer：中心内核矩阵。
- LabelBinarizer：标签二值化。
- LabelEncoder：把字符串类型的数据转化为整型编码。
- MultiLabelBinarizer：多标签二值化。
- MaxAbsScaler：按特征最大绝对值进行缩放。
- MinMaxScaler：最大值、最小值规范化。
- Normalizer：样品数据归一化处理，使数据特征值和为 1。
- OneHotEncoder：独热码编码。
- PolynomialFeatures：生成多项式特征参数。
- RobustScaler：使用对异常值可靠的统计信息来缩放。
- StandardScaler：通过删除平均值和缩放到单位方差来标准化特征，使各特征的均值为 0，方差为 1。
- add_dummy_feature：增强数据集。
- binarize：矩阵式布尔阈值。
- label_binarize：以 one-vs-all 的方式对标签进行二值化。
- maxabs_scale：将每个特征缩放到[-1,1]范围，而不破坏稀疏度。
- minmax_scale：通过将每个功能缩放到给定范围来转换功能。
- normalize：将输入向量分别缩放到单位范数（向量长度）。

- robust_scale：沿着任何轴标准化数据集。

我们会通过案例对有关函数进行讲解，更多的函数和细节请大家参考 sklearn 用户手册，逐一进行研究。

案例 3-6：比特币与标准化

数据标准化，英文名称是 DataStandardization，即将特征数据的分布调整成标准正态分布，也叫高斯分布。

数据标准化最常用的方法是 z-score 规范化，也称零均值规范化，也就是使得数据的均值为 0，方差为 1。

标准化的原因在于有些数据特征的方差过大，会干扰、影响数据集当中的其他特征。

标准化的过程为两步：去均值的中心化（均值变为 0）；方差的规模化（方差变为 1）。

在 sklearn.preprocessing 模块当中内置有 scale 函数。

案例 3-6 的文件名是 kb306_pre01.py，本案例用于介绍 scale 函数的使用以及数据标准化的具体编程方法。

前面 3 组代码和以前的案例类似，先读取数据，再进行准备工作，代码如下：

```
#1
print('#1,rd data')
rss='data/'
stklst=['btc2018','eth2018','doge2018']
stkPools=zdat.pools_frd4lst(rss,stklst)

#2
print('#2,edit data')
df9=zdat.pools_lnk4lst(stkPools,stklst,'close')
zt.prDF('df9',df9)
```

```
#3
print('#3,cut data time')
tim0str,tim9str='2016-01-01','2016-12-31'
df2=zdat.df_kcut8tim(df9,'',tim0str,tim9str)
df2=df2.sort_index()
#
df2=df2.tail(10)
zt.prDF('df2',df2)
df2.plot()
```

这里我们使用了 BTC（比特币）、DOG（狗狗币）、ETH（以太币）3 种不同的数字货币作为数据归一化章节案例的数据源，这 3 种货币的市值差距较大，便于对比说明。

代码当中使用了前面我们学过的股票池数据读取合并技术，对应的输出信息如下：

```
#3,cut data time

 df2
            btc2018  eth2018  doge2018
time
2016-12-22   864.54     7.55       0.0
2016-12-23   921.98     7.16       0.0
2016-12-24   898.82     7.24       0.0
2016-12-25   896.18     7.17       0.0
2016-12-26   907.61     7.26       0.0
2016-12-27   933.20     7.18       0.0
2016-12-28   975.92     7.60       0.0
2016-12-29   973.50     8.32       0.0
2016-12-30   961.24     8.15       0.0
2016-12-31   963.74     8.01       0.0
```

另外，因为我们稍后在调用 scale 函数时，会涉及 list 列表内部单个数据的数值，所以，以上代码中只使用最后的 10 个数据值作为分析数据，语句如下：

```
df2=df2.tail(10)
```

图 3.4 是对应的价格曲线图。

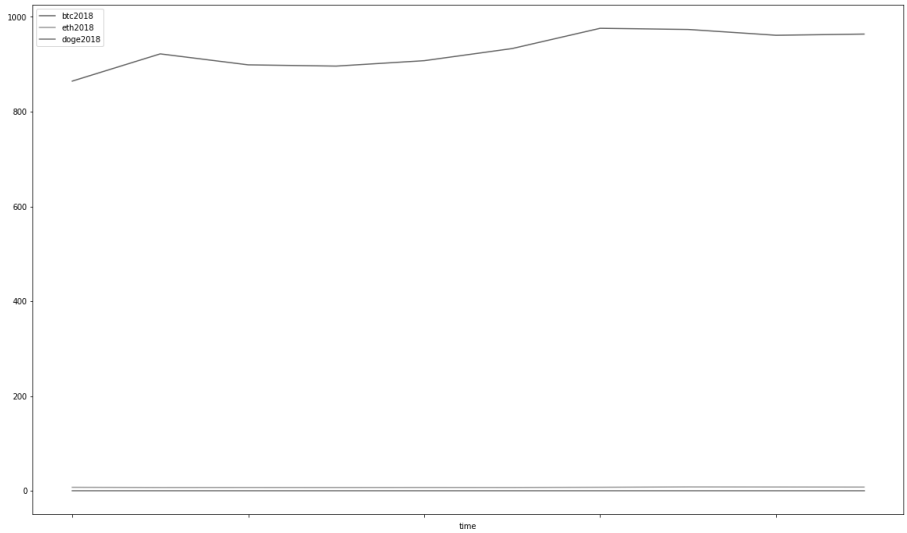


图 3.4　对应的价格曲线图

由图 3.4 可以看出，由于 BTC（比特币）的价格远远高于 DOG（狗狗币）、ETH（以太币），后两种数字货币的价格曲线类似直线，无法正确反映价格数据。

第 4 组代码如下：

```
#4
print('#4,edit data')
x=df2[stklst].values
print('x',x)
x10 = preprocessing.scale(x)
print('x10',x10)
```

调用 scale 函数，对数据进行标准化处理，对应的输出信息如下：

```
#4,edit data
x [[8.64540e+02 7.55000e+00 2.25000e-04]
 [9.21980e+02 7.16089e+00 2.36000e-04]
 [8.98820e+02 7.23712e+00 2.34000e-04]
```

```
 [8.96180e+02 7.16800e+00 2.30000e-04]
 [9.07610e+02 7.26000e+00 2.33000e-04]
 [9.33200e+02 7.18229e+00 2.27000e-04]
 [9.75920e+02 7.60000e+00 2.23000e-04]
 [9.73500e+02 8.31666e+00 2.28000e-04]
 [9.61240e+02 8.15443e+00 2.28000e-04]
 [9.63740e+02 8.00851e+00 2.23000e-04]]
x10 [[-1.80017331 -0.03269951 -0.86233265]
 [-0.21262238 -0.95537578  1.70135901]
 [-0.85272822 -0.77461556  1.23523326]
 [-0.92569365 -0.93851621  0.30298174]
 [-0.60978649 -0.7203614   1.00217038]
 [ 0.09748071 -0.90463108 -0.39620689]
 [ 1.27819408  0.08586289 -1.32845841]
 [ 1.2113091   1.78524141 -0.16314401]
 [ 0.87246205  1.40055387 -0.16314401]
 [ 0.94155811  1.05454137 -1.32845841]]
```

需要注意的是，sklearn 使用 NumPy 的 ndarray 数组格式，所以先要转换一下，再调用函数。

第 5 组代码如下：

```
#5
print('#5,data anz')
x_mean=np.round(x10.mean(axis=0),2)
x_std=x10.std(axis=0)
print('x_mean',x_mean)
print('x_std',x_std)
```

验证 scale 函数，查看标准化后的效果，对应的输出信息如下：

```
#5,data anz
x_mean [-0. -0. -0.]
x_std [1. 1. 1.]
```

从以上输出信息可以看到，标准化后各组数据的均值和方差已经分别变成 0 和 1 了。

第 6 组代码如下：

```
#6
print('#6,data anz')
x20=x10.T
print('x20',x20)
#
df3=pd.DataFrame()
df3['btc'],df3['eth'],df3['dog']=x20[0],x20[1],x20[2]
zt.prDF('df3',df3)
df3.plot()
```

这是附带的可视化分析，需要注意的是，scale 函数标准化处理后要用内置的 T 函数进行转置处理，才能传递给 DataFrame。图 3.5 是对应的曲线图。

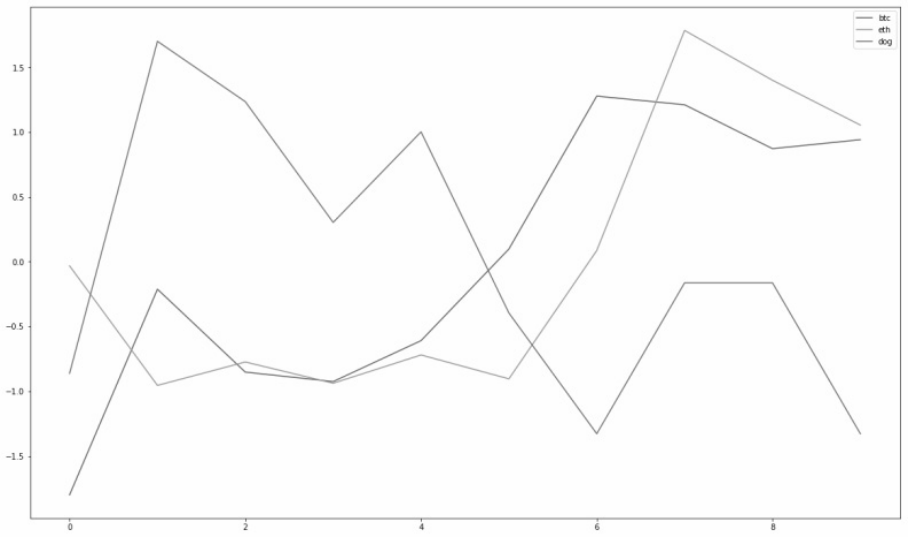


图 3.5　标准化处理数据曲线图

案例 3-7：比特币与归一化

数据标准化和数据归一化，是数据预处理的两个主要手段。

归一化的具体作用是归纳统一样本的统计分布性。归一化在(0,1)之间是统计的概率分布，归一化在(–1,1)之间是统计的坐标分布。之所以需要将特征规一化到一定的范围内，是为了应对那些标准差相当小的特征并且保留下稀疏数据中的 0 值。

归一化有同一、统一和合一的意思，归根结底，神经网络是以统计概率作为核心运算模式的。

从运作流程的角度而言，其实也可以不进行归一化处理。不过，归一化可以增加训练神经网络的收敛性，提高模型运算效率。通常在模型训练前对数据进行归一化处理。

在 sklearn 模块库中提供了 MinMaxScaler、MaxAbsScaler 等函数用于数据归一化处理。

案例 3-7 的文件名是 kb307_pre02.py，本案例用于介绍数据归一化函数的使用。

案例 3-7 的代码与前面案例代码大体相同，只是第 4 组代码不同，如下所示。

```
#4
print('#4,edit data')
x=df2[stklst].values
print('x',x)
xfun = preprocessing.MinMaxScaler()
x10= xfun.fit_transform(x)
print('x10',x10)
```

调用 MinMaxScaler 函数进行数据归一化处理。大家也可以调用 MaxAbsScaler 函数，看看两者之间的具体差异。

图 3.6 是对应的归一化数据曲线图。

从图 3.6 可以看出，经过归一化处理后，数据的分布都在(0,1)之间。

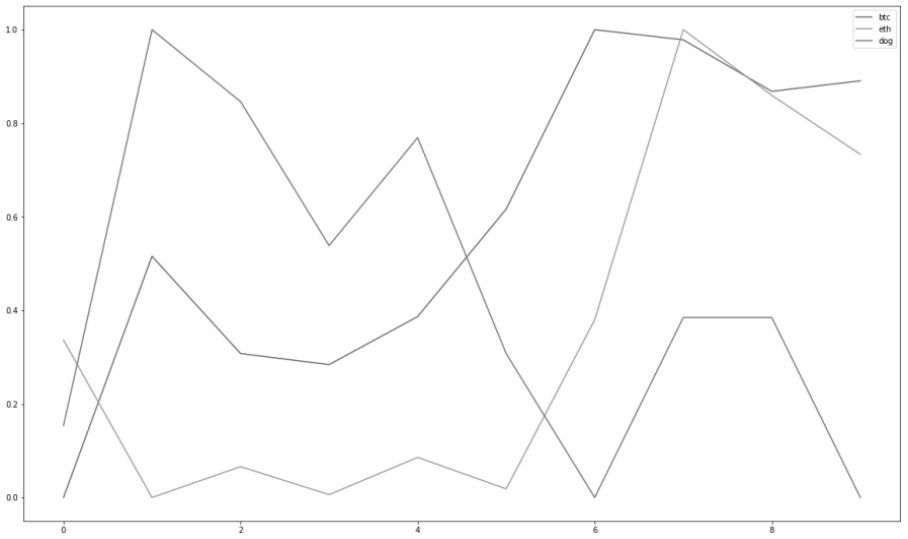


图 3.6　归一化数据曲线图

第 4 章

机器学习快速入门

对于机器学习、神经网络而言，数据是根基。而机器学习、神经网络，就是通过现代计算机的高速运算手段，不断地从各个角度对数据进行组合、分析，从中找出一些内在的规律、模式。

本章我们通过回归算法模型，找一组完整的案例，介绍数据准备、转换、模型建立与训练、数据预测等内容，以及最终的效果评估方法。

4.1 回归算法

Regression（回归）算法，是最常见的机器学习算法之一，其原理是通过对误差的衡量，来探索变量之间的关系。

回归算法本质上是基于数据统计分析的机器学习算法，目前已被纳入统计机器学习之中。

回归算法用于估计两种变量之间关系的统计过程。当用于分析因变量与一个或多个自变量之间的关系时，该算法能提供很多建模和分析多个变量的技巧。

具体而言，回归算法分析可以帮助我们理解当任意一个自变量变化而另一个自变量不变时，因变量变化的典型值。最常见的是回归算法分析能

在给定自变量的条件下估计出因变量的条件期望。

在图 4.1 中，利用回归算法，将垃圾邮件和非垃圾邮件进行了区分。

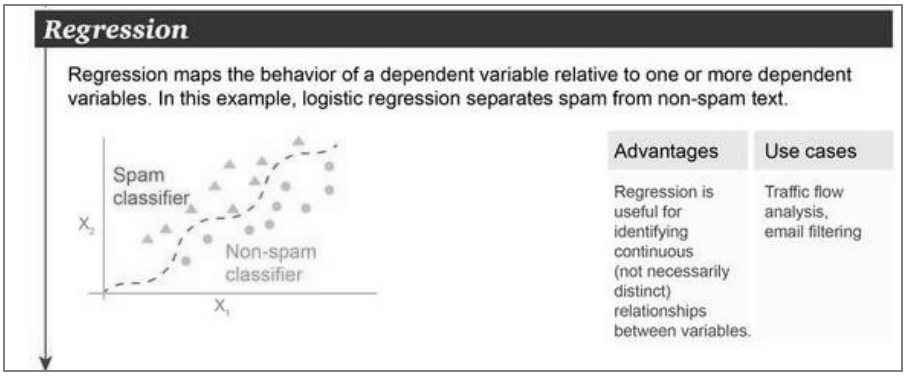


图 4.1　回归算法区分垃圾和非垃圾邮件

- 回归算法的优点：直接、快速、知名度高，可用于识别变量之间的连续关系，即使这个关系不是非常明显。
- 回归算法的缺点：要求严格的假设，需要处理异常值。

常用的回归算法模型如下。

- 普通最小二乘回归（Ordinary Least Squares Regression，OLSR）。
- 线性回归（Linear Regression）。
- 逻辑回归（Logistic Regression）。
- 逐步回归（Stepwise Regression）。
- 多元自适应回归样条（Multivariate Adaptive Regression Splines，MARS）。
- 本地散点平滑估计（Locally Estimated Scatterplot Smoothing，LOESS）。

4.2　LR线性回归模型

线性回归算法（Linear Regression），简称 LR 算法，可能是统计学和机器学习中最为知名、最易于理解的一种算法。

线性模型是最简单的模型，已有两百多年的历史了，其原理是用二元一次回归方程，根据 X 轴值的变化，生成用于分类和回归 Y 值最适合的一条线，如图 4.2 所示。

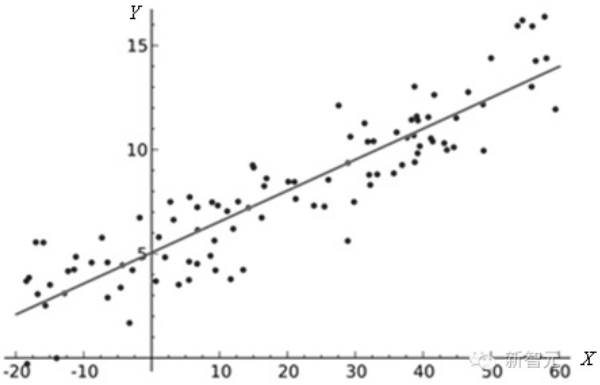


图 4.2　线性回归算法

有学者称，从本质上来说，所有的神经网络模型都是线性回归模型，这句话虽然有些偏颇，但也充分说明了线性回归模型的重要性。

线性回归算法的核心思想是基于最小二乘法方程式：

$$y = a \times x + b$$

其中的线性指的是用于拟合数据的模型，而最小二乘法指的是待优化的损失函数。

线性回归算法，通常用于根据连续变量估计实际数值（房价、呼叫次数、总销售额等）。例如你手头有附近一组房屋的大小和价格，就能用线性模型预测给定大小的房屋价格。

需要指出的是，线性模型可以接受多个 x 特征输入。例如上面的房屋例子中，我们可以根据房子大小、房间数量和浴室数量以及价格来构建一个线性模型，然后利用这个线性模型和给定房子的大小，房间以及浴室个数来预测价格。

线性回归的两种主要类型：一元线性回归和多元线性回归。

一元线性回归的特点是只有一个自变量。多元线性回归的特点正如其名，存在多个自变量，在找最佳拟合直线的时候，你可以拟合到多项或者曲线回归，这些就被叫作多项或曲线回归。

线性回归模型拟合的方法有许多种，如常用的最小二乘法，以及目前最热门的梯度下降算法等。

在 Sklearn 模块库中有多种不同的线性回归函数，都位于 linear_model 模块中，函数名及说明如下。

- LinearRegression：最小二乘法线性回归函数。
- LogisticRegression：逻辑回归算法。
- Ridge：岭回归函数。
- Lasso：Lasso（套索）回归算法。
- MultiTaskLasso：多任务 Lasso（套索）回归算法。
- ElasticNet：ElasticNet（弹性网眼）算法。
- MultiTaskElasticNet：多任务 ElasticNet（弹性网眼）算法。
- LARS：LARS（最小角回归）算法。
- LassoLars：LARS 套索算法。
- OrthogonalMatchingPursuit：正交匹配追踪（OMP）算法。
- BayesianRidge：贝叶斯岭回归算法。
- ARDRegression：ARD（自相关）回归算法。
- SGDClassifier：SGD（随机梯度下降）算法。
- Perceptron：感知器算法。
- PassiveAggressiveClassifier：PA（被动感知）算法。
- RANSACRegressor：鲁棒回归算法。
- HuberRegressor：Huber 回归算法。
- TheilSenRegressor：Theil-Sen 回归算法。

- PolynomialFeatures：多项式函数回归算法。

案例 4-1：上证指数之 LR 回归事件

案例 4-1 的文件名是 kb401-ln.py，本案例是上证指数的线性回归模型版本，介绍如何建立收益线性回归模型，预测上证指数第二天的收盘价格。

前面几组代码都很简单，就是读取数据，并且转换为 sklearn 机器学习所需要的格式，相关代码如下：

```
#1
print('#1,rd data')
fss='data/sh2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
zt.prDF('xdf',xdf)

#2
print('#2,xed data')
xdf['y']=xdf['close'].shift(-1)
zt.prDF('xdf#2.1',xdf)
xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf#2.2',xdf)



#3.1
print('#3,cut data')
tim0str,tim9str='2010-10-10','2017-12-31'
df_train=zdat.df_kcut8tim(xdf,'',tim0str,tim9str)
zt.prDF('#3.1 df_train\n',df_train)
#
#3.2
df_test=xdf[xdf.index>'2018'] #.tail(100)
zt.prDF('#3.2 df_test\n',df_test)
```

```
#------------------------
#4
print('#4,准备 AI 数据')

clst=['open','high','low','close']
x=df_train[clst].values
y=df_train['y'].values
#
xtst=df_test[clst].values
```

以上代码需要注意的是第 4 组：

```
print('#4,准备 AI 数据')

clst=['open','high','low','close']
x=df_train[clst].values
y=df_train['y'].values
#
xtst=df_test[clst].values
```

按 sklearn 需要的 NumPy 的 ndarray 数组格式准备好相关的机器学习训练、测试数据，其中的要点在于取字段的 values 值，很多初学者都容易在这里出现问题。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = linear_model.LinearRegression()
```

从 sklearn 的 linear_model 子模块调用 LinearRegression 函数，生成线性模型，并保存到变量 mx 中。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

调用模型的内置训练函数 fit，通过训练数据集 x、y，对模型进行训练。通常，这个阶段是耗时最多的。

sklearn 的 fit 训练函数和 predict 预测函数是机器学习中非常简单的 API 接口，也是各种神经网络、深度学习的简化接口标准，包括 tflearn、keras 等模块库，都参考了这种设计。

这种模式相对于 TensorFlow 等神经网络平台的 batch 迭代模型的确简单明了，但对于模型当中的参数细节优化却无能为力。

第 7 组代码如下：

```
    #7
    print('#7,predict 模型预测数据')
    df_test['y2']=mx.predict(xtst) #cross_val_predict(mx,x, y,
cv=20)
    zt.prDF('df',df_test)
    df_test[['y','y2']].plot()
```

对训练好的模型变量 mx，通过内置的 predict 预测函数按测试数据 xtst 产生预测数据。

对应的输出信息如下：

```
     df
                 open    high    close     low      volume
amount       y      y2
    date
    2018-01-02  3314.03  3349.05  3348.33  3314.03  20227886000
227788461113  3369.11  3350.08
    2018-01-03  3347.74  3379.92  3369.11  3345.29  21383614900
258366523235  3385.71  3370.36
    2018-01-04  3371.00  3392.83  3385.71  3365.30  20695528800
243090768694  3391.75  3384.97
    2018-01-05  3386.46  3402.07  3391.75  3380.24  21306068100
248187840542  3409.48  3390.09
    2018-01-08  3391.55  3412.73  3409.48  3384.56  23616510600
286213219095  3413.90  3408.34
```

```
    2018-01-09  3406.11  3417.23  3413.90  3403.59  19148855100
238249975070  3421.83  3411.70
    2018-01-10  3414.11  3430.21  3421.83  3398.84  20909499700
254515441261  3425.34  3419.21
    2018-01-11  3415.58  3426.48  3425.34  3405.64  17381213300
218414134129  3428.94  3422.29
    2018-01-12  3423.88  3435.42  3428.94  3417.98  17406340400
215961455748  3410.49  3426.49
    2018-01-15  3428.95  3442.50  3410.49  3402.31  23200928300
286362732919  3436.59  3406.94

                open    high    close     low      volume
amount       y      y2
    date
    2018-05-11  3179.80  3180.76  3163.26  3162.21  13065974900
167364290366  3174.03  3159.28
    2018-05-14  3167.04  3183.82  3174.03  3163.48  12932735300
172410691054  3192.12  3173.56
    2018-05-15  3180.42  3192.81  3192.12  3164.52  12454905100
162990790010  3169.56  3189.51
    2018-05-16  3180.23  3191.95  3169.56  3166.81  13052496800
174590979834  3154.28  3167.68
    2018-05-17  3170.01  3172.77  3154.28  3148.62  11399556700
150598842185  3193.30  3150.25
    2018-05-18  3151.08  3193.45  3193.30  3144.78  13651691800
168038057477  3213.84  3196.06
    2018-05-21  3206.18  3219.74  3213.84  3203.34  16445941300
202663464515  3214.35  3212.73
    2018-05-22  3211.25  3214.59  3214.35  3192.23  14429268400
185721667752  3168.96  3210.01
    2018-05-23  3205.44  3205.44  3168.96  3168.96  15780764800
199358101015  3154.65  3163.40
    2018-05-24  3167.94  3173.53  3154.65  3152.07  12408580000
160658185502  3154.65  3151.59
```

预测数据保存在 df_test 的 y2 字段，对应的真实数据保存在 y 字段，请注意两者的不同。

图 4.3 是预测数据和真实数据的对比图。

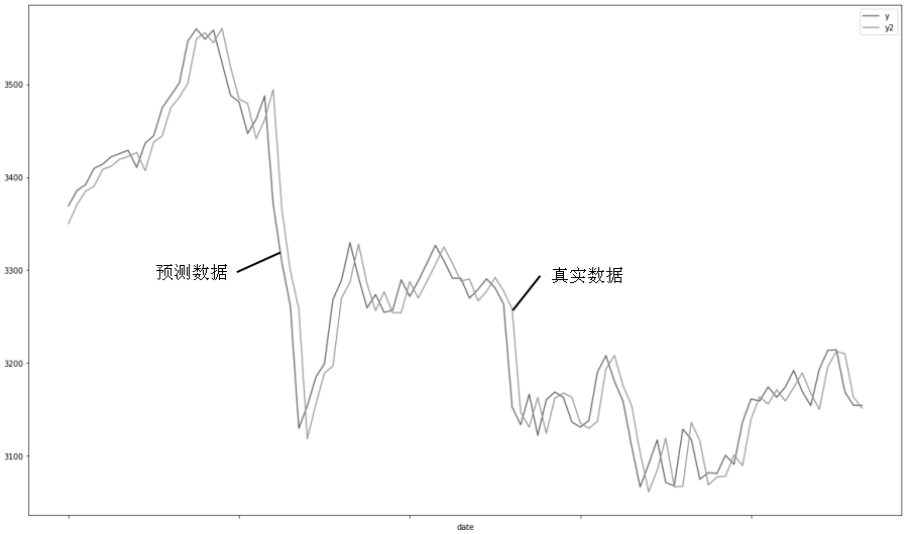


图 4.3　预测数据和真实数据的对比图

第 8.1 组代码如下：

```
#8.1
print('#8,验证模型预测效果')
print('\n#8.1,按 5%精度验证模型')
dacc,df=zai.ai_acc_xed2x(df_test['y'],df_test['y2'],5,True)
print('acc',dacc)
```

按 5%的精度验证模型的准确度，对应的结果是：

```
#8.1,按 5%精度验证模型
acc 100.0
```

具体数据验证是通过极宽 zai 模块库的 ai_acc_xed2x 函数计算的，有关细节请大家参看相关代码。

验证结果在 5%的误差范围内，准确度是 100%。

第 8.2 组代码如下：

```
    #8.2
    print('\n#8.2,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['y'],df_test['y2'],1,
True)
```

按 1%的精度，验证模型的准确度，对应的结果是：

```
    #8.2,按 1%精度验证模型
    ky0=1; n_df9,94,n_dfk,69
    acc: 73.40%;  MSE:1132.60, MAE:24.89,  RMSE:33.65, r2score:0.94,
@ky0:1.00
```

验证结果在 1%的误差范围内，准确度是 73.4%。

4.3　常用评测指标

在上文案例中，当误差精度在 1%时，有 70%左右的准确度，数据看起来还不错，但没有任何实盘价值。当误差精度在 5%时，模型准确度高达 100%，这个准确度看起来很高，其实也没有任何实盘价值。

股市大盘指数、日线的波动一般都在 3%以内，这种模型虽然预测精度高达 95%，但是在正常的波动范围内，这类似预测巴西队和中国队的足球比赛结果一样，随便采用什么模型，都可以达到 95%的预测精度效果。

案例中的 3%～5%的误差精度围绕真实数据上下波动 3%～5%，这个震荡区间可能高达 100%。

这种误差精度无法用于实盘分析，实盘分析时波动的误差精度往往需要在万分之一以内，外汇日内交易更是需要在十万分之一的误差精度。

此外，模型的预测精度虽然高达 95%，但是案例当中的预测数据，包含上涨、下跌两种可能，无法实际反映次日的涨跌情况，还需要进一步分析。

本书目前使用的是大盘指数单一的数据，而且也只是使用其中的一组数据作为模型训练数据。在实盘分析中建立模型和使用模型训练数据时，往往需要各种股票的数据，而且配合各种参数一起进行训练。

ai_acc_xed2ext 效果评估函数当中的 MAE、MSE、RMSE 是统计术语，用于结果评测，具体含义如下。

- MAE：Mean Absolute Error，平均绝对误差。
- MSE：Mean Squared Error，均方差、方差，结果数据越接近于 0，说明模型选择和拟合度更好，数据预测也越成功。
- RMSE：Root Mean Squared Error，均方根、标准差，结果数据越接近于 0，说明模型选择和拟合度更好，数据预测也越成功。

还有以下类似术语。

- SSE：The Sum of Squares Due to Error，方差、误差平方和。SSE 越接近于 0，说明模型选择和拟合度更好，数据预测也越成功。MSE 和 RMSE 与 SSE 同出一宗，所以效果一样。
- R-square：Coefficient of Determination，确定系数。确定系数通过数据的变化来表征一个拟合算法的好坏，正常取值范围为[0, 1]，越接近 1，表明方程的变量对 y 的解释能力越强，算法模型对数据的拟合度也越好。
- Adjusted R-square：Adjusted Coefficient of Determination，修正确定系数。
- Precision：又称 P 指标，表示准确率，即检索出来的条目（比如文档、网页等）有多少是准确的。
- Recal：又称 R 指标，表示召回率，即所有准确的条目有多少被检索出来了。
- F 值（F-Measure）：又称为 F-Score，F 值是正确率和召回率的调和平均值；F 值= 正确率 × 召回率 × 2 / (正确率 + 召回率)，F 值即正确率和召回率的调和平均值。
- E 值：表示查准率 P 和查全率 R 的加权平均值，当其中一个为 0 时，E 值为 1。
- AP：Average Precision，平均正确率，表示不同查全率的点上的正确率

的平均值。

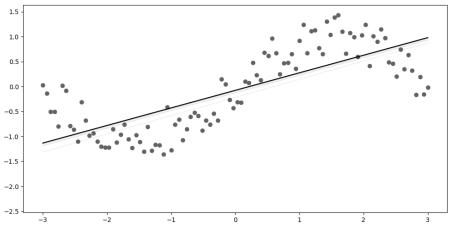
- R2 决定系数（拟合优度）：表示回归方程在多大程度上解释了因变量的变化，或者说方程对观测值的拟合程度如何。

sklearn 专门内置了一个 metrics 评估子模块，提供多种方式的评估函数。

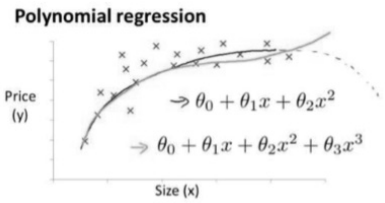
对于量化交易而言，需要采用简单、灵活，以及更加适合金融数据分析的评估函数，所以这里采用的是我们自己开发的评估函数。

4.4　多项式回归

在前面的案例中，我们使用的是线性回归模型。不过从上下波动的财经曲线可以明显看出线性回归模型并不适用于金融市场数据，我们需要其他模型来适应我们的数据，比如一个二次方模型，或者一个三次方模型。图 4.4 是线性模型和多项式回归模型的示意图。



线性回归



多项式回归

图 4.4　线性模型和多项式回归模型

从上图中可以看出，多项式回归模型更加适用于金融数据分析。

案例 4-2：上证指数的多项式故事

案例 4-2 的文件名是 kb402-poly.py，本案例通过多项式回归模型来预测第二天上证指数的收盘价。

在 sklearn 模块库当中，并没有独立的多项式回归模型函数，而是直接

调用 LinearRegression（线性回归）函数。这是因为线性回归本身就是多项式回归模型的一次方简化形式。

案例 4-2 多项式回归模型与案例 4-1 的线性回归模型类似，下面我们只对代码当中不同的地方进行说明。

第 4 组代码如下：

```
#4
print('#4,准备 AI 数据')
clst=['open','high','low','close']
x0=df_train[clst].values
y=df_train['y'].values

#
xtst0=df_test[clst].values
```

在数据准备部分中，因为多项式回归模型需要对输入数据进行高次元变换，所以变量名 x 和 xtst 分别加了后缀 0，表示原始数据。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = linear_model.LinearRegression()



# 5.b
print('#5,多项式参数变换')
poly = PolynomialFeatures(degree=2,include_bias=False)
x = poly.fit_transform(x0)
xtst = poly.fit_transform(xtst0)
```

第 5 组代码开头和线性回归模型是相同的。

```
mx = linear_model.LinearRegression()
```

也就是调用 LinearRegression 函数产生线性模型。

不同之处在于 5.b 部分的新增代码，通过 PolynomialFeatures 函数对输入数据进行高次元变换。

其他部分代码与线性回归模型的代码完全一样，程序运行的结果如下：

```
    #8,验证模型预测效果

    #8.1,按 5%精度验证模型
    acc 100.0

    #8.2,按 1%精度验证模型
    ky0=1; n_df9,94,n_dfk,70
    acc: 74.47%;  MSE:1094.09, MAE:23.99,  RMSE:33.08, r2score:0.94,
@ky0:1.00
```

在 1%的误差范围内，准确度是 74.47%，比线性模型的 73.4%高一些。

5.b 部分的新增代码中 degree=2，表示进行二次方变换，用户可以自己设置为 3、4、5 等高次方，也可以设置为 1、0 等低次方，看看运行结果。

笔者的测试结果如下。

- degree=0，运行出错。
- degree=1，准确度 73.4%，和线性回归案例一样。
- degree=2，准确度 74.47%。
- degree=3，准确度 70.21%。
- degree=5，准确度 41.49%。
- degree=10，准确度 5.32%。

当 degree 参数越大，计算的方程式越复杂，时间越长，不过结果却并非最好。

初步测试发现，当 degree=2 时，准确度最高。

图 4.5 是 degree=10 时的预测数据与真实数据对比曲线图。

在图 4.5 当中，平滑的一条线是真实数据的曲线，波动大的线是高次元模型预测数据的曲线。

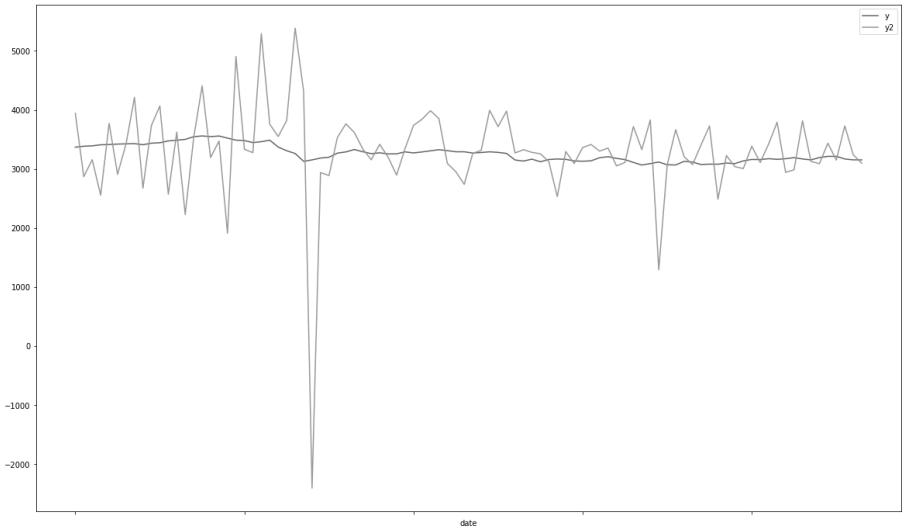


图 4.5　十次元多项式模型预测数据与真实数据对比曲线图

案例 4-3：预测比特币价格

近年来，数字货币市场风起云涌，特别是比特币价格，2018 年年初一度冲破两万美元关口。

案例 4-3 的文件名是 kb403-polybtc.py，本案例通过多项式回归模型来预测第二天比特币的收盘价。

案例 4-3 比特币价格预测与上证指数多项式回归模型版本除了数据源不同以外，其他完全相同，这也说明了我们案例程序的灵活性。

不同之处就在于第 1 组代码，数据文件名称不同：

```
#1
print('#1,rd data')
fss='data/btc2018.csv'
```

虽然只是数据源的不同，不过比特币和上证指数的运行结果却差别很大。

以下是程序运行结果数据：

```
    #8,验证模型预测效果

    #8.1,按 5%精度验证模型
    acc 54.867

    #8.2,按 1%精度验证模型
    ky0=1; n_df9,113,n_dfk,16
    acc: 14.16%;  MSE:511920.68, MAE:533.96,  RMSE:715.49,
r2score:0.92, @ky0:1.00
```

从结果数据当中我们可以看出，即使是 5%的容错度，模型准确度也只有 54.8%；如果采用 1%的容错度，模型准确度只有 14.16%，远远低于上证指数的案例。

这充分说明数字货币市场的残酷性，每天的波动是传统金融市场的 *n* 倍。这也间接证明了：数字货币和量化交易是天生的一对。

面对每天如此巨大的波动，传统的日线数据完全没有价值。在实盘当中，一般使用 5 分钟、15 分钟的分时数据，这个对于算力、量化平台的要求远远高于传统金融投资领域。

4.5　逻辑回归算法模型

逻辑回归算法模型（Logistic Regression）虽然名称里面有回归二字，不过却是一个分类算法。逻辑回归算法虽然也属于线性回归算法，但由于使用较多，目前已经成为一个独立的机器学习算法类别，如图 4.6 所示。

简单来说，逻辑回归算法通过将数据拟合进一个逻辑函数，来预估一个事件出现的概率。因此，它也被叫作逻辑回归。

逻辑回归类似线性回归，因为它的目标是找出每个输入变量的加权系数值。与线性回归不同的是，逻辑回归预测输出值的函数是非线性的，也被称为逻辑函数。

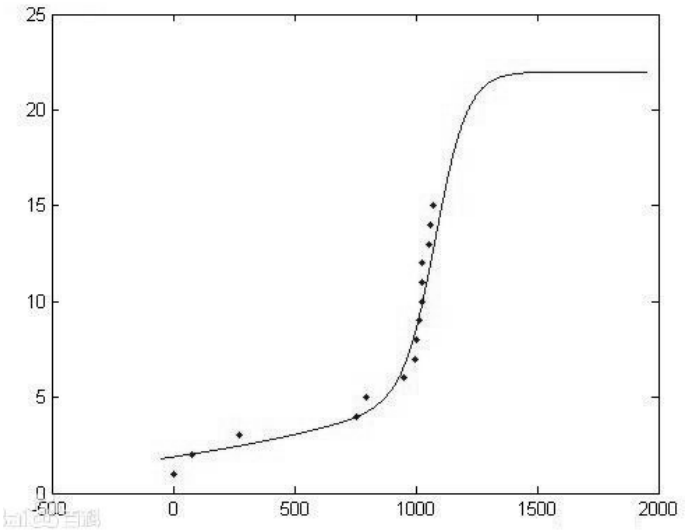


图 4.6　逻辑回归算法

逻辑回归算法函数位于 sklearn 模块库的 linear_model 线性回归子模块，函数接口是：

```
    LogisticRegression(penalty='l2', dual=False, tol=0.0001, C=1.0,
fit_intercept=True, intercept_scaling=1, class_weight=None,
random_state=None, solver='liblinear', max_iter=100,
multi_class='ovr', verbose=0, warm_start=False, n_jobs=1)
```

案例 4-4：上证指数预测逻辑回归版

案例 4-4 的文件名是 kb404_log.py，本案例介绍逻辑回归算法在上证指数预测当中的应用。

逻辑回归算法的输入类似线性回归模型，不过也有很多不同之处，程序代码如下。

第 1 组代码如下：

```
    #1
    print('#1,rd data')
```

```
    fss='data/sh2018.csv'
    xdf=pd.read_csv(fss,index_col=0)
    xdf=xdf.sort_index()
    zt.prDF('xdf',xdf)
```

此处为读取数据，这个和案例 4-1 完全相同。

第 2 组代码如下：

```
    #2
    print('#2,xed data')
    xdf['xprice']=xdf['close'].shift(-1)
    xdf['y']=0
    xdf.loc[xdf.xprice>xdf.close,'y']=2
    xdf.loc[xdf.xprice<xdf.close,'y']=1
    xdf.fillna(method='pad',inplace=True)
    zt.prDF('xdf',xdf)
```

第 2 组代码与前面的案例 4-1 差异很大，其中使用了很多在实盘操作中的技巧。

为了方便讲解，我们首先看程序的输出信息：

```
    len-DF: 1822
    #2,xed data


     xdf
                open    high    low   close  vol      cap  xprice  y
    time
    2013-04-28  135.30  135.98  132.10  134.21   -1  1500520000
144.54  2
    2013-04-29  134.44  147.49  134.00  144.54   -1  1491160000
139.00  1
    2013-04-30  144.00  146.93  134.05  139.00   -1  1597780000
116.99  1
    2013-05-01  139.00  139.89  107.72  116.99   -1  1542820000
105.21  1
    2013-05-02  116.38  125.60   92.28  105.21   -1  1292190000
97.75  1
```

```
    2013-05-03  106.25  108.13   79.10   97.75   -1  1180070000
112.50  2
    2013-05-04   98.10  115.00   92.50  112.50   -1  1089890000
115.91  2
    2013-05-05  112.90  118.80  107.14  115.91   -1  1254760000
112.30  1
    2013-05-06  115.98  124.66  106.64  112.30   -1  1289470000
111.50  1
    2013-05-07  112.25  113.44   97.70  111.50   -1  1248470000
113.57  2

                open    high     low   close       vol
cap   xprice  y
    time
    2018-04-14  7874.67  8140.71  7846.00  7986.24  5191430000
133682000000  8329.11  2
    2018-04-15  7999.33  8338.42  7999.33  8329.11  5244480000
135812000000  8058.67  1
    2018-04-16  8337.57  8371.15  7925.73  8058.67  5631310000
141571000000  7902.09  1
    2018-04-17  8071.66  8285.96  7881.72  7902.09  6900880000
137070000000  8163.42  2
    2018-04-18  7944.43  8197.80  7886.01  8163.42  6529910000
134926000000  8294.31  2
    2018-04-19  8159.27  8298.69  8138.78  8294.31  7063210000
138591000000  8845.83  2
    2018-04-20  8286.88  8880.23  8244.54  8845.83  8438110000
140777000000  8895.58  2
    2018-04-21  8848.79  8997.57  8652.15  8895.58  7548550000
150337000000  8802.46  1
    2018-04-22  8925.06  9001.64  8779.61  8802.46  6629900000
151651000000  8930.88  2
    2018-04-23  8794.39  8958.55  8788.81  8930.88  6925190000
149448000000  8930.88  0
```

逻辑回归模型属于分类模型。事实上，绝大部分机器学习、神经网络

模型都属于分类模型，真正用于数值分析的模型只占小部分。

为了分类，我们需要对目标数据 y 值进行处理，这里采用的是类似足彩的“3、1、0，胜、平、负”模式，不过略有修改，具体如下。

- 数值 2，代表第二天的收盘价高于当天。
- 数值 1，代表第二天的收盘价低于当天。
- 数值 0，代表第二天的收盘价和当天持平。

因为是浮点数据，所以持平的数据很少。在实盘当中，为了精确会采用±0.5%或者其他幅度作为持平的标准。本案例中为了简化，直接采用数学比较模式。

程序中具体的分类语句如下：

```
xdf.loc[xdf.xprice>xdf.close,'y']=2
xdf.loc[xdf.xprice<xdf.close,'y']=1
```

以上代码，使用了 pandas 的 loc 定位和逻辑判断的组合模式，是编程常用的实战技巧，相关细节请参考 pandas 用户手册。

第 3 组、第 4 组代码如下：

```
#3.1
print('#3,cut data')
tim0str,tim9str='2010-10-10','2017-12-31'
df_train=zdat.df_kcut8tim(xdf,'',tim0str,tim9str)
zt.prDF('#3.1 df_train\n',df_train)
#
#3.2
df_test=xdf[xdf.index>'2018'] #.tail(100)
zt.prDF('#3.2 df_test\n',df_test)

#4
print('#4,准备 AI 数据')

clst=['open','high','low','close']
x0=df_train[clst].values
```

```
    y=df_train['y'].values
    #
    xtst0=df_test[clst].values
```

此处按时间切分模型训练数据和测试数据，并准备机器学习需要的数据，这些和案例 4-1 类似。

第 5 组代码如下：

```
    #5
    print('#5,模型设置')
    mx = linear_model.LogisticRegression()
```

调用 sklearn 的 LogisticRegression（逻辑回归）函数，生成相关的模型，并保存在变量 mx 中。

第 6 组代码如下：

```
    #6
    print('#6,数据变换')
    ss = StandardScaler()
    x = ss.fit_transform(x0)
    xtst = ss.transform(xtst0)
```

因为逻辑回归模型对数据有严格要求，使用前要进行变换处理。

如果省略以上代码，程序运行会出错。

第 7 组代码如下：

```
    #7
    print('#7,fit 训练模型')
    mx.fit(x,y)
```

调用模型内置的 fit 函数，根据输入数据进行训练。

第 8 组代码如下：

```
    #8
    print('#8,predict 模型预测数据')
    df_test['y2']=mx.predict(xtst) #cross_val_predict(mx,x, y,
cv=20)
    #predicted = cross_val_predict(mx,x, y, cv=10)
    #df['y2']=predicted
```

```
zt.prDF('df',df_test)
df_test[['y','y2']].plot()
#
print("\ndf_test['y'].value_counts()")
print(df_test['y'].value_counts())
print("\ndf_test['y2'].value_counts()")
print(df_test['y2'].value_counts())
```

调用训练后的模型进行数据预测，这些与前面的案例 4-1 都类似。

注意最后几条语句：

```
print("\ndf_test['y'].value_counts()")
print(df_test['y'].value_counts())
print("\ndf_test['y2'].value_counts()")
print(df_test['y2'].value_counts())
```

因为是分类数据，所以使用 pandas 的 value_counts 函数进行简单的统计分析，对应的输出信息是：

```
df_test['y'].value_counts()
2    51
1    42
0     1
Name: y, dtype: int64

df_test['y2'].value_counts()
2    94
Name: y2, dtype: int64
```

在真实数据字段 y 当中，上涨、下跌、持平的数据是：51、42、1。

在可预测数据字段 y2 当中，只有上涨数据：94。

这说明案例模型存在严重问题。

第 9 组代码如下：

```
#9
print('#9,验证模型预测效果')
#9.1
print('\n#9.1,按 5%精度验证模型')
```

```
    dacc,df=zai.ai_acc_xed2x(df_test['y'],df_test['y2'],5,True)
    print('acc',dacc)

    #9.2
    print('\n#9.2,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['y'],df_test['y2'],1,
True)
```

对应的输出信息是：

```
    #9,验证模型预测效果

    #9.1,按 5%精度验证模型
    acc 54.255

    #9.2,按 1%精度验证模型
    ky0=1; n_df9,94,n_dfk,51
    acc: 54.26%;  MSE:0.49, MAE:0.47,  RMSE:0.70, r2score:-0.81,
@ky0:1.00
```

图 4.7 是对应的分类对比图。

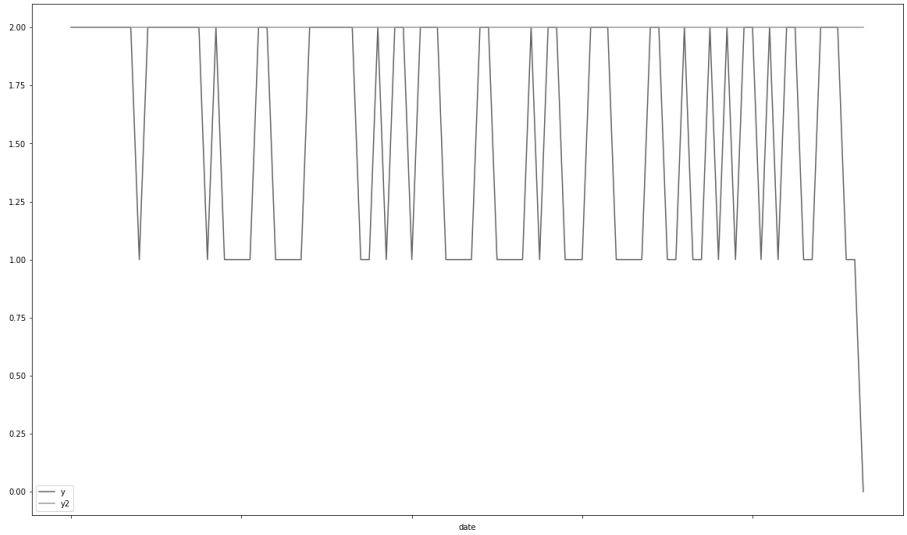


图 4.7　分类对比图形

在图 4.7 当中，没有波动的曲线表示 y2 预测数据，有上下波动的是真实数据 y。

虽然输入模型的验证结果在 5%和 1%的容错范围内，准确度都是 54%左右，但是结合图 4.7 的对比图形以及前面的预测数据字段 y2 的统计数据，可以看出在这个案例当中，采用逻辑回归模型直接进行预测分析是错误的。

本书中保留这个案例也是为了作为负面案例参考，以此说明并非所有的模型都可以简单直接地套用到实际工作当中。

第 5 章

模型验证优化

通常关于模型验证优化的内容都是放在教程的最后讲解，但是经验表明，对于初学者而言，模型的评估（比如谷歌 TensorFlow 深度学习平台当中的可视化分析模块 TensorBoard）越早学习，效果越好。

在 sklearn 模块库当中，内置了多种模型验证优化（Optimizers Model Validation）函数，常用的有如下几种。

- cross_val_score：交叉验证评估得分。
- cross_val_predict：对每一个输入数据点生成交叉验证评估器。
- permutation_test_score：评估置换交叉验证测试得分的重要性。
- learning_curve：学习曲线。
- validation_curve：验证曲线。

5.1 交叉验证评估器

交叉验证评估器，函数名称是 cross_val_predict，可以对每一组输入数据点生成交叉验证评估器。

对于机器学习而言，通常需要 train_data（训练数据）集和 test_data（测

试数据）集两个部分。

测试数据（test_data）的验证结果叫作错误验证值（validation_error）。

对于任何一种算法或者模型，若只用一次或者少数几次的数据进行训练、测试，就作为评估算法、模型好坏的标准，那么这样的结果存在很大的偶然性。

在实战当中，必须多次、随机划分训练数据集和测试数据集，分别计算多次的错误验证值。这样，根据多次的错误验证值才可以较为客观地评估算法、模型的好坏。

交叉验证是在数据量有限的情况下，采用的一种模型评估手段，它可以自动、随机地按不同比例对数据进行划分、测试。

sklearn 中的交叉验证模块最主要的函数是：

```
model_selection.cross_val_predict
```

函数调用接口如下：

```
cross_val_predict(estimator, X, y=None, groups=None, cv=None,
n_jobs=1, verbose=0, fit_params=None, pre_dispatch='2*n_jobs',
method='predict')
```

常用的参数如下。

- estimator：表示不同模型的变量。
- X：训练数据集。
- y：训练数据集所对应的正确结果数值。
- cv：代表的是不同的交叉验证方法。

需要注意的是参数 cv 比较复杂，既可以是自定义的验证函数名称，也可以是 int 整数数值。

当 cv 是 int 数值时，默认使用验证函数 StratifiedKFold，或者 KFold 交叉验证函数，如果指定了类别标签，则使用的交叉验证函数是 StratifiedKFold。

案例 5-1：交叉验证

案例 5-1 的文件名是 kb501_cross_val_predict.py，本案例用于介绍交叉验证函数 cross_val_predict 的使用方法，核心代码如下。

第 1 组、第 2 组代码如下：

```
#1
print('#1,rd data')
fss='data/sh2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
zt.prDF('xdf',xdf)

#2
print('#2,xed data')
xdf['y']=xdf['close'].shift(-1)
zt.prDF('xdf#2.1',xdf)
xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf#2.2',xdf)
```

此为读取数据。

第 3 组代码如下：

```
#3
print('#3,cut data')
df=xdf[xdf.index>'2018']
```

此为切分数据，需要注意的是，这里没有再把数据分为训练数据集和测试数据集两个部分，而是使用一个数据集。

这是因为交叉验证函数 cross_val_predict 会在内部对数据自动进行切分，而且会进行多次切分、测试，使模型训练效果更加客观。

第 4 组代码如下：

```
#4
print('#4,准备 AI 数据')
clst=['open','high','low','close']
```

```
x=df[clst].values
y=df['y'].values
```

把数据转换为 NumPy 的 ndarray 数组格式，以适合于 sklearn 函数调用。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = linear_model.LinearRegression()
```

此为生成训练模型，这里是最简单的线性模型。

第 6 组代码如下：

```
#6
print('#6,交叉验证')
df['y2']= cross_val_predict(mx,x, y, cv=20)
zt.prDF('df',df)
df[['y','y2']].plot()
```

这里没有使用 fit 函数训练模型，也没有使用 predict 函数生成预测数据，而是调用 cross_val_predict 交叉验证函数直接生成了预测数据。

第 7 组代码如下：

```
#7
print('#7,验证模型预测效果')
#9.1
print('\n#7.1,按 5%精度验证模型')
dacc,df2=zai.ai_acc_xed2x(df['y'],df['y2'],5,True)
print('acc',dacc)

#7.2
print('\n#7.2,按 1%精度验证模型')
dacc,df2,xlst=zai.ai_acc_xed2ext(df['y'],df['y2'],1,True)
```

此为调用自定义函数，验证模型效果。

为了简化流程，本案例调用 cross_val_predict 交叉验证函数时，cv 参数直接使用了 int 整数数值，可以理解为进行了 20 次拆分、验证训练。

按 1%的精度验证模型，结果如下。

- cv=20 时，准确度为 74.47%，如图 5.1 所示。
- cv=2 时，准确度为 41.49%，如图 5.2 所示。
- cv=1 或 cv=0 时，运行出错。

通常情况下，cv 的取值为 10 或者 20。

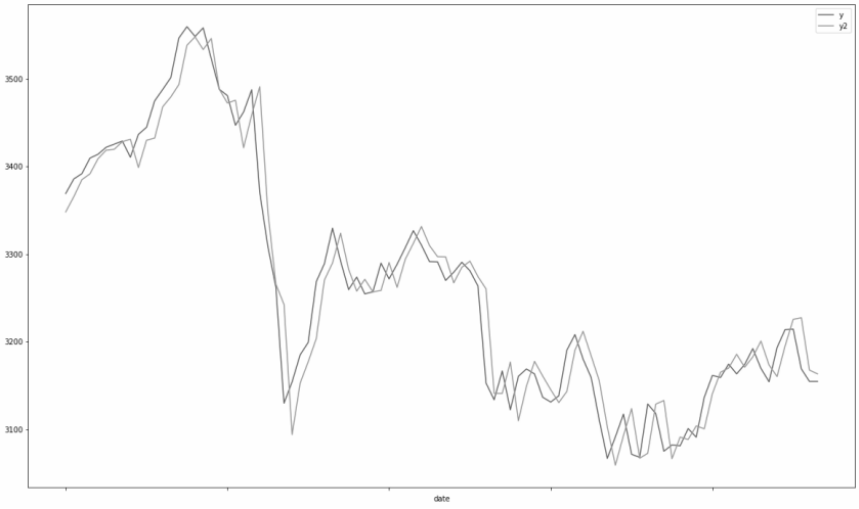


图 5.1　结果数据曲线图，cv=20

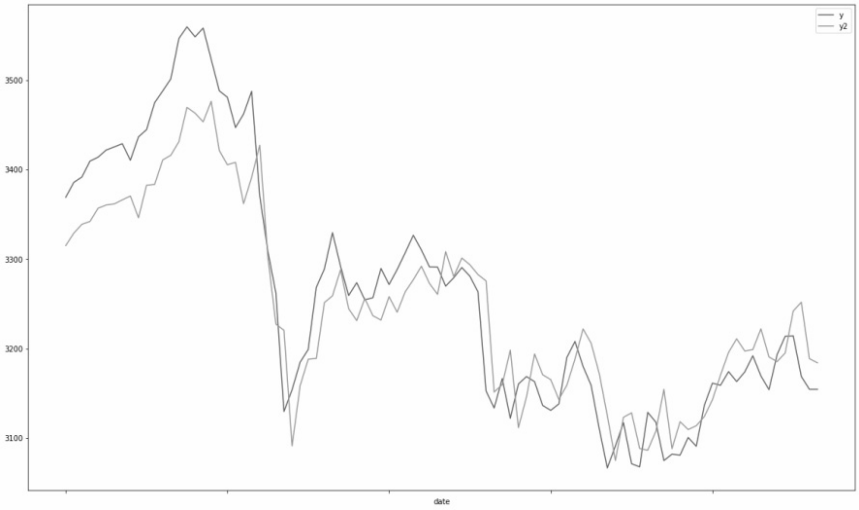


图 5.2　结果数据曲线图，cv=2

由图 5.1 和图 5.2 可以看出，当 cv 数值过小时，模型训练轮回次数就会变少，模型误差就较大。

5.2　交叉验证评分

在前面的案例当中，我们使用的是自定义的函数来对模型进行准确度评分，这是因为使用自定义评估函数更加灵活。

其实在 sklearn 模块库当中，也有基于交叉验证的评估函数，函数名称是：

```
    cross_validation.cross_val_score
```

函数调用接口如下：

```
    cross_val_score(estimator, X, y=None, groups=None, scoring=None,
cv=None, n_jobs=1, verbose=0, fit_params=None,
pre_dispatch='2*n_jobs')
```

案例 5-2：交叉验证评分

案例 5-2 的文件名是 kb501_cross_val_score.py，本案例介绍交叉验证评分函数 cross_val_score 的使用方法。

本案例与案例 5-1 基本相同，不同之处在于第 6 组代码，代码如下。

```
    #6
    print('#6,交叉验证')
    c10 = cross_validation.cross_val_score(mx,x,y, cv=2)
    print('cross_val_score: ',c10)
    print('mean: ',np.mean(c10))
```

对应的输出信息如下：

```
    #6,交叉验证
    cross_val_score:  [0.68804092 0.50176344]
    mean:  0.5949021788403082
```

为了简化案例，在函数调用时 cv 参数值为 2，表示两次训练轮回。

函数返回值保存在变量 c10 当中，是 list 列表格式。

奇怪的是，在 cross_val_score 函数的返回值当中可以有负数。当案例中 cv 参数为 10 时，对应的输出数据如下：

```
    #6,交叉验证
    cross_val_score:  [ 0.1824246   0.33670479  0.78512653
0.04713863 -0.09994952  0.44728146   0.00533982 -1.7067912
0.75230078 -0.39694398]
    mean:  0.03526319049678991
```

c10 的返回值不能用绝对值简单取整，不然平均值会出现大于 100%的情况。

第 6 章

决策树

在通常情况下，决策树在介绍完机器学习的基本模型后再进行讲解。

本书把决策树放在前面讲解，是因为目前机器学习、神经网络平台已经非常成熟，各自经典的 AI 算法模型都有专门的函数。对于程序员而言，采用决策树模型或者最简单的线性模型，只是程序名称和调用参数的不同。

此外，决策树的模型已经涉及深度学习、神经网络，尽早接触决策树对于日后学习神经网络也有所帮助。决策树的模型也是少数可以可视化分析机器学习的模型，易于初学者理解。

为了结合量化交易的实际情况，本书内容的设计做了以下优化。

- 本节案例采用的数据全部是上证指数的数据 。
- 对比曲线当中的字段 y 是实盘当中第二天的收盘价。
- 字段 y2 是模型生成的第二天收盘价的预测数据。
- 对于分类模型，采用的是上涨、下跌、平稳三种模式。

6.1 决策树算法

决策树算法（Decision Tree Algorithm），简称 DT 算法。

决策树算法是一种无监督的学习方法，用于分类和回归。它对数据中蕴含的决策规则进行建模，以预测目标变量的值。

决策树算法最早产生于 20 世纪 60 年代，是一种逼近离散函数值的方法，本质上决策树是通过一系列规则对数据进行分类的过程。

在进行逐步应答过程中，典型的决策树分析会使用分层变量或决策节点。例如，可将一个给定用户分类成信用可靠或不可靠。

决策树模型中的目标是可变的，可以是一组有限值，被称为分类树。在这些树结构中，叶子表示类标签，分支表示表征这些类标签的连接特征，如图 6.1 所示。

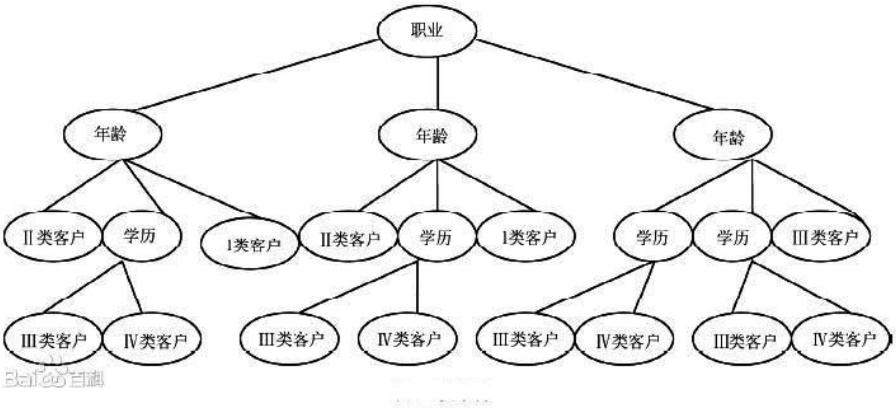


图 6.1　决策树算法

决策树算法的优点如下。

- 模型便于可视化。
- 无须对数据进行预处理。
- 擅长对人、地点、事物的一系列不同特征、品质、特性进行评估。

决策树算法的缺点如下。

- 容易出现过拟合，虽然创建了一个决策树，却过于复杂无法应用在工程实践当中。

- 决策树模型不稳定，即使很小的变异，也会产生完全不同的模型。
- 可能陷于局部最小值中。
- 没有在线学习。

传统决策树算法往往基于启发式算法对样本和特征进行随机抽样，例如贪婪算法，即每个节点创建最优决策。这样一来，虽然该算法不能产生一个全局最优的决策树，却可以降低整体效果偏差。

决策树模型的表现形式为二叉树，是来自算法和数据结构方面的二叉树。树上每个节点代表一个输入变量（x）与一个基于该变量的分离点（假定这个变量是数字）。

决策树构造方法其实就是每次选择一个好的特征和分裂点作为当前节点的分类条件。

决策树算法同样适用于数值分析，是一种逼近离散函数值的方法。它是一种典型的分类方法，首先对数据进行处理，利用归纳算法生成可读的规则和决策树，然后使用决策对新数据进行分析。

决策树算法学习起来很快，预测速度也很快。决策树对于各种各样的问题都能做出准确的预测，并且无须对数据做任何特殊的预处理。

从商业决策的角度来看，决策树就是通过尽可能少的干扰来判断问题，预测决策正确的概率，从而用一种结构性的、系统性的方法来得出合理的结论。

6.1.1　ID3 算法与 C4.5 算法

决策树算法的核心是 ID3 的改进算法 C4.5 算法。

1986 年，人工智能专家 J.Ross Quinlan 提出了著名的 ID3 算法，可以减少树的深度，加快运行速度。

ID3 算法的目的在于减少树的深度，但是忽略了对叶子数目的研究。

C4.5 算法在 ID3 算法的基础上进行了改进，在预测变量的缺值处理、剪枝技术、派生规则等方面有了较大改进，既适合于分类问题，又适合于回归问题。

6.1.2 常用决策树算法

常用的决策树算法如下。

- ID3（Iterative Dichotomiser 3）算法：采用熵参数来度量信息的不确定度，选择“信息增益”最大作为节点特征。它是多叉树，即一个节点可以有多个分支。
- C4.5 和 C5.0 算法：是 ID3 算法的改进版本，同样采用熵参数来度量信息的不确定度，选择“信息增益比”最大作为节点特征。同样是多叉树，即一个节点可以有多个分支。
- CART（Classification and Regression Tree）算法：分类和回归树算法。采用基尼指数来度量信息的不纯度，选择基尼指数最小作为节点特征，它是二叉树。
- RF（随机森林）算法：通过使用多个带有随机选取的数据子集的树来改善决策树的精确性。
- CHi-squared Automatic Interaction Detection，CHAID 算法。
- Decision Stump 算法。
- MARS（多元自适应回归样条）算法。
- GBM（Gradient Boosting Machine 梯度推进机）算法。
- PUBLIC 算法。
- SLIQ 算法。
- SPRINT 算法。
- M5 算法。

- Conditional Decision Tree 算法。

这些算法的主要区别在于度量信息的方法、选择节点特征和分支数量的不同。

6.1.3　sklearn 内置决策树算法

在 sklearn 模块库中，与 DecisionTreeClassifier 相关的算法函数位于 Tree 模块中，其中相关的机器学习算法函数如下。

- DecisionTreeClassifier：决策树分类算法，如图 6.2 所示。
- DecisionTreeRegressor：决策树回归算法，如图 6.3 所示。
- ExtraTreesRegressor：极端随机树回归算法。
- ExtraTreesClassifier：极端随机树分类算法。
- export_graphviz：辅助函数，输出决策树图形，如图 6.4 所示。

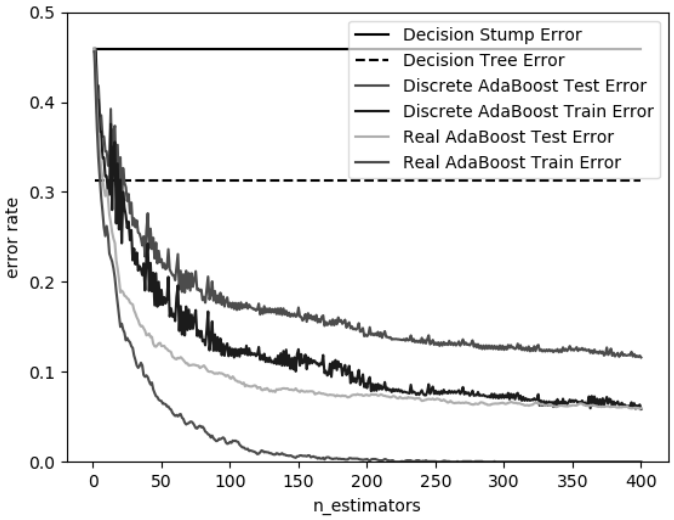


图 6.2　决策树算法

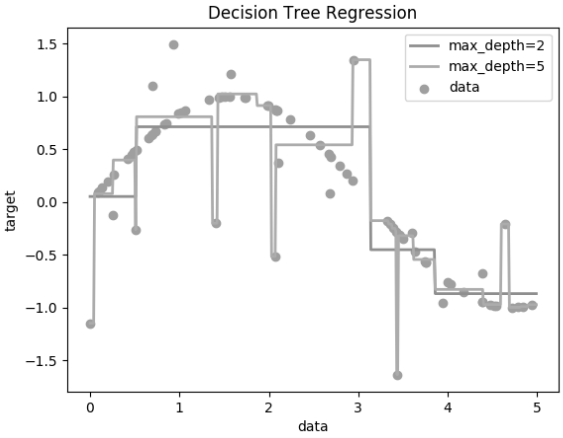


图 6.3　决策树回归算法

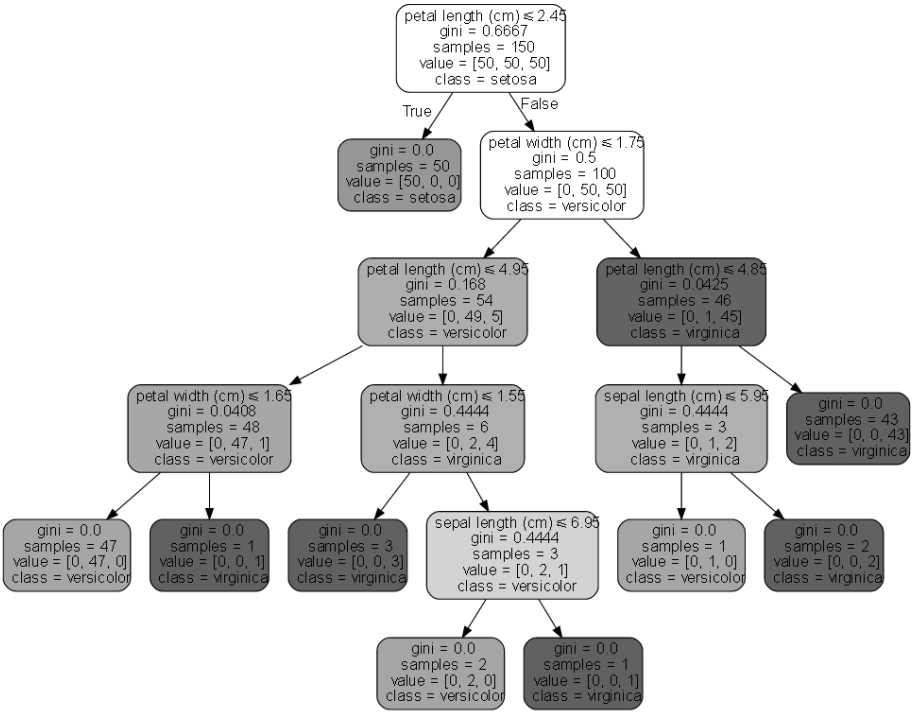


图 6.4　决策树图形

在神经网络、深度学习流行以前，最流行的是 SVM（支持向量机）算法，它也是一个有监督的集成决策树算法模型。

SVM（支持向量机）算法的全称是 Support Vector Machine，通常用来进行模式识别、分类和回归分析，支持向量机算法的提出，有很深的理论背景。

sklearn 模块库的内置决策树函数有以下两个重要函数：

- DecisionTreeClassifier，决策树分类函数。
- DecisionTreeRegressor，决策树回归函数。

这两个函数使用的算法都是 CART 算法，也就是分类与回归树算法，划分标准默认使用基尼指数。在稍后的案例当中，我们可以在决策树可视化图片当中看到基尼指数的数值。

sklearn 模块库当中的决策树算法，其内部实现使用优化过的 CART 树算法，既可以做分类，也可以做回归。

6.2　决策树回归函数

量化交易真正需要的是基于人工智能的数值计算模型，这也是为什么在金融工程领域，人工智能始终处在艰难摸索阶段的一个重要原因。

sklearn 模块库当中的决策树回归函数是少有的可用于数值计算的成熟的机器学习模型。

DecisionTreeRegressor 函数的调用接口如下：

```
    DecisionTreeRegressor(criterion='mse', splitter='best',
max_depth = None, min_samples_split=2, min_samples_leaf=1,
max_features = None, min_weight_fraction_leaf = 0.0,
random_state=None, max_leaf_nodes=None, min_impurity_decrease=0.0,
min_impurity_split=None, presort=False)
```

以上接口参数看起来很多，其实常用的一般只有 max_depth 参数，即决策数的深度。

函数常用的参数如下。

- max_depth：决策树的最大深度，当树的深度到达 max_depth 的时候，无论还有多少可以分支的特征，决策树都会停止运算。
- min_samples_split：分裂所需的最小数量的节点数，当叶节点的样本数量小于该参数后，则不再生成分支。
- min_samples_leaf：一个分支所需要的最少样本数，如果在分支之后，某一个新增叶节点的特征样本数小于该参数，则退回，不再进行剪枝。退回后的叶节点的标签以该叶节点中最多的标签为准。
- min_weight_fraction_leaf：最小的权重系数。
- max_leaf_nodes：最大叶节点数，None 时无限制，取整数时忽略 max_depth。

案例 6-1：决策树回归算法

案例 6-1 的文件名是 kb601_treeReg.py，本案例主要介绍 sklearn 模块库当中决策树回归函数的具体应用和决策树的可视化编程。

前面 4 组代码属于数据准备，代码如下：

```
#1
print('#1,rd data')
fss='data/sh2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
zt.prDF('xdf',xdf)

#2
print('#2,xed data')
xdf['xprice']=xdf['close'].shift(-1)
xdf['y']=xdf['xprice']
xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf',xdf)
```

```
#3.1
print('#3,cut data')
tim0str,tim9str='2010-10-10','2017-12-31'
df_train=zdat.df_kcut8tim(xdf,'',tim0str,tim9str)
zt.prDF('#3.1 df_train\n',df_train)
#
#3.2
df_test=xdf[xdf.index>'2018'] #.tail(100)
zt.prDF('#3.2 df_test\n',df_test)



#4
print('#4,准备 AI 数据')
clst=['open','high','low','close']
x=df_train[clst].values
y=df_train['y'].values
#
xtst=df_test[clst].values
```

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = tree.DecisionTreeRegressor(max_depth=3)
```

调用决策树回归函数生成模型，并保存在变量 mx 当中，调用 max_depth 参数为 3，表示决策数的深度只有 3 层。

这个深度效果有些差，只是作为教学案例使用，是为了方便决策树模型的输出。在实盘当中，往往会使用数十层的深度。

不过决策树算法容易出现过拟合的问题，所以也很少使用类似深度学习及神经网络数百层的深度进行运算。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
```

```
    mx.fit(x,y)
```

按惯例，调用 fit 函数训练模型。

第 7 组代码如下：

```
    #7
    print('#7,输出模型')
    dot_data = StringIO()
    tree.export_graphviz(mx, out_file=dot_data,
              feature_names=clst,
              #class_names=vlst,
              filled=True, rounded=True,
              special_characters=True)
    graph = pydot.graph_from_dot_data(dot_data.getvalue())
    #graph.write_pdf('tmp/tree01.pdf')
    graph.write_png('tmp/tree01.png')
```

这组代码是以往案例当中少有的。

机器学习模型的可视化是非常有价值的，但是模型理论和编程都比较复杂，只有很少的模型能够实现可视化。所以 TensorFlow 深度学习平台的 TensorBoard 被人称为：好用的神经网络可视化神器。

以上代码的决策树模型可视化实际上是通过调用第三方绘图软件 Graphviz 完成的，所以在运行前，请大家自己先下载、安装好相关的软件，并在 Windows 环境中配置好 path 系统路径参数。

第 8 组代码如下：

```
    #8
    print('#8,predict 模型预测数据')
    df_test['y2']=mx.predict(xtst) #cross_val_predict(mx,x, y,
cv=20)
    zt.prDF('df',df_test)
    df_test[['y','y2']].plot(linewidth=3)
    #print(predicted)
```

模型训练完成后，调用模型内置的 predict 预测函数，生成验证数据，并绘制对比图。

第 9 组代码如下：

```
    #9
    print('#9,验证模型预测效果')
    #9.1
    print('\n#9.1,按 5%精度验证模型')
    dacc,df=zai.ai_acc_xed2x(df_test['y'],df_test['y2'],5,True)
    print('acc',dacc)

    #9.2
    print('\n#9.2,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['y'],df_test['y2'],1,
True)
```

验证决策树的预测结果。

表 6.1 是调用决策树回归函数时，max_depth 参数分别为不同值的运算结果。

表 6.1　运算结果

max_depth 值	准确度，按 1%精度	准确度，按 5%精度
3	9.57%	70.2%
5	28.72%	100%
30	52.13%	100%
50	51.06%	100%

验证结果表明，50 层的决策树的准确度比 30 层的决策树的准确度还低，这也是决策树模型一种过拟合的表现。

图 6.5～图 6.8 所示是对应不同 max_depth 决策树深度参数时，预测数据与实际数据的对比图。图中，曲线 y 是真实数据，曲线 y2 是决策树生成的预测数据。

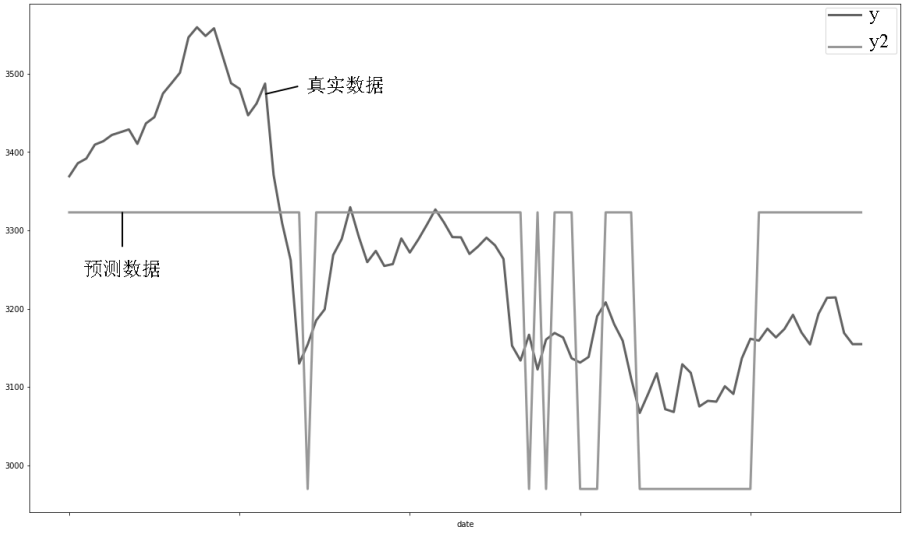


图 6.5　决策树模型对比图，决策树深度：3 层

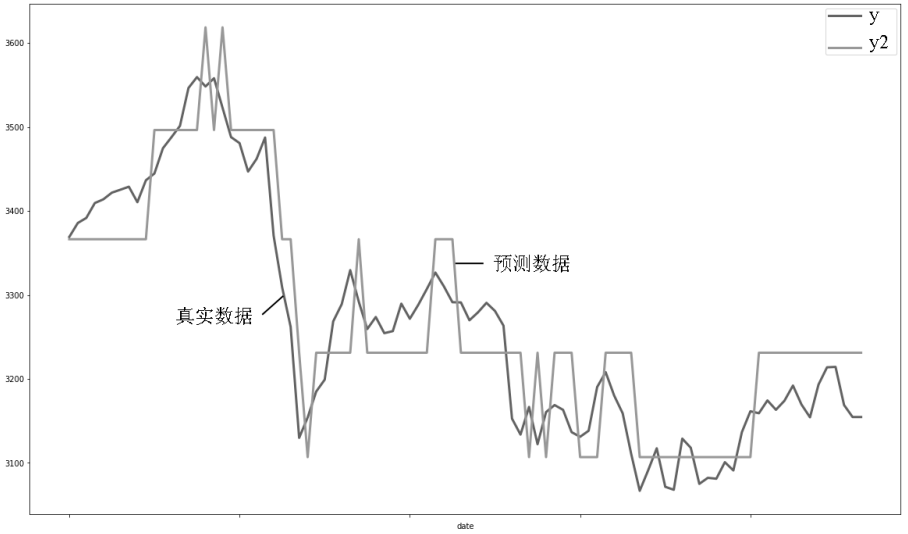


图 6.6　决策树模型对比图，决策树深度：5 层

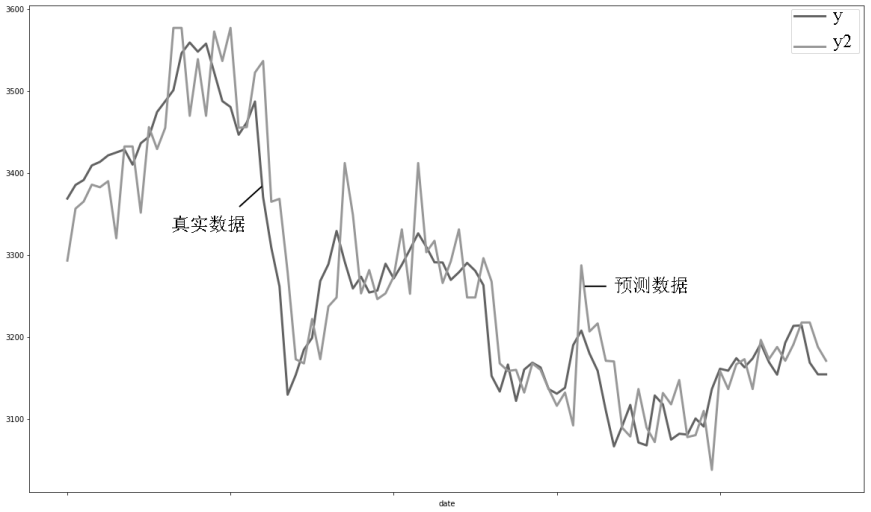


图 6.7　决策树模型对比图，决策树深度：30 层

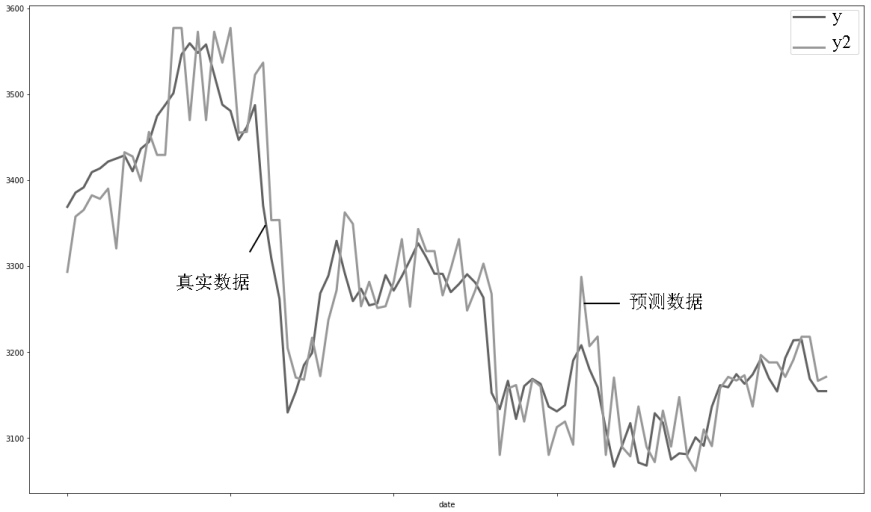


图 6.8　决策树模型对比图，决策树深度：50 层

6.3　决策树分类函数

DecisionTreeClassifier 函数位于决策树模块，函数调用接口如下：

```
DecisionTreeClassifier(criterion='gini', splitter='best',
```

```
max_depth=None, min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_features=None, random_state=None,
max_leaf_nodes=None, min_impurity_split=1e-07, class_weight=None,
presort=False)
```

以上接口参数看起来很多，其实常用的一般只有 max_depth 参数，即决策数的深度。

其他函数常用的参数与 DecisionTreeRegressor 函数的接口类似，这里不再赘述。

案例 6-2：决策树分类算法

案例 6-2 的文件名是 kb602_treeClass.py，本案例介绍的是 sklearn 模块库中决策树分类函数 DecisionTreeClassifier 的使用方法。

前面 4 组代码都是数据准备，如下所示：

```
#1
print('#1,rd data')
fss='data/sh2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
zt.prDF('xdf',xdf)

#2
print('#2,xed data')
xdf['xprice']=xdf['close'].shift(-1)
#xdf['y']=xdf['xprice']
#
xdf['kpr']=xdf['xprice']/xdf['close']*1000
xdf['y']=0
xdf.loc[xdf.kpr>1005,'y']=2
xdf.loc[xdf.kpr<995,'y']=1
xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf',xdf)
```

```
#3.1
print('#3,cut data')
tim0str,tim9str='2010-10-10','2017-12-31'
df_train=zdat.df_kcut8tim(xdf,'',tim0str,tim9str)
zt.prDF('#3.1 df_train\n',df_train)
#
#3.2
df_test=xdf[xdf.index>'2018'] #.tail(100)
zt.prDF('#3.2 df_test\n',df_test)



#4
print('#4,准备 AI 数据')
clst=['open','high','low','close']
x=df_train[clst].values
y=df_train['y'].values
#
xtst=df_test[clst].values
```

数据准备阶段，因为是分类模型，所以需要先把数据转换为分类类型，这个主要在第 2 组代码当中完成：

```
xdf['kpr']=xdf['xprice']/xdf['close']*1000
xdf['y']=0
xdf.loc[xdf.kpr>1005,'y']=2
xdf.loc[xdf.kpr<995,'y']=1
```

因为股票市场波动很小，我们按收盘价千分之五的上下波动对数据进行分类：

- 2，波动大于 100.5%，up 上涨模式。
- 1，波动小于 99.5%，down 下跌模式。
- 0，波动在 0.5%之间，eq 平稳模式。

以上分类类似足彩博弈的“3、1、0，胜、平、负”模式，稍后第 7 组代码的变量列表 vlst 会在图 6.9 当中说明：

```
vlst=['up','down','eq']
```

第 5 组代码如下：

```
#5
print('#5,模型设置')
#mx = tree.DecisionTreeRegressor(max_depth=3)
mx = tree.DecisionTreeClassifier(max_depth=3)
```

调用 DecisionTreeClassifier 函数，生成模型，并保存在变量 mx 当中，调用参数 max_depth 为 3，表示决策数的深度只有 3 层。

第 6 组代码对模型进行训练，具体如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

第 7 组代码输出模型结构，具体如下：

```
#7
print('#7,输出模型')
vlst=['up','down','eq']
dot_data = StringIO()
tree.export_graphviz(mx, out_file=dot_data,
          feature_names=clst,
          class_names=vlst,
          filled=True, rounded=True,
          special_characters=True)
graph = pydot.graph_from_dot_data(dot_data.getvalue())
#graph.write_pdf('tmp/tree01.pdf')
graph.write_png('tmp/tree02.png')
```

为了进一步说明模型内部的分类情况，我们增加了一组 vlst 分类列表变量。

图 6.9 是对应的决策树分类算法模型。

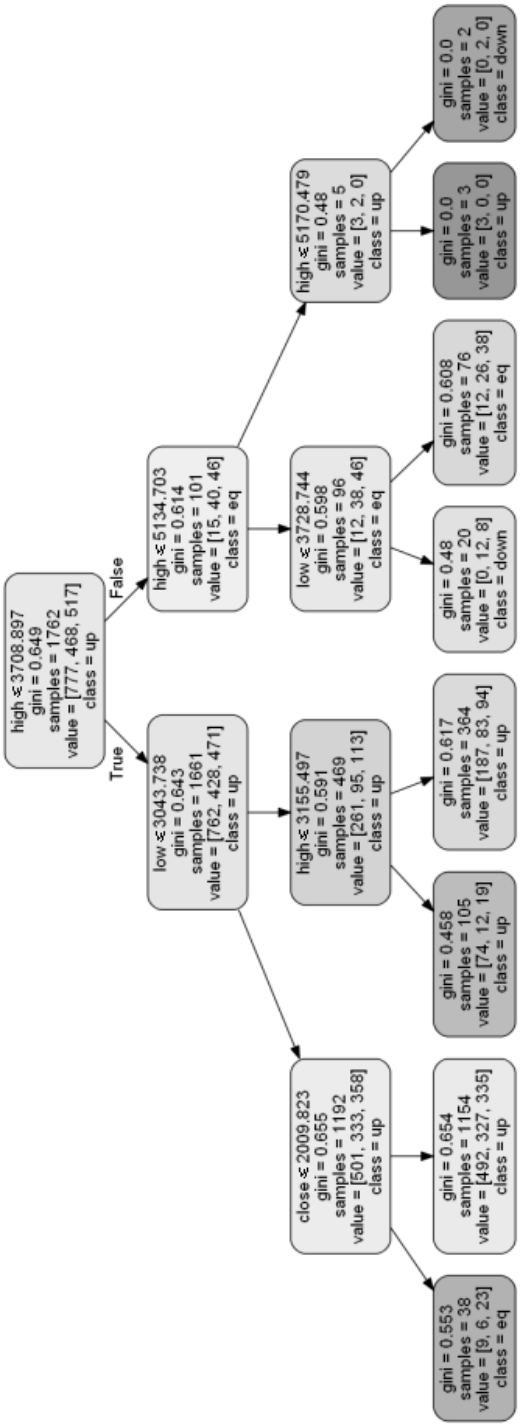


图 6.9　决策树分类算法模型

图 6.9 中每个节点最下面一行的“class=xx”就是分类的意思，对应 vlst 当中定义的类别。

此外图 6.9 模型节点当中的基尼指数和 vlst 对应的特征字段都可以用于模型的分析、优化。

第 8 组代码如下：

```
    #8
    print('#8,predict 模型预测数据')
    df_test['y2']=mx.predict(xtst) #cross_val_predict(mx,x, y,
cv=20)
    zt.prDF('df',df_test)
```

模型训练完成后，调用模型内置的 predict 预测函数，生成验证数据，并绘制对比图。

第 9 组代码如下：

```
    #9
    print('#9,验证模型预测效果')
    #9.1
    print('\n#9.1,按 5%精度验证模型')
    dacc,df=zai.ai_acc_xed2x(df_test['y'],df_test['y2'],5,True)
    print('acc',dacc)

    #9.2
    print('\n#9.2,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['y'],df_test['y2'],1,
True)

    #9.3
    print('\n#9.3,value_counts')
    print("\ndf_train['y'].value_counts()")
    print(df_train['y'].value_counts())
    #
    print("\ndf_test['y'].value_counts()")
```

```
print(df_test['y'].value_counts())
print("\ndf_test['y2'].value_counts()")
print(df_test['y2'].value_counts())
```

验证决策树的预测结果。

表 6.2 是调用 DecisionTreeRegressor 函数时，max_depth 参数分别为不同值的运算结果。

表 6.2　运算结果

max_depth 值	准确度，按 1%精度	准确度，按 5%精度
3	41.49%	41.489%
5	41.49%	41.489%
10	35.11%	35.106%
30	38.30%	38.298%

此外，程序还输出了不同决策树层数 max_depth，对应在预测数据字段 y2 的结果中，以及在不同分类数据的 value_counts 统计结果中。

6.4　GBDT算法

GBDT 算法英文全称是 Gradient Boosting Decision Tree，简称 GBDT 算法。

GBDT 算法是一种集成模型，也是一个集成学习算法。简单来说，GBDT 算法是指用许多交叉决策树算法组成了一个更强大的学习算法。

GBDT 算法由多棵决策树组成，将所有树的结论累加起来作为最终结果。它在被提出之初就和 SVM 一起被认为是泛化能力较强的算法。近几年更因为被用于搜索排序的机器学习模型而引起大家关注，如图 6.10 所示。

GBDT 算法是一个应用很广泛的算法，可以用来做分类和回归，在很多的数据上都有不错的效果。GBDT 这个算法还有一些其他的名字，比如 MART（Multiple Additive Regression Tree）、GBRT（Gradient Boosting Regression Tree）、Tree Net 等。

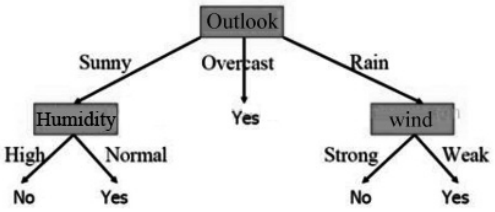


图 6.10　GBDT 算法

6.5　迭代决策树函数

sklearn 模块内置的迭代决策树函数包括两个：

- GradientBoostingRegressor，迭代回归决策树函数。
- GradientBoostingClassifier，迭代分类决策树函数。

两个函数都位于 Ensemble（集成）算法模块，这也说明 GBDT 算法是一个集成学习算法。

迭代回归决策树函数的调用接口如下：

```
    GradientBoostingRegressor(loss='ls', learning_rate=0.1,
n_estimators=100, subsample=1.0, criterion='friedman_mse',
min_samples_split=2, min_samples_leaf=1, max_depth=3,
min_weight_fraction_leaf=0.0, min_impurity_decrease=0.0,
min_impurity_split=None, init=None, random_state=None,
max_features=None, alpha=0.9, verbose=0, max_leaf_nodes=None,
warm_start=False, presort='auto')
```

迭代分类决策树函数的调用接口如下：

```
    GradientBoostingClassifier(loss='deviance', learning_rate=0.1,
n_estimators=100, subsample=1.0, criterion='friedman_mse',
min_samples_split=2, min_samples_leaf=1, max_depth=3,
min_weight_fraction_leaf=0.0, min_impurity_decrease=0.0,
min_impurity_split=None, init=None, random_state=None,
max_features=None, verbose=0, max_leaf_nodes=None,
warm_start=False, presort='auto')
```

两个函数的调用接口基本相同，主要参数如下。

- n_estimators：梯度提升的迭代次数，也是弱分类器的个数，默认值是 100 次。
- loss：损失函数，包括“ls”“lad”“huber”“quantile”，默认值是“ls”。
- learning_rate：学习步长，默认值是 0.1。
- max_depth：决策树最大深度，默认值是 3。
- warm_start：如果是 True，会存储之前的拟合结果以增加迭代次数，默认值是 False。

由以上参数可以看出，GBDT 算法已经包括了神经网络的不少特点，参数 learning_rate、loss 与神经网络模型中完全相同，如果结合 BP 反馈模式调用，就是一个简单的神经网络模型。

案例 6-3：GBDT 回归算法

案例 6-3 的文件名是 kb603_GBDT_reg.py，本案例介绍的是 GBDT 回归函数的应用，核心代码如下。

前面 4 组代码属于数据准备，代码如下所示：

```
#1
print('#1,rd data')
fss='data/sh2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
zt.prDF('xdf',xdf)

#2
print('#2,xed data')
xdf['xprice']=xdf['close'].shift(-1)
xdf['y']=xdf['xprice']
xdf.fillna(method='pad',inplace=True)
```

```
    zt.prDF('xdf',xdf)



    #3.1
    print('#3,cut data')
    tim0str,tim9str='2010-10-10','2017-12-31'
    df_train=zdat.df_kcut8tim(xdf,'',tim0str,tim9str)
    zt.prDF('#3.1 df_train\n',df_train)
    #
    #3.2
    df_test=xdf[xdf.index>'2018'] #.tail(100)
    zt.prDF('#3.2 df_test\n',df_test)



    #4
    print('#4,准备AI数据')
    clst=['open','high','low','close']
    x=df_train[clst].values
    y=df_train['y'].values
    #
    x = x.astype(np.float32)
    #
    xtst=df_test[clst].values
```

数据源采用的是上证指数实盘数据，为便于大家以后结合实盘操作，所以导入数据、训练数据字段就是标准的 OHLC 金融数据。

以下是对应的输出信息：

```
     #3.2 df_test

                  open    high    close     low      volume
amount   xprice     kpr  y
    date
    2018-01-02  3314.03  3349.05  3348.33  3314.03  20227886000
227788461113  3369.11  1006.21  2
    2018-01-03  3347.74  3379.92  3369.11  3345.29  21383614900
```

```
258366523235  3385.71  1004.93  0
    2018-01-04  3371.00  3392.83  3385.71  3365.30  20695528800
243090768694  3391.75  1001.78  0
    2018-01-05  3386.46  3402.07  3391.75  3380.24  21306068100
248187840542  3409.48  1005.23  2
    2018-01-08  3391.55  3412.73  3409.48  3384.56  23616510600
286213219095  3413.90  1001.30  0
    2018-01-09  3406.11  3417.23  3413.90  3403.59  19148855100
238249975070  3421.83  1002.32  0
    2018-01-10  3414.11  3430.21  3421.83  3398.84  20909499700
254515441261  3425.34  1001.03  0
    2018-01-11  3415.58  3426.48  3425.34  3405.64  17381213300
218414134129  3428.94  1001.05  0
    2018-01-12  3423.88  3435.42  3428.94  3417.98  17406340400
215961455748  3410.49   994.62  1
    2018-01-15  3428.95  3442.50  3410.49  3402.31  23200928300
286362732919  3436.59  1007.65  2

                open    high    close     low      volume
amount   xprice     kpr  y
    date
    2018-05-11  3179.80  3180.76  3163.26  3162.21  13065974900
167364290366  3174.03  1003.40  0
    2018-05-14  3167.04  3183.82  3174.03  3163.48  12932735300
172410691054  3192.12  1005.70  2
    2018-05-15  3180.42  3192.81  3192.12  3164.52  12454905100
162990790010  3169.56   992.93  1
    2018-05-16  3180.23  3191.95  3169.56  3166.81  13052496800
174590979834  3154.28   995.18  0
    2018-05-17  3170.01  3172.77  3154.28  3148.62  11399556700
150598842185  3193.30  1012.37  2
    2018-05-18  3151.08  3193.45  3193.30  3144.78  13651691800
168038057477  3213.84  1006.43  2
    2018-05-21  3206.18  3219.74  3213.84  3203.34  16445941300
202663464515  3214.35  1000.16  0
    2018-05-22  3211.25  3214.59  3214.35  3192.23  14429268400
```

```
185721667752  3168.96   985.88  1
    2018-05-23  3205.44  3205.44  3168.96  3168.96  15780764800
199358101015  3154.65   995.48  0
    2018-05-24  3167.94  3173.53  3154.65  3152.07  12408580000
160658185502  3154.65   995.48  0
```

第 5 组代码如下：

```
    #5
    print('#5,模型设置')
    mx = ensemble.GradientBoostingRegressor(n_estimators=500,
max_depth=3)
```

调用 GradientBoostingRegressor 函数生成模型，并保存在变量 mx 当中，调用参数如下。

- n_estimators=100，梯度提升的迭代次数，也是弱分类器的个数。
- max_depth=3，表示决策树的深度只有 3 层，这个也是函数的默认值。

由参数 n_estimators=100 可以看出，GBDT 算法虽然对于决策树的要求不高，但对于决策树的种类数量却要求很多，默认值就是 100。

第 6 组代码如下：

```
    #6
    print('#6,fit 训练模型')
    mx.fit(x,y)
```

按惯例，调用 fit 函数训练模型。

第 7 组代码如下：

```
    #7
    '''
    print('#7,输出模型')
    dot_data = StringIO()
    tree.export_graphviz(mx, out_file=dot_data,
              #feature_names=clst,
              #class_names=vlst,
              filled=True,
              rounded=True,
```

```
                special_characters=True
                )
    graph = pydot.graph_from_dot_data(dot_data.getvalue())
    #graph.write_pdf('tmp/tree01.pdf')
    graph.write_png('tmp/tree01.png')
    '''
```

虽然是决策树算法，但是由于是多种决策树的集成模式，所以无法绘制算法内部的模型结构。

第 8 组代码如下：

```
    #8
    print('#8,predict 模型预测数据')
    df_test['y2']=mx.predict(xtst) #cross_val_predict(mx,x, y,
cv=20)
    zt.prDF('df',df_test)
    df_test[['y','y2']].plot(linewidth=3)
    #print(predicted)
```

模型训练完成后，调用模型内置的 predict 预测函数，生成验证数据。

第 9 组代码如下：

```
    #9
    print('#9,验证模型预测效果')
    #9.1
    print('\n#9.1,按 5%精度验证模型')
    dacc,df=zai.ai_acc_xed2x(df_test['y'],df_test['y2'],5,True)
    print('acc',dacc)

    #9.2
    print('\n#9.2,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['y'],df_test['y2'],1,
True)
```

验证决策树的预测结果。

表 6.3 是调用 GradientBoostingRegressor 函数时，n_estimators 参数分别

为不同值的运算结果。

表 6.3　运算结果

n_estimators 值	准确度，按 1%精度	准确度，按 5%精度
10	0%	41.489%
50	100%	68.09%
100	100%	64.89%
300	100%	63.83%
500	100%	62.77%

需要注意的是，在以上测试数据中，当 n_estimators=50 时，准确度比 n_estimators 值是 100、300、500 时的决策树集成模型还高。这也可能是决策树模型过拟合的一种表现。

案例 6-4：GBDT 分类算法

案例 6-4 的文件名是 kb604_GBDT_cla.py，本案例介绍的是 GBDT 分类函数 GradientBoostingClassifier 的应用，核心代码如下。

数据源采用的是上证指数实盘数据，为便于大家以后结合实盘操作，所以导入数据、训练数据字段就是标准的 OHLC 金融数据。

前面 4 组代码，属于数据准备，代码如下：

```
#1
print('#1,rd data')
fss='data/sh2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
zt.prDF('xdf',xdf)

#2
print('#2,xed data')
xdf['xprice']=xdf['close'].shift(-1)
#xdf['y']=xdf['xprice']
```

```
#
xdf['kpr']=xdf['xprice']/xdf['close']*1000
xdf['y']=0
xdf.loc[xdf.kpr>1005,'y']=2
xdf.loc[xdf.kpr<995,'y']=1
xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf',xdf)

#3.1
print('#3,cut data')
tim0str,tim9str='2010-10-10','2017-12-31'
df_train=zdat.df_kcut8tim(xdf,'',tim0str,tim9str)
zt.prDF('#3.1 df_train\n',df_train)
#
#3.2
df_test=xdf[xdf.index>'2018'] #.tail(100)
zt.prDF('#3.2 df_test\n',df_test)



#4
print('#4,准备 AI 数据')

clst=['open','high','low','close']
x=df_train[clst].values
y=df_train['y'].values
#
xtst=df_test[clst].values
```

在数据准备阶段，因为是分类模型，所以需要先把数据转换为分类类型，这个主要在第 2 组代码当中完成：

```
xdf['kpr']=xdf['xprice']/xdf['close']*1000
xdf['y']=0
xdf.loc[xdf.kpr>1005,'y']=2
xdf.loc[xdf.kpr<995,'y']=1
```

因为股票市场波动很小，所以我们按收盘价千分之五的上下波动对数

据进行分类，如下所示：

- 2，波动大于 100.5%，up 上涨模式。
- 1，波动小于 99.5%，down 下跌模式。
- 0，波动在 0.5%之间，eq 平稳模式。

以上分类类似足彩博弈的“3、1、0，胜、平、负”模式。

第 5 组代码如下：

```
    #5
    print('#5,模型设置')
    #mx = tree.DecisionTreeRegressor(max_depth=50)
    mx = ensemble.GradientBoostingClassifier ( n_estimators = 100 ,
max_depth=3)
```

调用 GradientBoostingClassifier 函数生成模型，并保存在变量 mx 当中，调用参数如下。

- n_estimators=100，梯度提升的迭代次数，也是弱分类器的个数。
- max_depth=3，表示决策树的深度只有 3 层，这也是函数的默认值。

由参数 n_estimators=100 可以看出，GBDT 算法虽然对于决策树的要求不高，但是对于决策树的种类数量要求很多，默认值就是 100。

第 6 组代码如下：

```
    #6
    print('#6,fit 训练模型')
    mx.fit(x,y)
```

第 7 组代码如下：

```
    '''
    #7
    print('#7,输出模型')
    vlst=['up','down','eq']
    dot_data = StringIO()
    tree.export_graphviz(mx, out_file=dot_data,
                         feature_names=clst,
```

```
                class_names=vlst,
                filled=True, rounded=True,
                special_characters=True)
    graph = pydot.graph_from_dot_data(dot_data.getvalue())
    #graph.write_pdf('tmp/tree01.pdf')
    graph.write_png('tmp/tree02.png')
    '''
```

虽然是决策树算法，但是由于是多种决策树的集成模式，所以无法绘制算法内部的模型结构。

第 8 组代码如下：

```
    #8
    print('#8,predict 模型预测数据')
    df_test['y2']=mx.predict(xtst) #cross_val_predict(mx,x, y,
cv=20)
    zt.prDF('df',df_test)
```

模型训练完成后，调用模型内置的 predict 预测函数，生成验证数据。

第 9 组代码如下：

```
    #9
    print('#9,验证模型预测效果')
    #9.1
    print('\n#9.1,按 5%精度验证模型')
    dacc,df=zai.ai_acc_xed2x(df_test['y'],df_test['y2'],5,True)
    print('acc',dacc)

    #9.2
    print('\n#9.2,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['y'],df_test['y2'],1,
True)

    #9.3
    print('\n#9.3,value_counts')
    print("\ndf_train['y'].value_counts()")
    print(df_train['y'].value_counts())
```

```
#
print("\ndf_test['y'].value_counts()")
print(df_test['y'].value_counts())
print("\ndf_test['y2'].value_counts()")
print(df_test['y2'].value_counts())
```

验证决策树的预测结果。

表 6.4 是调用 GradientBoostingClassifier 函数时，n_estimators 参数分别为不同值的运算结果。

表 6.4　运算结果

n_estimators 值	准确度，按 1%精度	准确度，按 5%精度
10	40.43%	40.426%
50	40.43%	40.426%
100	42.55%	42.553%
300	44.68%	44.681%
500	41.49%	41.489%

需要注意的是，在以上测试数据中，当 n_estimators=300 时，准确度比 n_estimators 值是 500 时的决策树集成模型还高，这也可能是决策树模型过拟合的一种表现。

第 7 章

随机森林算法和极端随机树算法

决策树算法还有许多衍生版本，常见的有 GBDT 算法、RF 算法等。

在 sklearn 模块库中的 Ensemble（集成）算法的子模块当中，收录了以下几种决策树衍生算法的调用函数。

- GradientBoostingRegressor：GBDT 回归算法。
- GradientBoostingClassifier：GBDT 分类算法。
- RandomForestRegressor：随机森林回归算法。
- RandomForestClassifier：随机森林分类算法。
- ExtraTreesRegressor：极端随机树回归算法。
- ExtraTreesClassifier：极端随机树分类算法。

本章介绍的随机森林算法、极端随机树算法，也算是决策树算法的一种衍生版本。

随机森林（Random Forest）算法，简称 RF 算法，是最流行、最强大的机器学习算法之一，也是一种机器学习集成算法。该算法的核心思路是利用多棵树对样本进行训练并预测，RF 算法提高了决策树的精确性，如

图 7.1 所示。

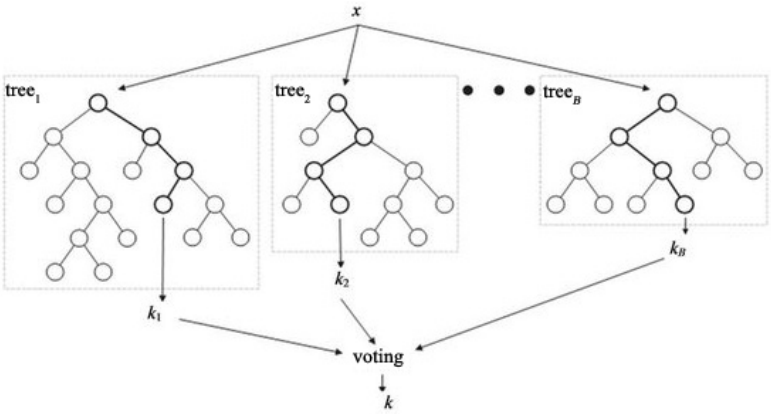


图 7.1　RF 随机森林算法

随机森林这个术语是 1995 年由贝尔实验室的 Tin Kam Ho 所提出的随机决策森林（Random Decision Forests）理论发展而来的。后来，Leo Breiman 和 Adele Cutler 推论出随机森林的算法，并注册了 Random Forests 商标。

随机森林算法是一个包含多个决策树的算法，并且其输出的类别是由个别树输出的类别的众数而定的，这个算法结合了 Leo Breimans 的“Bootstrap Aggregating”想法和 Tin Kam Ho 的“Random Subspace Method”来建造决策树的集合。

随机森林算法是 DT（决策树）算法的一种衍生算法，通过对 Bagging 算法的调整，引入随机性来得到次优分割点，而不是选择最佳分割点来创建决策树。

它针对每个数据样本所创建的模型与其他模型有所不同，但仍能以其独特的方式准确预测，并且可以更好地估计潜在的真实输出。

如图 7.2 所示，是一个随机森林算法案例，在基因表达方面，在考察了大量与乳腺癌复发相关的基因后，通过随机森林算法计算出复发风险。

在很多领域，随机森林算法都是最好的工程解决方案，遗憾的是，随机森林算法背后的理论至今依然是一个黑箱，目前依然没有系统、科学的

理论依据。

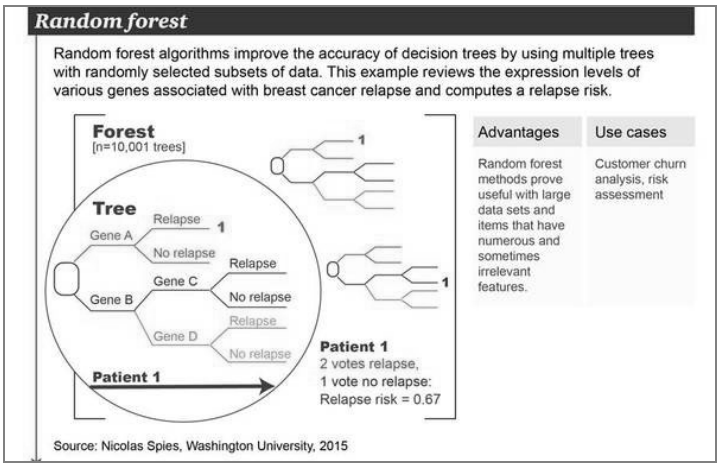


图 7.2　RF（随机森林）算法案例

随机森林算法中有很多不同的分类树，每个分类树都可以投票来对物体进行分类，从而选出票数最多的类别。

随机森林算法非常类似 GBDT（迭代决策树）算法，但区别在于它没有迭代，并且森林里树的长度不受限制。

因为随机森林算法没有迭代过程，不像 GBDT 算法需要迭代，不断更新每个样本以及子分类器的权重，因此模型相对简单，不容易出现过拟合，可以随便增加树的数量，而且执行的速度也是相对较快的。此外，随机森林方法对大规模数据集和存在大量且不相关特征项的项目的相关性非常好。

7.1　随机森林函数

在 sklearn 模块库当中，与随机森林算法相关的函数位于集成算法模块 Ensemble 当中，其中相关的机器学习算法函数如下。

- RandomForestClassifier：随机森林分类算法。
- BaggingClassifier：装袋分类算法，相当于多个专家投票表决，对于多

次测试，每个样本返回的是多次预测结果较多的那个。

- ExtraTreesClassifier：极端随机树分类算法。
- Adaboost 迭代算法：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）。
- GradientBoostingClassifier：梯度分类算法。
- GradientBoostingRegressor：梯度回归算法。
- VotingClassifier：投票分类算法。

随机森林回归函数 RandomForestRegressor 的接口定义如下：

```
    RandomForestRegressor(n_estimators=10, criterion='mse',
max_depth = None, min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf = 0.0, max_features='auto',
max_leaf_nodes=None, min_impurity_decrease = 0.0,
min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs =
1, random_state=None, verbose=0, warm_start=False)
```

随机森林分类函数 RandomForestClassifier 的接口定义如下：

```
    RandomForestClassifier(n_estimators=10, criterion='gini',
max_depth=None, min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_features='auto',
max_leaf_nodes=None, min_impurity_decrease=0.0,
min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=1,
random_state=None, verbose=0, warm_start=False, class_weight=None)
```

以上两个函数接口，参数定义都差不多，常用的参数如下。

- n_estimators：梯度提升的迭代次数，也是弱分类器的个数，默认值是 100。
- max_depth：决策树最大深度，默认值是 3。
- max_features：单个决策树使用的最大特征数量。

在以上参数当中，max_features 有多个选项。

- Auto/None：简单地选取所有特征，每棵树都可以利用。在这种情况下，

每棵树都没有任何的限制。

- sqrt：此选项是每棵子树可以取总特征数的平方根的个数使用。例如，如果变量（特征）的总数是 100，则每棵子树只能取其中的 10 个使用。
- log2：是另一种相似类型的选项。
- 0.2：此选项允许每个随机森林的子树可以利用变量（特征）数的 20%。如果想考察部分特征的作用，可以使用“0.X”的格式。

增加 max_features 的值，一般能提高模型的性能，因为在每个节点上，有更多的选择可以考虑。

不过它降低了单棵树的多样性，而这正是随机森林算法独特的优点，同时也会降低模型的运算量。

7.2　决策树测试框架

由于 sklearn 模块在设计时强调了函数 API 调用接口的统一性，因此，基本上模型的训练都是调用 fit 函数，模型的预测都是调用 predict 函数。

而模型的预测和准确度也可以由 sklearn 内置的函数或者 TOP 极宽自定义的函数进行统一评估。这样一来，就为机器学习算法、模型的统一测试奠定了基础。

我们首先设计一个基于决策树模板的机器学习测试框架，在数据源方面，为了结合金融量化实际情况，做了以下优化。

- 本节案例采用的数据全部是金融实盘数据，如上证指数、BTC（比特币）交易数据等。
- 对比曲线当中的字段 y，就是实盘当中第二天的收盘价。
- 字段 y2 是模型生成的第二天收盘价的预测数据。
- 对于分类模型，采用的是上涨、下跌、平稳三种模式。

案例 7-1：RF 回归算法大测试

案例 7-1 的文件名是 kb701_RF_reg.py，本案例介绍的是决策树测试框架，案例具体测试算法使用的是 RandomForestRegressor 函数。

由于对有关的测试函数进行了集成，所以代码非常简单。

前面三组代码是数据准备，案例采用上证指数作为数据源，在以后进行实盘金融量化分析时，可作为参考案例。

```
#1
print('#1,rd data')
fss='data/sh2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
zt.prDF('xdf',xdf)

#2
print('#2,xed data')
xdf['close_next']=xdf['close'].shift(-1)
xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf',xdf)

#3.1
print('#3,cut data')
tim0str,tim9str='2010-10-10','2017-12-31'
df_train=zdat.df_kcut8tim(xdf,'',tim0str,tim9str)
zt.prDF('#3.1 df_train\n',df_train)
#
#3.2
df_test=xdf[xdf.index>'2018'] #.tail(100)
zt.prDF('#3.2 df_test\n',df_test)
```

以上代码当中，需要注意的是第 2 组代码中的如下行：

```
xdf['close_next']=xdf['close'].shift(-1)
```

使用 close_next 作为实盘第二天收盘价的字段名称，代替了在以往机

器学习当中实盘数据的习惯性变量名称，更加贴近金融量化实盘，也便于随后的图表对比分析。

整个案例最核心的部分在第 4 组，函数测试如下：

```
    #4,
    print('#4,模型测试')
    mfun=ensemble.RandomForestRegressor
    zai.mxtst_DTree010(mfun,df_train,df_test,ntree=100,ndepth=3,f
tg='tmp/pic001.png')
```

调用 TOP 极宽模块库的 mxtst_DTree010 函数，对模型函数进行训练、预测，以及数据验证，返回值只有一个 Dacc Degree of Accuracy，准确度。

其中代码如下：

```
    mfun=ensemble.RandomForestRegressor
```

用于定义 sklearn 内置的各种机器学习的模型。

以下是程序运行的输出信息：

```
    按 1%精度验证模型
    acc: 23.40%;
```

按 1%精度验证模型，案例的准确度为 23.4%。

图 7.3 是对应的收盘价实盘数据和预测数据的对比曲线图。

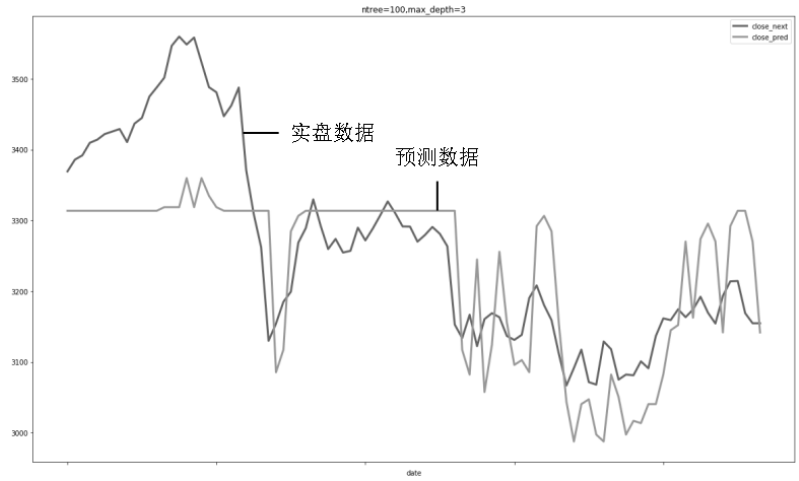


图 7.3　案例收盘价实盘数据和预测数据的对比曲线图

由图 7.3 可以看出，函数对图形进行了优化，曲线图的图标名称不再是简单抽象的 y、y2 变量名称了，而是具体的 close_next 实盘次日收盘价、close_pred 实盘预测收盘价。此外，图中标题部分，也标注了决策树模型最重要的两个参数 n_estimators（决策树数目）和 max_depth（决策树最大深度）。

7.3 决策树测试函数

在前面案例当中，核心部分全部调用 TOP 极宽模块库的决策树测试函数 mxtst_DTree010 完成。

该函数对模型函数进行训练、预测，以及数据验证，返回值只有一个 DAcc。

函数接口如下：

```
    mxtst_DTree010(mfun,df_train,df_test,ntree=100,ndepth=3,ftg='
tmp/pic001.png',xsgnlst=['open','high','low','close'],ysgnlst=['c
lose_next','close_pred'])
```

主要参数如下。

- mfun：决策树模型定义函数。
- df_train：训练数据集，字段名称由 xsgnlst 决定，一般为标准的 OHLC 数据。包括结果数据字段，默认名为 close_next，不是传统的变量名称 y。
- df_test：测试数据集，字段名称由 xsgnlst 决定，一般为标准的 OHLC 数据。包括结果数据字段，默认名为 close_pred，不是传统的变量名称 y2。
- ntree：决策树数目，默认值为 100。
- ndepth：决策树最大深度，默认值为 3。
- ftg：测试数据对比图形文件名，默认值为 tmp/pic001.png。文件名为空时，不绘制图形。
- xsgnlst：训练、测试数据集，对应的是输入数据，默认值是标准的 OHLC

字段名称列表。

- ysgnlst：训练、测试数据集，对应的是结果字段名称，训练结果字段名为 close_next，测试结果字段名为 close_pred。

通常在调用 mxtst_DTree010 函数时，只使用以下参数：

```
    mfun,df_train,df_test,ntree=100,ndepth=3
```

mxtst_DTree010 测试函数，全部代码如下：

```
    def mxtst_DTree010(mfun, df_train,df_test, ntree=100,ndepth=3,
ftg='tmp/pic001.png',xsgnlst=['open','high','low','close'],ysgnls
t=['close_next','close_pred']):
        #1
        print('#1,模型设置')
        mx =mfun(n_estimators=ntree,max_depth=ndepth)
        #2
        print('#2,准备 AI 数据')
        x_train=df_train[xsgnlst].values
        ysgn=ysgnlst[0]
        y_train=df_train[ysgn].values
        #
        x_tst=df_test[xsgnlst].values
        ysgn2=ysgnlst[1]
        #y_tst=df_test[ysgn2].values

        #3
        print('#3,fit 训练模型')
        mx.fit(x_train,y_train)

        #4
        print('#4,predict 模型预测数据')
        df_test[ysgn2]=mx.predict(x_tst) #cross_val_predict(mx,x,
y, cv=20)

        #5
        print('#5,按 1%精度验证模型')
```

```
        dacc,df,xlst=ai_acc_xed2ext(df_test[ysgn],df_test[ysgn2],
1,True)
        #zt.prDF('df',df_test)

        #6
        print('#6,绘制对比曲线')
        if ftg!='':
            css='ntree={0},max_depth={1}'.format(ntree,ndepth)
            pic=df_test[ysgnlst].plot(linewidth=3,title=css)
            fig =pic.get_figure()
            fig.savefig(ftg)
        #
        return dacc
```

关于函数程序，简单说明如下。

- 第 1 组代码，保存模型函数到变量模型。
- 第 2 组代码，准备数据。
- 第 3 组代码，调用 fit 函数，对模型进行训练。
- 第 4 组代码，调用 predict 函数，用训练过的模型生成预测数据。
- 第 5 组代码，对实盘数据、预测数据进行对比分析。
- 第 6 组代码，绘制实盘、预测数据对比曲线图。

为了方便测试，在 TOP 极宽模块库当中，还集成了一个批量测试函数 ai_DT_tst100，函数调用接口如下：

```
    def ai_DT_tst100(mfun, df_train,df_test,ndepth=3,
xsgnlst=['open','high','low','close'],ysgnlst=['close_next','clos
e_pred']):
```

案例 7-2：上证的 RF 回归频道

案例 7-2 的文件名是 kb702_RF_reg100.py，本案例介绍的是决策树测试框架批量测试函数的使用。

案例采用上证指数作为数据源，在以后实盘金融量化分析时，可作为参考案例。

案例核心代码是第 4 组：

```
#4,
print('#4,模型测试')
mfun=ensemble.RandomForestRegressor
#
df9=zai.mxtst_DTree100(mfun,df_train,df_test,ndepth=3)
df9['k3']=df9['k']
df2=zai.mxtst_DTree100(mfun,df_train,df_test,ndepth=5)
df9['k5']=df2['k']
df2=zai.mxtst_DTree100(mfun,df_train,df_test,ndepth=10)
df9['k9']=df2['k']
zt.prDF('df9',df9)
df9[['k3','k5','k9']].plot(linewidth=3)
```

调用 TOP 极宽模块库的 mxtst_DTree100 批量测试函数。注意，在案例中，分别按决策树深度为 3 层、5 层、10 层进行测试。

图 7.4 至图 7.6，分别是决策树深度（3 层、5 层、10 层）的数据测试图。

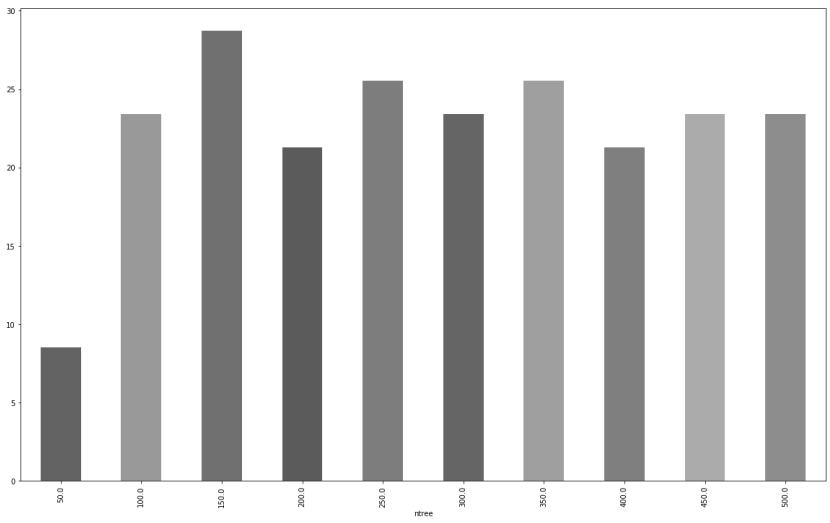


图 7.4　数据测试图，决策树深度 3 层

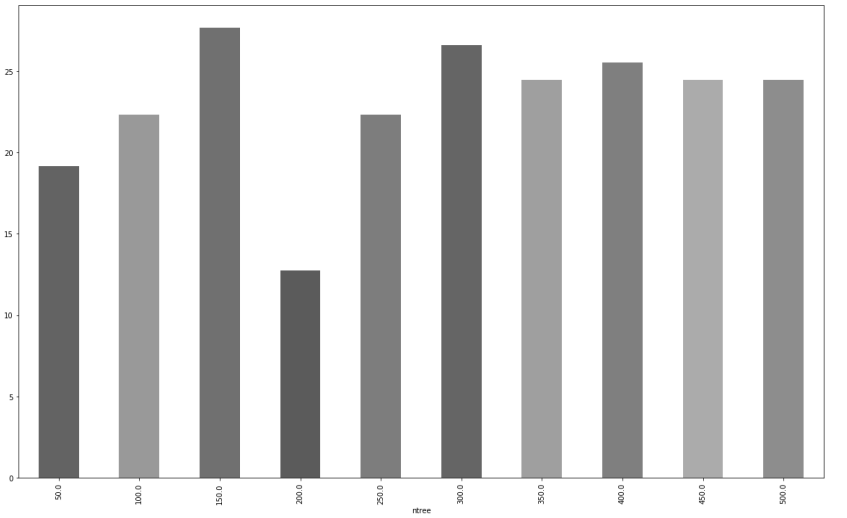


图 7.5　数据测试图，决策树深度 5 层

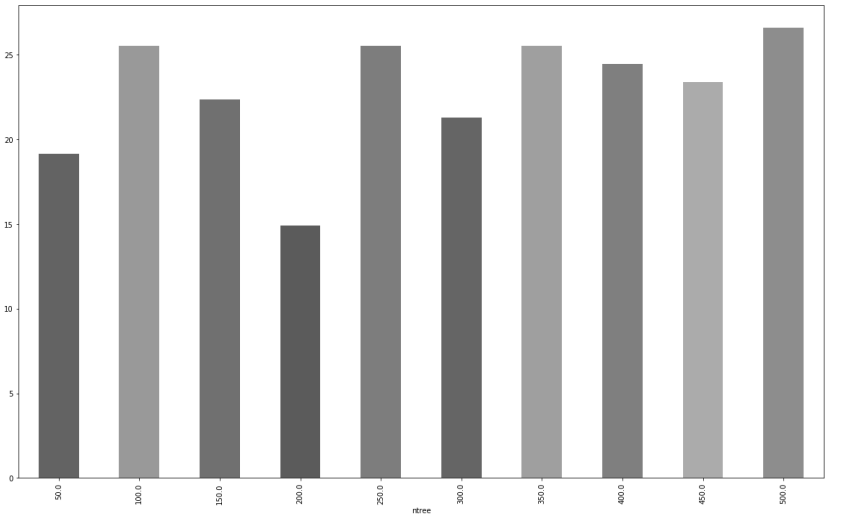


图 7.6　数据测试图，决策树深度 10 层

为了方便对比，我们把三次测试的数据汇总，绘制了一张对比曲线图，如图 7.7 所示。

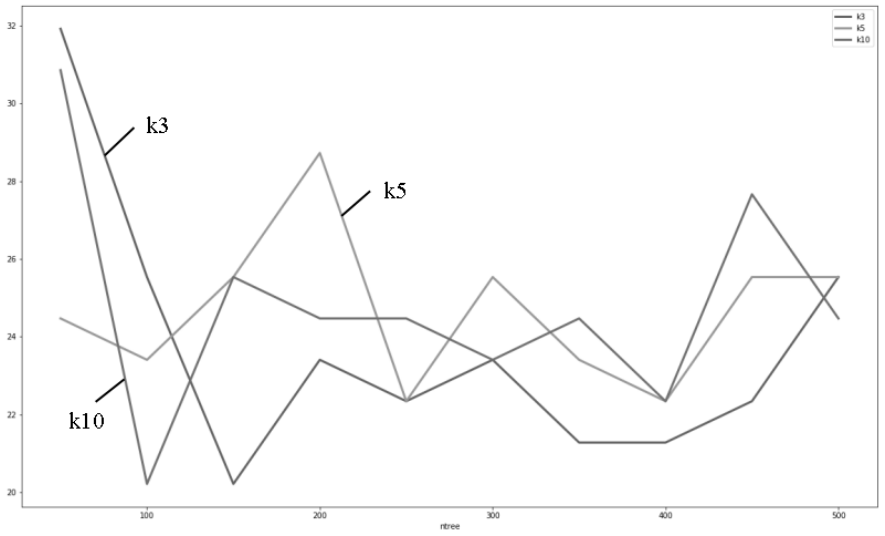


图 7.7　数据测试对比曲线图

在图 7.7 当中，曲线 k3、k5、k10 分别代表决策树深度为 3 层、5 层、10 层的测试数据，图 7.7 对应的测试数据如下：

```
df9
          k  ntree  ndepth     k3     k5    k10
ntree
50.0   31.92   50.0     3.0  31.92  24.47  30.85
100.0  25.53  100.0     3.0  25.53  23.40  20.21
150.0  20.21  150.0     3.0  20.21  25.53  25.53
200.0  23.40  200.0     3.0  23.40  28.72  24.47
250.0  22.34  250.0     3.0  22.34  22.34  24.47
300.0  23.40  300.0     3.0  23.40  25.53  23.40
350.0  21.28  350.0     3.0  21.28  23.40  24.47
400.0  21.28  400.0     3.0  21.28  22.34  22.34
450.0  22.34  450.0     3.0  22.34  25.53  27.66
500.0  25.53  500.0     3.0  25.53  25.53  24.47
```

由以上信息和图 7.7 可以看出，最终的准确度和随机森林算法当中的参数 n_estimators（决策树数目）、max_depth（决策树最大深度）都没有明显的线性关系。所以并非决策树的数目越多，深度越大，准确度就越高。

案例 7-3：当比特币碰到 RF 回归算法

数字货币和量化分析是天生一对。

下面，我们采用 BTC 的实盘交易数据，运行 RF 回归算法批量测试框架，看看效果如何。

案例 7-3 的文件名是 kb703_RF_reg120.py，代码与案例 7-2 基本相同，只是在第 1 组程序当中，数据源部分采用的是 BTC 的实盘交易数据，代替上证指数数据。

```
#1
print('#1,rd data')
fss='data/btc2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
zt.prDF('xdf',xdf)
```

采用 OHLC 标准金融数据格式作为数据分析源非常灵活方便。

以下是部分程序运行的输出数据：

```
df9
          k  ntree  ndepth     k3     k5    k10
ntree
50.0   12.39   50.0     3.0  12.39  11.50  13.27
100.0  13.27  100.0     3.0  13.27  11.50   9.74
150.0  15.04  150.0     3.0  15.04  11.50  12.39
200.0  11.50  200.0     3.0  11.50  12.39  11.50
250.0  13.27  250.0     3.0  13.27  12.39  15.93
300.0  10.62  300.0     3.0  10.62  15.04  11.50
350.0  11.50  350.0     3.0  11.50  14.16  15.93
400.0  13.27  400.0     3.0  13.27  11.50  11.50
450.0  13.27  450.0     3.0  13.27  12.39  12.39
500.0  14.16  500.0     3.0  14.16  13.27  14.16
```

图 7.8 是对比分析数据曲线图。

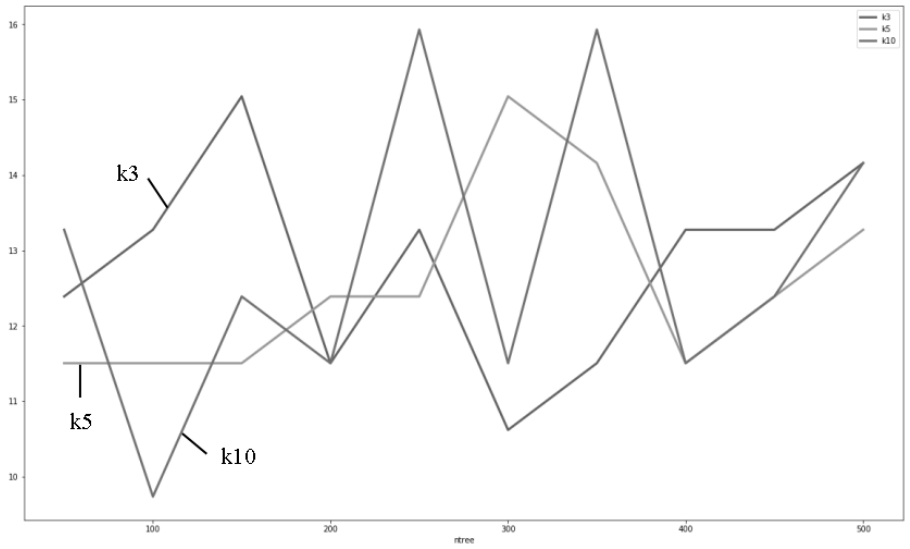


图 7.8　数据测试对比曲线图

由图 7.8 和输出信息可以看出，最终的准确度和随机森林算法当中的参数 n_estimators（决策树数目）、max_depth（决策树最大深度）都没有明显的线性关系。因此并非决策树的数目越多，深度越大，准确度就越高。

此外，BTC 分析平均准确度只有 15%，是上证指数平均准确度的一半。这也许可以从另一角度说明 BTC 的价格波动远远大于股票市场。

案例 7-4：上证和 RF 分类算法

案例 7-4 的文件名是 kb704_RF_cla.py，本案例基于随机森林分类算法对上证指数的趋势进行分析。

分类函数使用的是 ensemble.RandomForestClassifier（随机森林分类）函数。

前面 3 组代码属于数据准备，具体如下：

```
#1
```

```
print('#1,rd data')
fss='data/sh2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
zt.prDF('xdf',xdf)

#2
print('#2,xed data')
#xdf['close_next']=xdf['close'].shift(-1)
xdf['xprice']=xdf['close'].shift(-1)
#xdf['y']=xdf['xprice']
#
xdf['kpr']=xdf['xprice']/xdf['close']*1000
xdf['close_next']=0
xdf.loc[xdf.kpr>1005,'close_next']=2
xdf.loc[xdf.kpr<995,'close_next']=1
xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf',xdf)
#

xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf',xdf)

#3.1
print('#3,cut data')
tim0str,tim9str='2010-10-10','2017-12-31'
df_train=zdat.df_kcut8tim(xdf,'',tim0str,tim9str)
zt.prDF('#3.1 df_train\n',df_train)
#
#3.2
df_test=xdf[xdf.index>'2018'] #.tail(100)
zt.prDF('#3.2 df_test\n',df_test)
```

因为是分类模型，所以首先需要对数据进行分类处理。

分类后的字段名称用 close_next 代替了原来的变量名称 y，以方便调用 mxtst_DTree010 函数。

第 4 组代码如下：

```
    #4,
    print('#4,模型测试')
    mfun=ensemble.RandomForestClassifier
    zai.mxtst_DTree010(mfun,df_train,df_test,ntree=100,ndepth=3,f
tg='tmp/pic001.png')
```

调用 TOP 极宽模块库的 mxtst_DTree010 函数，测试模型的具体效果。

以下是对应的输出信息：

```
    acc: 41.49%
```

41.49%的准确度，看起来模型测试效果不错。

图 7.9 是对应的数据对比曲线图。

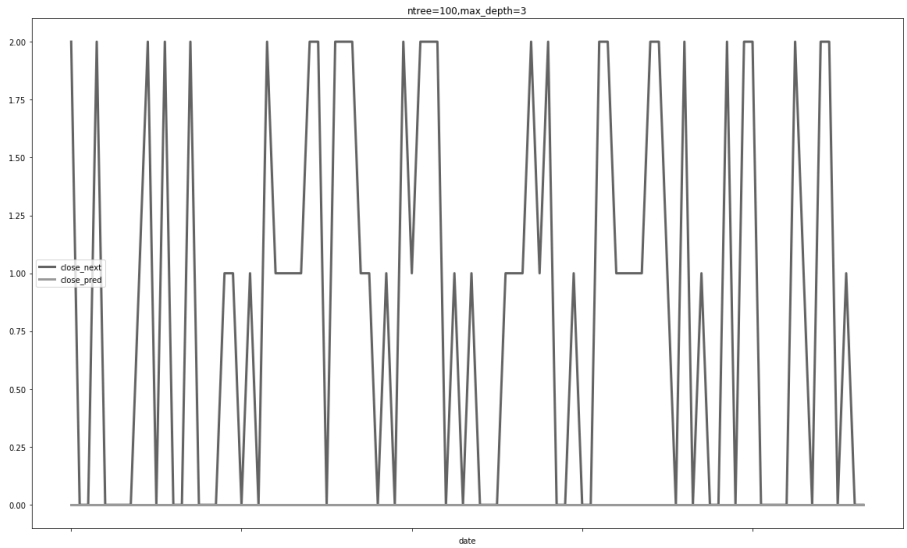


图 7.9　数据对比曲线图

在图 7.9 中，最下面的一条直线是 close_pred 字段，全部为 0，这说明模型预测有着严重的问题。

这个案例也说明，在测试机器学习模型时，一定不能只看表面的测试数据，要深入分析，找到数据背后的真相。

7.4 极端随机树算法

极端随机树（Extremely Randomized Trees）算法，简称是 ET 算法，也称为非常随机树算法或完全随机树算法，如图 7.10 所示。

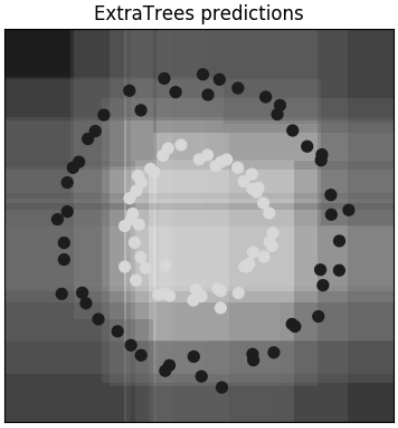


图 7.10 极端随机树算法

ET 算法是由 Pierre Geurts 等人于 2006 年提出，该算法与随机森林算法十分相似，都是由许多决策树构成的，但该算法与随机森林算法有两点主要的区别。

- 随机森林算法采用的是 Bagging 模型，而极端随机树算法使用所有的训练样本得到每棵决策树，也就是说每棵决策树用的是相同的全部训练样本。
- 随机森林算法在一个随机子集内得到最佳分叉属性，而极端随机树算法完全随机地得到分叉值，从而实现对决策树进行分叉。

极端随机树算法是随机森林算法的扩展版本，在 RF 算法基础上增加了一层随机性分析来选择分裂特征，在对连续变量特征选取最优分裂值时，不会计算所有分裂值的效果。

极端随机树算法在 RF 算法采取 Bagging 和 Random Subspace 的基础上，对每一棵决策树选取的 Split Value 采用随机生成。

原先决策树针对的是连续数值的特征，会计算局部 Split Value，但是极端随机树算法，对每一个特征在它的特征取值范围内都会随机生成一个 Split Value，再计算选取一个特征来进行分裂（树多一层）。

极端随机树算法通过遍历节点内的所有特征属性，得到所有特征属性的分叉值，再选择分叉值最大的那种形式实现对该节点的分叉，这种方法比随机森林算法的随机性更强。

对于某棵决策树，由于它的最佳分叉属性是随机选择的，因此它的预测结果往往是不准确的，但将多棵决策树组合在一起，就可以达到很好的预测结果。

7.5　极端随机树函数

sklearn 模块库内置的极端随机树函数收录在 Ensemble（集成学习）子模块当中。

ET 回归函数 ExtraTreesRegressor 的接口定义如下：

```
    ExtraTreesRegressor(n_estimators=10, criterion='mse',
max_depth= None, min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf =0.0, max_features='auto',
max_leaf_nodes=None, min_impurity_decrease =0.0,
min_impurity_split=None, bootstrap=False, oob_score=False, n_jobs
=1, random_state=None, verbose=0, warm_start=False)
```

ET 分类函数 ExtraTreesClassifier 的接口定义如下：

```
    ExtraTreesClassifier(n_estimators=10, criterion='gini',
max_depth=None, min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_features='auto',
max_leaf_nodes=None, min_impurity_decrease=0.0,
min_impurity_split=None, bootstrap=False, oob_score=False,
n_jobs=1, random_state=None, verbose=0, warm_start=False,
class_weight=None)
```

以上两个函数接口，参数定义都差不多，常用的参数如下。

- n_estimators：梯度提升的迭代次数，也是弱分类器的个数，默认值是100。
- max_depth：决策树最大深度，默认值是 3。
- max_features：单个决策树使用的最大特征数量。

案例 7-5：极端随机树回归算法

案例 7-5 的文件名是 kb705_ET_reg.py，本案例介绍的是极端随机树回归算法的使用方法，具体测试算法使用的是极端随机树回归函数。

```
ensemble.ExtraTreesRegressor
```

案例使用了决策树测试框架，由于对有关的测试函数进行了集成，所以代码非常简单。

前面三组代码是数据准备，案例采用上证指数作为数据源。

```
#1
print('#1,rd data')
fss='data/sh2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
zt.prDF('xdf',xdf)

#2
print('#2,xed data')
xdf['close_next']=xdf['close'].shift(-1)
xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf',xdf)

#3.1
print('#3,cut data')
tim0str,tim9str='2010-10-10','2017-12-31'
df_train=zdat.df_kcut8tim(xdf,'',tim0str,tim9str)
zt.prDF('#3.1 df_train\n',df_train)
```

```
    #
    #3.2
    df_test=xdf[xdf.index>'2018'] #.tail(100)
    zt.prDF('#3.2 df_test\n',df_test)
```

在以上代码当中，需要注意的是第 2 组代码当中：

```
    xdf['close_next']=xdf['close'].shift(-1)
```

使用 close_next 作为实盘第二天收盘价的字段名称，代替了以往机器学习当中实盘数据的习惯性变量名称，更加贴近金融量化实盘，也便于随后的图表对比分析。

整个案例最核心的部分在第 4 组，函数测试如下：

```
    #4,
    print('#4,模型测试')
    mfun=ensemble.ExtraTreesRegressor
    zai.mxtst_DTree010(mfun,df_train,df_test,ntree=100,ndepth=3,f
tg='tmp/pic001.png')
```

调用 TOP 极宽模块库的 mxtst_DTree010 函数，对模型函数进行训练、预测，以及数据验证，返回值只有一个 Dacc（准确度）。

其中代码如下：

```
    mfun=ensemble.ExtraTreesRegressor
```

用于定义 sklearn 内置的各种机器学习的模型。

以下是程序运行的输出信息：

```
    按 1%精度验证模型
    acc: 31.91%;
```

按 1%精度验证模型，案例的准确度为 31.9%，这个结果比 RF 随机森林回归算法的 23.4%高 8%左右。

函数对图形进行了优化，曲线图的图标名称不再是简单抽象的 y、y2 变量名称了，而是具体的 close_next（实盘次日收盘价）、close_pred（实盘预测收盘价）。

案例 7-6：上证指数案例应用

案例 7-6 的文件名是 kb706_ET_reg100.py，本案例介绍 ET 回归的批量测试效果。

案例通过调用 TOP 极宽模块库当中集成的决策树测试框架函数进行测试。案例采用上证指数作为数据源。

案例核心代码是第 4 组：

```
#4,
print('#4,模型测试')
mfun=ensemble.ExtraTreesRegressor
#
df9=zai.mxtst_DTree100(mfun,df_train,df_test,ndepth=3)
df9['k3']=df9['k']
df2=zai.mxtst_DTree100(mfun,df_train,df_test,ndepth=5)
df9['k5']=df2['k']
df2=zai.mxtst_DTree100(mfun,df_train,df_test,ndepth=10)
df9['k10']=df2['k']
zt.prDF('df9',df9)
df9[['k3','k5','k10']].plot(linewidth=3)
```

调用 TOP 极宽模块库的 mxtst_DTree100 函数，分别按决策树深度为 3 层、5 层、10 层进行测试。

以下是部分对应的测试数据：

```
df9
          k  ntree  ndepth     k3     k5    k10
ntree
50.0   44.68   50.0     3.0  44.68  36.17  38.30
100.0  45.74  100.0     3.0  45.74  41.49  43.62
150.0  39.36  150.0     3.0  39.36  40.43  40.43
200.0  37.23  200.0     3.0  37.23  41.49  43.62
250.0  40.43  250.0     3.0  40.43  39.36  45.74
300.0  39.36  300.0     3.0  39.36  41.49  40.43
350.0  43.62  350.0     3.0  43.62  41.49  41.49
```

```
400.0  43.62  400.0    3.0  43.62  38.30  44.68
450.0  39.36  450.0    3.0  39.36  37.23  42.55
500.0  42.55  500.0    3.0  42.55  40.43  41.49
```

图 7.11 是对应的是测试数据对比曲线图。

在图 7.11 当中，曲线 k3、k5、k10 分别代表决策树深度为 3 层、5 层、10 层的测试数据。

由以上信息和图 7.11 可以看出，最终的准确度和 ET（极端随机树）算法当中的参数 n_estimators（决策树数目）、max_depth（决策树最大深度）都没有明显的线性关系，因此，并非决策树的数目越多，深度越大，准确度就越高。

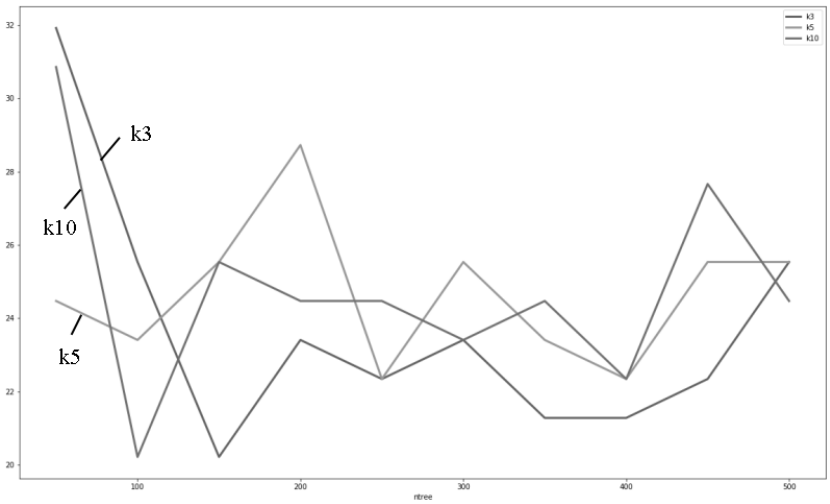


图 7.11　数据测试对比曲线图

案例 7-7：ET、比特币，谁更极端

案例 7-7 的文件名是 kb707_ET_cla.py，本案例是基于 ET 分类函数，对 BTC 的价格趋势进行分析。

分类函数使用的是 ensemble. ExtraTreesClassifier 函数。

数据源采用的是 BTC（比特币）实盘交易数据，为便于大家以后结合实盘操作，导入数据、训练数据字段是标准的 OHLC 金融数据。

前面 3 组代码属于数据准备，具体如下：

```
#1
print('#1,rd data')
fss='data/btc2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
zt.prDF('xdf',xdf)

#2
print('#2,xed data')
#xdf['close_next']=xdf['close'].shift(-1)
xdf['xprice']=xdf['close'].shift(-1)
#xdf['y']=xdf['xprice']
#
xdf['kpr']=xdf['xprice']/xdf['close']*1000
xdf['close_next']=0
xdf.loc[xdf.kpr>1005,'close_next']=2
xdf.loc[xdf.kpr<995,'close_next']=1
xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf',xdf)
#

xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf',xdf)

#3.1
print('#3,cut data')
tim0str,tim9str='2010-10-10','2017-12-31'
df_train=zdat.df_kcut8tim(xdf,'',tim0str,tim9str)
zt.prDF('#3.1 df_train\n',df_train)
#
#3.2
```

```
    df_test=xdf[xdf.index>'2018'] #.tail(100)
    zt.prDF('#3.2 df_test\n',df_test)
```

因为是分类模型，所以首先需要对数据进行分类处理。

分类后的字段名称用 close_next 代替了原来的变量名 y，以方便调用 mxtst_DTree010 函数。

第 4 组代码如下：

```
    #4,
    print('#4,模型测试')
    mfun=ensemble.ExtraTreesClassifier
    zai.mxtst_DTree010(mfun,df_train,df_test,ntree=100,ndepth=3,f
tg='tmp/pic001.png')
```

调用 TOP 极宽模块库的 mxtst_DTree010 函数，测试模型的具体效果。

以下是对应的输出信息：

```
    acc: 48.67%;
```

48.67%的准确度，表面上看起来模型测试效果不错。

图 7.12 是对应的数据对比曲线图。

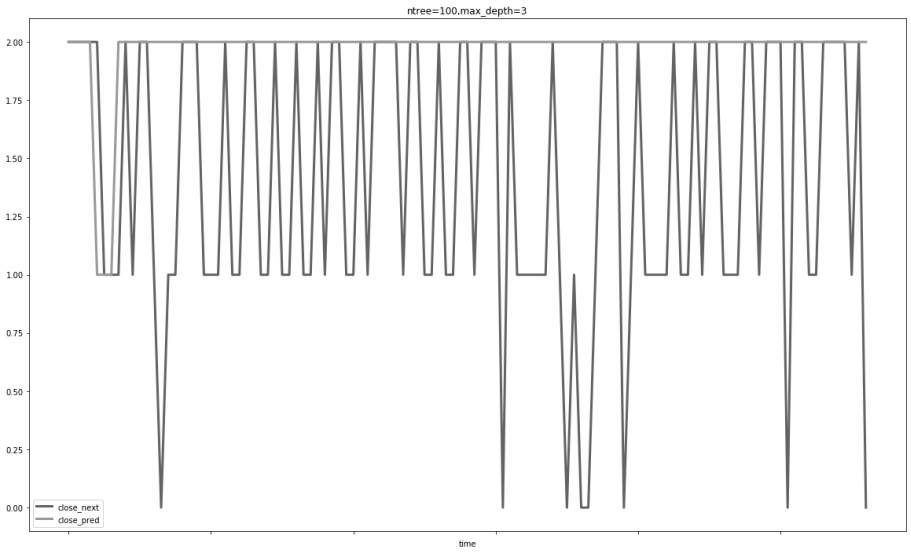


图 7.12　数据对比曲线图

在图 7.12 中，最上面的一条线是 close_pred 字段，基本全部为 2，只是在刚开始的时候，有部分预测值是 1。这个模型预测依然有着严重的问题。

这个案例也说明，在测试机器学习模型时，一定不能只看表面的测试数据，要深入分析，找到数据背后的真相。

第 8 章

机器学习算法模式

由图 8.1 可以看出机器学习的三大要素：数据、算法模型、计算。

- 数据：目前是大数据时代，各行各业基本上都不缺数据，缺乏的只是从数据当中提炼出有价值的参数。
- 计算：目前都有封装好的函数，学习训练期间采用 fit 训练函数，实盘采用 predict 预测函数，只是不同模型的 API 接口有所差别。
- 算法模型：这个需要经验和数据测试，根据不同的项目挑选不同的模型。

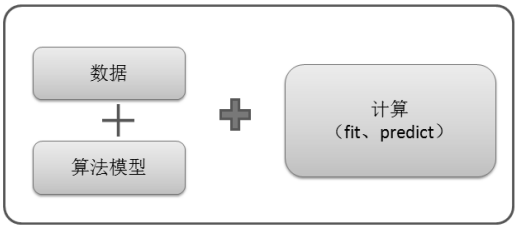


图 8.1　机器学习三大要素

有专家学者统计，目前常用的机器学习算法模型大约有 30 多种，加上衍生版本，大约有 300 多种（这么多深度学习算法模型，每个模型背后都涉及高深、专业的理论知识，比如数学、医学、编程，甚至哲学）。

本体论，目前也是深度学习、神经网络算法前沿最受关注的领域之一。

TensorFlow 的出现，让使用神经网络模型的门槛越来越低。对于读者而言，大家采用笔者介绍的黑箱模式，只需要使用 TensorFlow 模块库以及各种 TensorFlow 第三方软件定制中已经收录的算法模型即可。

目前人工智能大热，各种算法模型层出不穷，即使是谷歌公司也无法在 TensorFlow 当中收录所有的算法模型。不过，常用的、经典的算法模型都会收录在 TensorFlow 中。

对于初学者而言，sklearn 当中的经典机器学习算法模型都是非常基础的，也是一些最常用的、最经典的算法模型。

目前常用的机器学习算法模型如下。

- PP（概率编程）
- RA（回归算法）
- IA（实例算法）
- RA（正则化方法）
- DT（决策树）
- NB（贝叶斯算法）
- CA（聚类分析）
- DRA（降维算法）
- ARLA（关联算法）
- SVM（支持向量机）
- EA（集成算法）
- ANN（神经网络算法）
- DP（深度学习）

前面我们已经学习了最简单的线性回归算法、逻辑回归算法，以及完整的决策树算法。

下面介绍一些常见的、经典的机器学习算法，通过这种从局部、系统，再系统、局部的多次迭代式学习，大家应该会更全面、更深入地理解机器学习中的各种算法模型。

本章以理论性的介绍为主，没有案例，有些内容可能比较抽象、难于理解，因此建议大家还是采用“黑箱模式”，等以后有了基础再进行深入学习。

机器学习的算法模型很容易引起大家的困惑，我们从三个方面对常见的机器学习算法进行一些简单的分类。

- 机器学习的方式。
- 算法模型的类似性。
- 工程应用的规范性。

当然，以上三种机器学习算法分类都是基于不同的、独立的角度，相互之间有很大的重复性，这属于正常情况。本章只是按笔者个人的习惯，对一些常见的机器学习算法做一个汇总介绍，以便于初学者从全局把握学习。有关分类，纯属一家之言，只是抛砖引玉，欢迎大家对相关内容进一步优化。

8.1　学习模式

根据机器学习算法模型对于训练数据处理方式的不同，通常可以把机器学习算法分为两大类：监督学习、无监督学习。近年来，随着深度学习、神经网络的发展，又增加了一类：强化学习，不过这个属于神经网络领域，本书会稍加说明。

在企业实际应用当中，最常用的就是监督学习和无监督学习两种模式。在图像识别等领域，由于存在大量的非标识的数据和少量的可标识数据，半监督式学习成为一个很热门的话题。强化学习更多应用在机器人控制和其他需要进行系统控制的领域。

1．监督学习

监督学习主要用于一部分数据集（训练数据）有某些可以获取的标签，但剩余的样本缺失并且需要预测的场景中。

监督学习模式采用的输入数据被称为“训练数据”，每组训练数据有一个明确的标识或结果，如防垃圾邮件系统中的“垃圾邮件”“非垃圾邮件”，手写数字识别中的“1”“2”“3”“4”等。

利用“训练数据”，算法模型会生成映射函数，训练过程会不断持续，直到模型在训练数据上获得期望的精确度。

监督学习的常见应用场景有分类问题和回归问题，常见监督学习算法有回归算法、DT（决策树）算法、RF（随机森林）算法、K 近邻算法，以及 BP（反向传递）神经网络算法等。

2．sklearn监督学习

在 sklearn 模块库当中，与监督学习有关的算法模型如下。

- 广义线性模型（Generalized Linear Models）
- 线性和二次判别分析（Linear and Quadratic Discriminant Analysis）
- 内核岭回归（Kernel Ridge Regression）
- 支持向量机（Support Vector Machines）
- 随机梯度下降（Stochastic Gradient Descent）
- 最近邻（Nearest Neighbors）
- 高斯过程（Gaussian Processes）
- 交叉分解（Cross decomposition）
- 朴素贝叶斯（Naive Bayes）
- 决策树（Decision Trees）
- 集成方法（Ensemble methods）
- 多分类和多标签算法

- 特征选择
- 等式回归
- 概率校准
- 神经网络模型（监督）

3．无监督学习

无监督学习主要用于从未标注数据集中挖掘相互之间的隐含关系的情况。

无监督学习模式，数据无须预先进行标识，学习模型是为了推断出数据的一些内在结构，例如关联规则的学习和聚类分析等。

无监督学习模式没有任何目标变量或结果变量要预测或估计。这个算法用在不同的组内聚类分析。这种分析方式被广泛地用来细分客户，根据干预的方式分为不同的用户组。

常见的无监督学习算法模型包括 Apriori 关联算法和 k-Means 算法。

4．sklearn无监督学习

在 sklearn 模块库当中，与无监督学习有关的算法模型如下。

- 高斯混合模型
- 流形学习
- 聚类
- 双聚类
- 分解成分中的信号（矩阵分解问题）
- 协方差估计
- 新奇和异常检测
- 密度估计
- 神经网络模型（无监督）

5. 半监督学习

严格说来，半监督学习模式也是监督学习的一种。

只是半监督学习模式输入数据的一部分被标识，一部分没有被标识，特别在“训练数据”有限的情况下。

半监督学习模式可以用来进行预测，但是模型首先需要学习数据的内在结构，以便合理地组织数据来进行预测。

半监督学习算法主要是一些常用监督学习、分类回归算法的延伸，算法包括图论推理算法（Graph Inference）和拉普拉斯支持向量机（Laplacian SVM）等。

6. 强化学习

强化学习是近年神经网络、深度学习的产物，主要用于动态系统处理以及机器人控制等。强化学习算法可以训练机器进行决策。

在强化学习模型当中，输入数据作为模型的反馈，不像监督模型，输入数据仅仅是作为一个检查模型对错的方式。

强化学习介于监督学习、无监督学习两者之间，每一步预测或者行为都或多或少有一些反馈信息，但是却没有准确的标签或者错误提示。

在强化学习模型当中，系统被放在一个能让它通过反复试错来训练自己的环境中。机器从过去的经验中进行学习，并且尝试做出精确的判断。

强化学习常见算法包括马尔可夫决策过程、Q-Learning 和时间差学习（Temporal Difference Learning）。

8.2 机器学习五大流派

在网络上，曾经有人把机器学习分成五大流派。

- Symbolists（符号学派）：使用符号、规则和逻辑来表征知识和进行逻辑推理，关注哲学、逻辑学和心理学，将机器学习视为逆向演绎（Inverse

of Deduction），常用算法是决策树。

- Bayesians（贝叶斯学派）：注重统计学和概率推理，获取发生的可能性来进行概率推理，常用算法是朴素贝叶斯或马尔可夫。
- Connectionists（联结学派）：使用概率矩阵和加权神经元来动态地识别和归纳模式，专注物理学和神经科学，重点研究大脑的逆向工程，常用算算是法神经网络。
- Evolutionary（进化学派）：在遗传学和进化生物学的基础上，设计算法模型，计算并分析其中的最优解，常用算法是遗传算法。
- Analogizer（类推学派）：根据约束条件来优化函数，关注心理学和数学优化，以此来推断结果，常用算法是 SVM（支持向量机）。

8.3　经典机器学习算法

经典机器学习算法不是通过算法模型的内在逻辑和理论关联，而是看机器学习算法模型在学术工程领域的应用程度，或者知名度来对算法模型进行排名。

这种排名虽然看起来有些不够客观，但是换个角度思考，这未免不是一种实用主义哲学在人工智能领域的应用。

机器学习常用的经典算法如下。

- PP（概率编程）
- RA（回归算法）
- IA（实例算法）
- RA（正则化方法）
- DT（决策树）
- NB（贝叶斯算法）
- CA（聚类分析）

- DRA（降维算法）
- ARLA（关联算法）
- SVM（支持向量机）
- EA（集成算法）
- ANN（神经网络算法）
- DP（深度学习）

8.4 小结

机器学习最大的用处是通过对历史数据的分析，找出其中的潜在规律，从而对未来进行预测。

传统的机器学习软件库 sklearn 比 TensorFlow 等新一代神经网络、深度学习系统更加直截了当。

sklearn 机器学习软件用 fit 函数建模后，直接使用 predict 函数对新数据进行分析。

通过本章，我们从多个角度对机器学习算法模型进行了分类梳理，大家对常用的机器学习模型应该有了基本了解。

本书后面的章节，会进一步介绍这些机器算法模型在金融量化领域的具体使用。

第 9 章

概率编程

概率编程，英文全称是 Probabilistic Programming，简称 PP，其更加流行的专业名称是概率编程语言（Probabilistic Programming Language，简称 PPL）。概率编程是与传统面向过程、OOP（面向对象）完全不同的编程模式。在人工智能领域，概率编程与传统的机器学习、神经网络完全不同，是完全独立的另外一个体系。

PP（概率）模型一直是机器学习、神经网络的重点领域之一。

- Pyro 概率编程语言：Uber 与斯坦福大学最新的研究成果，该语言基于 Python 与 PyTorch，专注于变分推理，同时支持可组合推理算法。Pyro 的推理算法，通过随机函数快速构建神经网络模型。
- WebPPL：一个基于浏览器的在线概率编程网站，以教学为主。
- Edward：谷歌推出的概率编程框架，是一个用于概率建模、推理和评估的 Python 库。它是一个用于快速实验和研究概率模型的测试平台，目标是融合贝叶斯统计学、机器学习、深度学习、概率编程几个领域。Edward 构建于 TensorFlow 之上，它支持计算图、分布式训练、CPU/GPU 集成、自动微分等功能，可以使用 TensorBoard 进行可视化分析。

对于金融量化分析领域，PP 模型可能比神经网络、深度学习算法更重要，理由如下。

- 金融量化项目本身就是基于概率的推理编程。
- PP 模型比其他神经网络有更加深厚的理论基础。
- PP 模型算法简单清晰，便于理解。

PP 模型大部分都是基于 Bayesian（贝叶斯）算法，目前常见的 PP 算法如下。

- Bayesian（贝叶斯）算法。
- Bayesian Inference（贝叶斯推理机）。
- MCMC（马尔科夫链蒙特卡罗）算法。
- 序列蒙特卡罗推理算法。
- HMC（哈密顿蒙特卡罗）算法。

PP 算法更多属于神经网络、深度学习领域，就本书的读者而言，有些过于专业了，不过在 sklearn 机器学习模块库当中，也集成了两组和 PP 算法相关的函数。

一组函数是基于（贝叶斯）算法，如下所示。

- MultinomialNB，多项式朴素贝叶斯算法。
- GaussianNB，高斯朴素贝叶斯算法。
- BernoulliNB，伯努利朴素贝叶斯算法。

另外一组函数是基于 HMM（隐马尔可夫）算法，如下所示。

- GaussianHMM，高斯分布 HMM 模型函数。
- GMMHMM，高斯混合的 HMM 模型函数。
- MultinomialHMM，多项式（离散）HMM 模型函数。

9.1 朴素贝叶斯的上证之旅

朴素贝叶斯分类是一系列分类算法的总称，这类算法均以朴素贝叶斯

定理为基础，故统称为朴素贝叶斯分类。

朴素贝叶斯算法，英文名称是 *Naive* Bayesian，简称 NB 算法。NB 算法模型易于建造，且对于大型数据集非常有用。朴素贝叶斯的表现超越了非常复杂的分类方法。朴素贝叶斯定理提供了一种 *P(c)*、*P(x)* 和 *P(x|c)* 计算后验概率 *P(c|x)* 的方法。

NB 算法模型由两种概率组成，它们都能从训练数据中直接计算出来。

- 每个类别的概率。
- 对于给定的 x 值，每个类别的条件概率。

一旦计算出来以上概率数值，NB 算法模型就可以使用朴素贝叶斯定理对新的数据进行预测。当数据是真实值时，通常会假定一个钟形的高斯分布曲线，这样就很容易计算出这些数据的概率。

理论上，NB 算法误差率很小，模型所需的参数也很少，对缺失数据不太敏感，算法也比较简单。

朴素贝叶斯假定每个输入变量都是独立的，所以被称为“朴素的”。学过统计学的读者一定都知道，朴素贝叶斯模型在信息领域内有着无与伦比的地位与应用，如图 9.1 所示。

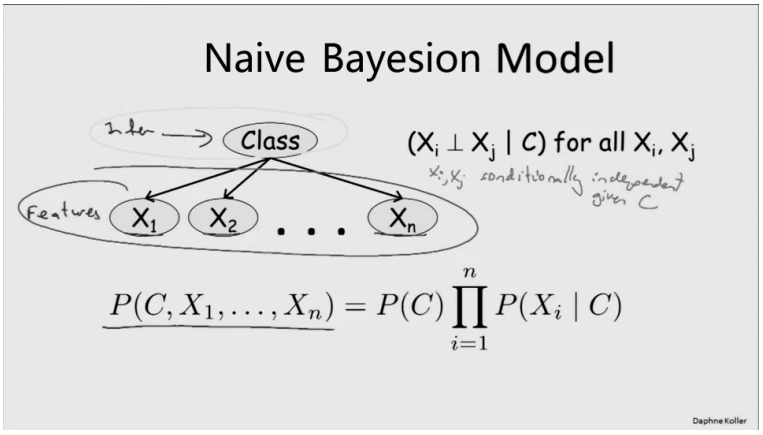


图 9.1　朴素贝叶斯模型

在属性个数比较多或者属性之间相关性较大时，NB 算法模型的分类效

率比不上决策树模型。而在属性相关性较小时，NB 算法模型的性能较好。

NB 算法的优点是快速、易于训练，给出所需的资源能带来良好的表现，对于在小数据集上有显著特征的相关对象可对其进行快速分类。NB 算法的缺点是如果输入的变量是相关的，则会出现问题。

NB 算法目前主要应用于以下领域：检测垃圾电子邮件、新闻自动分类、文字情绪分析、人脸检测等。

案例 9-1：上证朴素贝叶斯算法

案例 9-1 的文件名是 kb901_nbClass.py，本案例主要介绍朴素贝叶斯算法的应用及如何使用该算法对上证指数的趋势进行分析。

具体分类函数使用的是 GaussianNB（高斯朴素贝叶斯）分类函数。

前面几组代码属于数据准备，具体如下：

```
#1
print('#1,rd data')
fss='data/sh2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
zt.prDF('xdf',xdf)

#2
print('#2,xed data')
xdf['xprice']=xdf['close'].shift(-1)
#
xdf['kpr']=xdf['xprice']/xdf['close']*1000
xdf['close_next']=0
xdf.loc[xdf.kpr>1005,'close_next']=2
xdf.loc[xdf.kpr<995,'close_next']=1
xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf',xdf)
```

```
    #3.1
    print('#3,cut data')
    tim0str,tim9str='2010-10-10','2017-12-31'
    df_train=zdat.df_kcut8tim(xdf,'',tim0str,tim9str)
    zt.prDF('#3.1 df_train\n',df_train)
    #
    #3.2
    df_test=xdf[xdf.index>'2018'] #.tail(100)
    zt.prDF('#3.2 df_test\n',df_test)
```

在以上代码当中，需要注意的是第 2 组代码当中的如下行：

```
    xdf['close_next']=xdf['close'].shift(-1)
```

数据准备阶段，因为是分类模型，所以需要先把数据转换为分类类型，这个主要在第 2 组代码当中完成：

```
    xdf['kpr']=xdf['xprice']/xdf['close']*1000
    xdf['close_next']=0
    xdf.loc[xdf.kpr>1005,'close_next']=2
    xdf.loc[xdf.kpr<995,'close_next']=1
    xdf.fillna(method='pad',inplace=True)
```

因为股票市场波动很小，我们按收盘价千分之五的上下波动对数据进行分类，分别用“2”“1”“0”表示，如下所示。

- 2，波动大于 100.5%，up 上涨模式。
- 1，波动小于 99.5%，down 下跌模式。
- 0，波动在 0.5%之间，eq 平稳模式。

以下是前面几组代码运行后，对应的部分输出信息：

```
    #3.2 df_test

                  open    high    close     low      volume
amount   xprice     kpr  close_next
    date
    2018-01-02  3314.03  3349.05  3348.33  3314.03  20227886000
227788461113  3369.11  1006.21          2
    2018-01-03  3347.74  3379.92  3369.11  3345.29  21383614900
```

```
258366523235  3385.71  1004.93         0
    2018-01-04  3371.00  3392.83  3385.71  3365.30  20695528800
243090768694  3391.75  1001.78         0
    2018-01-05  3386.46  3402.07  3391.75  3380.24  21306068100
248187840542  3409.48  1005.23         2
    2018-01-08  3391.55  3412.73  3409.48  3384.56  23616510600
286213219095  3413.90  1001.30         0
    2018-01-09  3406.11  3417.23  3413.90  3403.59  19148855100
238249975070  3421.83  1002.32         0
    2018-01-10  3414.11  3430.21  3421.83  3398.84  20909499700
254515441261  3425.34  1001.03         0
    2018-01-11  3415.58  3426.48  3425.34  3405.64  17381213300
218414134129  3428.94  1001.05         0
    2018-01-12  3423.88  3435.42  3428.94  3417.98  17406340400
215961455748  3410.49   994.62         1
    2018-01-15  3428.95  3442.50  3410.49  3402.31  23200928300
286362732919  3436.59  1007.65         2

                open    high    close     low      volume
amount   xprice     kpr  close_next
    date
    2018-05-11  3179.80  3180.76  3163.26  3162.21  13065974900
167364290366  3174.03  1003.40         0
    2018-05-14  3167.04  3183.82  3174.03  3163.48  12932735300
172410691054  3192.12  1005.70         2
    2018-05-15  3180.42  3192.81  3192.12  3164.52  12454905100
162990790010  3169.56   992.93         1
    2018-05-16  3180.23  3191.95  3169.56  3166.81  13052496800
174590979834  3154.28   995.18         0
    2018-05-17  3170.01  3172.77  3154.28  3148.62  11399556700
150598842185  3193.30  1012.37         2
    2018-05-18  3151.08  3193.45  3193.30  3144.78  13651691800
168038057477  3213.84  1006.43         2
    2018-05-21  3206.18  3219.74  3213.84  3203.34  16445941300
202663464515  3214.35  1000.16         0
    2018-05-22  3211.25  3214.59  3214.35  3192.23  14429268400
```

```
185721667752  3168.96   985.88          1
    2018-05-23  3205.44  3205.44  3168.96  3168.96  15780764800
199358101015  3154.65   995.48          0
    2018-05-24  3167.94  3173.53  3154.65  3152.07  12408580000
160658185502  3154.65   995.48          0
```

第 4 组代码如下：

```
    #4
    print('#4,准备 AI 数据')


    clst=['open','high','low','close']
    x=df_train[clst].values
    y=df_train['close_next'].values
    #
    xtst=df_test[clst].values
```

按机器学习要求转换数据格式。

第 5 组代码如下：

```
    #5
    print('#5,模型设置')
    mx = GaussianNB()
```

调用 GaussianNB（高斯朴素贝叶斯）分类函数生成模型，并保存在变量 mx 当中。

第 6 组代码如下：

```
    #6
    print('#6,fit 训练模型')
    mx.fit(x,y)
```

对模型进行训练。

第 7 组代码如下：

```
    #7
    print('#7,predict 模型预测数据')
    df_test['close_pred']=mx.predict(xtst)
#cross_val_predict(mx,x, y, cv=20)
    zt.prDF('df',df_test)
```

模型训练完成后，调用模型内置的 predict 预测函数，使用测试数据集 df_test 生成验证数据，并绘制对比图。

第 8 组代码如下：

```
    #8
    print('#8,,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
    #
    df_test[['close_next','close_pred']].plot(linewidth=3)
```

验证测试数据集 df_test 的准确度，并绘制相关的对比曲线图。

对应的输出信息是：

```
    #8,,按 1%精度验证模型
    ky0=1; n_df9,94,n_dfk,31
    acc: 32.98%;  MSE:1.82, MAE:1.05,  RMSE:1.35, r2score:-1.60,
@ky0:1.00
```

准确度是 32.98%。

第 9 组代码如下：

```
    #9
    print('\n#9,value_counts')
    print("\ndf_train['close_next'].value_counts()")
    print(df_train['close_next'].value_counts())
    #
    print("\ndf_test['close_next'].value_counts()")
    print(df_test['close_next'].value_counts())
    print("\ndf_test['close_pred'].value_counts()")
    print(df_test['close_pred'].value_counts())
```

这个是分类案例附带的程序，用于统计实盘数据和预测数据当中数值为 2、1、0，即上涨、下跌、平稳的具体数值，对应的输出信息如下：

```
    #9,value_counts

    df_train['close_next'].value_counts()
    0    777
```

```
2    517
1    468
Name: close_next, dtype: int64

df_test['close_next'].value_counts()
0    39
2    28
1    27
Name: close_next, dtype: int64

df_test['close_pred'].value_counts()
0    69
2    25
Name: close_pred, dtype: int64
```

9.2　隐马尔可夫模型

隐马尔可夫模型，英文全称是 Hidden Markov Model，或简称为 HMM 模型。

隐马尔可夫模型，通过分析可见数据，来计算隐藏状态的发生。随后，借助隐藏状态分析，隐马尔可夫模型可以估计未来可能的观察模式，如图 9.2 所示。

在图 9.2 中，通过隐马尔可夫模型，已知高气压、低气压的发生概率（这是隐藏状态）可预测晴天、雨天、多云天的概率。

隐马尔可夫模型的优点是允许数据变化，常用于识别和预测操作。

与其他机器学习算法不同，隐马尔可夫模型我们没有直接使用 sklearn 内置的相关函数，而是使用 hmmlearn 模块库。

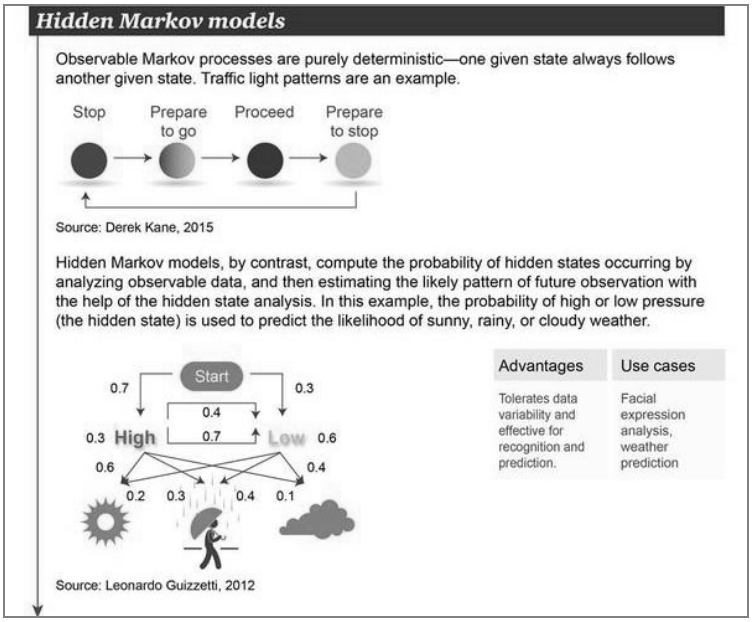


图 9.2　隐马尔可夫模型

hmmlearn 模块库是著名的 Python 机器学习库，也是从 sklearn 模块库衍生出的独立项目，函数的 API 接口和 sklearn 完全兼容。

hmmlearn 模块库提供了三个 HMM 模型。

- GaussianHMM，高斯分布 HMM 模型函数。
- GMMHMM，高斯混合的 HMM 模型函数。
- MultinomialHMM，多项式（离散）HMM 模型函数。

案例 9-2：HMM 模型与模型保存

相对于前面的案例而言，隐马尔可夫算法比较复杂，所以我们拆分成两个独立的案例进行介绍。此外，我们还顺带介绍 sklearn 机器学习模型的保存和读取。

案例 9-2 的文件名是 kb902_hmm01.py，本案例基于隐马尔可夫模型对

上证指数的价格趋势进行分析。

具体分类函数使用的是 hmm.GaussianHMM（高斯分布 HMM 模型）函数。

前面几组代码属于数据准备，代码如下：

```
#1
print('#1,rd data')
fss='data/sh2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
zt.prDF('xdf',xdf)

#2
print('#2,xed data')
xdf['xprice']=xdf['close'].shift(-1)
#xdf['y']=xdf['xprice']
#
xdf['kpr']=xdf['xprice']/xdf['close']*1000
#xdf['y']=0
xdf['close_next']=0
xdf.loc[xdf.kpr>1005,'close_next']=2
xdf.loc[xdf.kpr<995,'close_next']=1
xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf',xdf)

#3.1
print('#3,cut data')
tim0str,tim9str='2010-10-10','2017-12-31'
df_train=zdat.df_kcut8tim(xdf,'',tim0str,tim9str)
zt.prDF('#3.1 df_train\n',df_train)
#
#3.2
df_test=xdf[xdf.index>'2018'] #.tail(100)
zt.prDF('#3.2 df_test\n',df_test)
```

```
#4
print('#4,准备 AI 数据')
clst=['open','high','low','close']
x=df_train[clst].values
y=df_train['close_next'].values
#
xtst=df_test[clst].values
```

因为 GaussianHMM 函数采用分类模型，所以我们只能进行趋势预测，这需要事先对数据进行处理，把价格数据转换为类似足彩“310”（胜平负）的趋势数据。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mn9=6
mx = GaussianHMM(n_components=mn9)
```

调用 GaussianHMM（高斯隐马尔可夫）函数，生成模型，并保存在变量 mx 当中，调用参数：

```
n_components=mn9
```

表示隐马尔可夫生成的状态数目，这个参数涉及模型背后的理论知识。简单来说，模型生成的状态数目对应的是前面我们设置的价格趋势分类。

因为股票市场波动很小，我们按收盘价千分之五的上下波动，对数据进行分类，如下所示。

- 2，波动大于 100.5%，up 上涨模式。
- 1，波动小于 99.5%，down 下跌模式。
- 0，波动在 0.5%之间，eq 平稳模式。

以上分类类似足彩博弈的“310”（胜平负）模式。

所以，调用 GaussianHMM 函数时，一般采用实际分类的倍数进行调用，笔者测试结果如下。

- n=3，模型准确度为 37.23%。

- n=6，模型准确度为 41.49%。
- n=9，模型准确度为 41.49%。
- n=15，模型准确度为 41.49%。

分类数目为实盘分类两倍数值时，准确度有所提高，再继续增加分类数目，准确度并没有实质性提高。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
#mx.fit(x,y)
mx.fit(x)
```

输入训练数据集，对模型进行训练，也就是机器学习的核心。通过输入数据集，不断调整优化模型的内部参数。

需要注意的是，隐马尔可夫模型，调用 fit 训练函数时：

```
mx.fit(x)
```

输入数据只有一组数据 x，没有对照的标签数据 y 字段，而通常的模型函数 fit 调用格式如下：

```
#mx.fit(x,y)
```

需要同时输入训练属性数据 x 字段和标签数据 y 字段。

这是因为 HMM 隐马尔可夫算法模型会根据输入的属性数据 x 字段，自动对结果进行分类，也就是刚才调用 GaussianHMM 函数时设置的 n_components 参数数值。

GaussianHMM 函数生成的分类数据并不能直接使用，需要进行二次转换才行。

第 7 组代码如下：

```
#7
print('#7,保存训练模型')
fmx='tmp/hmm01.dat'
joblib.dump(mx,fmx)
print('fmx,',fmx)
```

保存算法模型。

需要注意的是，在案例当中，我们保存的是训练好的模型。

当然，也可以在模型调用 fit 训练函数前，额外保存于一个空白模型之中，然后通过输入不同的训练数据，再进行测试。

有机器学习、神经网络、深度学习经验的读者都知道，机器学习最耗费时间的环节就是 fit 模型训练。目前，标准的神经网络模型往往有上百层，采用 GPU 集群，也要数日的训练时间。

因此，及时保存训练好的模型，对于模型复用、实盘分析，非常重要。事实上，谷歌、Facebook 等大企业在 GitHub 项目网站上都提供了大量的预先训练好的经典模型供大家参考调用。

在案例当中使用 sklearn 模块库当中的 joblib 函数来保存、读取模型，参见模块导入语句：

```
from sklearn.externals import joblib
```

案例 9-3：HMM 算法与模型读取

案例 9-3 的文件名是 kb903_hmm02.py，本案例是前面案例的延伸，也基于隐马尔可夫模型对上证指数的价格趋势进行分析。

前面几组代码，属于数据准备，和前面的案例 9-2 大同小异，我们不再重复。

第 5 组代码如下：

```
#5
print('#5,读取 fit 训练好的模型数据')
fmx='data/hmm01.dat'
print('fmx,',fmx)
mx = joblib.load(fmx)
```

读取训练好的模型数据，并保存到 mx 变量。

第 6 组代码如下：

```
#6
print('#6,生成 hmm 模型状态数据')
print('\npred')
df_train['sta'] = mx.predict(x)
#zt.prDF('df_train',df_train)
```

调用模型的 predict 预测函数，生成 sta 状态数据。注意，函数输入的是 train 训练数据集。这也是隐马尔可夫模型与其他模型不同的地方。传统机器学习算法模型，在完成 fit 训练后，predict 预测函数输入的是 test 测试数据集。

第 7 组代码如下：

```
#7
print('#7,分析 hmm 模型状态数据')
mn9=6
ksgn,stalst='sta',list(range(mn9))
#
zt.prDF('df_train',df_train)
kstalst=zdat.df_xp100lst(df_train,ksgn,stalst)
print('kstalst',kstalst)
print(df_train['sta'].value_counts())
```

分析隐马尔可夫模型生成的 sta 状态字段数据，图 9.3 是对应的输出信息。

```
#7,分析hmm模型状态数据

 df_train
              open     high    close      low       volume        amount   xprice      kpr  close_next  sta
date
2010-10-11  2755.03  2823.60  2806.94  2755.03  20753526400  253469818880  2841.41  1012.28           2    2
2010-10-12  2795.76  2843.43  2841.41  2782.34  16742832000  211787464704  2861.36  1007.02           2    2
2010-10-13  2842.14  2862.04  2861.36  2826.22  18528166400  228385390592  2879.64  1006.39           2    2
2010-10-14  2881.55  2919.41  2879.64  2864.52  22946171200  264682455040  2971.16  1031.78           2    2
2010-10-15  2863.74  2971.16  2971.16  2858.22  23656856000  272950771712  2955.23   994.64           1    2
2010-10-18  2985.46  3026.10  2955.23  2944.72  26331694400  301231079424  3001.85  1015.78           2    2
2010-10-19  2949.63  3002.37  3001.85  2931.91  17066782400  199628275712  3003.95  1000.70           0    2
2010-10-20  2947.51  3041.15  3003.95  2942.50  20780441600  262064865280  2983.53   993.20           1    2
2010-10-21  3009.22  3018.24  2983.53  2958.55  16143614400  205073039360  2975.04   997.15           0    2
2010-10-22  2976.75  2997.62  2975.04  2954.62  14707988800  194901557248  3051.42  1025.67           2    2

              open     high    close      low       volume        amount   xprice      kpr  close_next  sta
date
2017-12-18  3268.03  3280.54  3267.92  3254.18  12070038900  149738224054  3296.54  1008.76           2    5
2017-12-19  3266.02  3296.94  3296.54  3266.02  11514013400  150786582711  3287.61   997.29           0    5
2017-12-20  3296.74  3300.21  3287.61  3276.12  13774511800  167961847923  3300.06  1003.79           0    5
2017-12-21  3281.12  3309.22  3300.06  3267.40  14212792700  173740095582  3297.06   999.09           0    5
2017-12-22  3297.68  3307.33  3297.06  3293.44  12404732600  152099443907  3280.46   994.96           1    5
2017-12-25  3296.21  3312.30  3280.46  3270.44  14689363500  177293589884  3306.12  1007.82           2    5
2017-12-26  3277.84  3307.30  3306.12  3274.33  14243450100  174679263836  3275.78   990.82           1    5
2017-12-27  3302.46  3307.08  3275.78  3270.35  16267489000  198264675156  3296.38  1006.29           2    5
2017-12-28  3272.29  3304.10  3296.38  3263.73  17537167000  208019859614  3307.17  1003.27           0    5
2017-12-29  3295.25  3308.22  3307.17  3292.77  14158683600  170356755129  3348.33  1012.44           2    5
```

图 9.3　HMM（隐马尔可夫）模型 sta 字段数据

由图 9.3 可以看出，实盘字段 close_next 和隐马尔可夫模型生成的 sta 状态字段数据并不能够直接匹配。

以下代码：

```
kstalst=zdat.df_xp100lst(df_train,ksgn,stalst)
```

调用 TOP 极宽模块的 df_xp100lst 函数，计算 sta 状态字段各种数值的百分比，具体细节请大家参看 df_xp100lst 函数代码，对应的输出数据是：

```
kstalst [9.19, 27.19, 20.6, 18.22, 5.11, 19.69]
```

这个表示在 sta 字段当中，数值为“0”至“5”，各自所占的百分比。

以下代码：

```
print(df_train['sta'].value_counts())
```

调用 pandas 的 value_counts 统计函数，输出各种 sta 对应的具体数值：

```
1    479
2    363
5    347
3    321
0    162
4     90
Name: sta, dtype: int64
```

第 8 组代码如下：

```
#8
print('\n#8,制作 HMM 模型状态数据和 sta 趋势对照表')
ksgn,klst='close_next',list(range(3))
staLib={}
for xc in range(mn9):
    df3=df_train[df_train.sta==xc]

    dlst=zdat.df_xp100lst(df3,ksgn,klst)
    d9,inx=zt.inxMax(dlst)
    print(xc,'#',inx,d9,'@',dlst)
    #if d9>stak0:
    staLib[xc]=inx
#
```

```
print('\n@staLib',staLib)
```

因为 sta 字段数据和实盘对应的字段 close_next 价格趋势无法直接对比，因此需要建立一个 sta 趋势对照表。

这个 sta 趋势对照表就是变量 staLib，采用的是 Python 的字典格式。

具体转换也是调用 TOP 极宽模块的 df_xp100lst 函数，程序运行结果如下：

```
8,制作 HMM 模型状态数据和预测趋势对照表
0 # 0 53.09 @ [53.09, 22.84, 24.07]
1 # 0 45.3 @ [45.3, 27.14, 27.56]
2 # 0 39.12 @ [39.12, 27.0, 33.88]
3 # 0 41.43 @ [41.43, 30.84, 27.73]
4 # 2 46.67 @ [16.67, 36.67, 46.67]
5 # 0 53.03 @ [53.03, 20.46, 26.51]

@staLib {0: 0, 1: 0, 2: 0, 3: 0, 4: 2, 5: 0}
```

第 9 组代码如下：

```
#9
print('#9,predict 模型预测数据')
df_test['sta_tst']=mx.predict(xtst)
```

输入 test 测试数据集，生成对应的 sta 预测状态数据。

第 10 组代码如下：

```
#10
print('#10,转换 predict 预测数据')
df_test['close_pred']=-1
for xc in stalst:
    xprd=staLib.get(xc)
    if xprd!=None:
        #print(xc,'#',xprd)
        df_test.loc[df_test.sta_tst==xc,'close_pred']=xprd
```

根据要求，生成的 sta 预测状态数据字段通过对照 sta 趋势对照表变量 staLib，转换成实际的预测价格趋势数据字段 close_pred。

第 11 组代码如下：

```
    #11
    print('\n#11,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

对应的输出信息如下：

```
    #11,按 1%精度验证模型
    acc: 41.49%
```

本案例精度为 41.49%。

本案例相对于前面的案例复杂很多，特别是 sta 数据字段和 close_pred 实际预测数据字段的转换，对于初学者来说可能难于理解。

这也是 HMM（隐马尔可夫）模型的学习瓶颈所在，大家可以多运行几次案例，最好采用单步运行模式，同步查看相关的变量数据。

第 10 章

实例算法

实例算法，英文全称是：Instance-based Algorithms，简称 IA。

实例算法模型首先选取一批样本数据，然后根据某些近似性，把新数据与样本数据进行比较，以寻找最佳的匹配。

实例算法是最简单的机器学习算法之一，它不像其他算法，需要在样本的基础上建立一般性的推理公式，而是直接通过输入的数据集进行分类或回归学习，并直接获得最终结果。

因此，实例算法也被称为“赢家通吃的学习”或者“基于记忆的学习”，如图 10.1 所示。

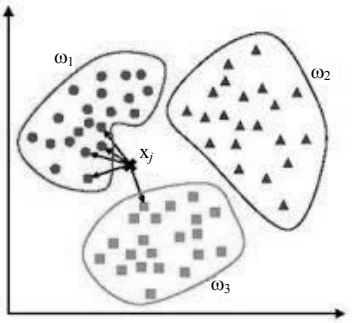


图 10.1　IA（实例算法）

实例算法最后建成的模型对原始数据样本实例有很强的依赖性。实例算法在做预测决策时，一般是在某个范围之内给出相应的预测结果。

实例算法的缺点是硬件成本需求较高。无论是 CPU 运行速度，还是数据存储软件要求都较高。算法占用的空间和运行的深度直接取决于实例数量的大小。

常见的实例算法如下。

- KNN：K 最近邻算法，英文全称是 k-Nearest Neighbor。
- LVQ：矢量学习算法，英文全称是 Learning Vector Quantization。
- SOM：自组织映射算法，英文全称是 Self-Organizing Map。
- LWL：局部加权学习算法，英文全称是 Locally Weighted Learning。

K最近邻算法

K 最近邻算法如图 10.2 所示。

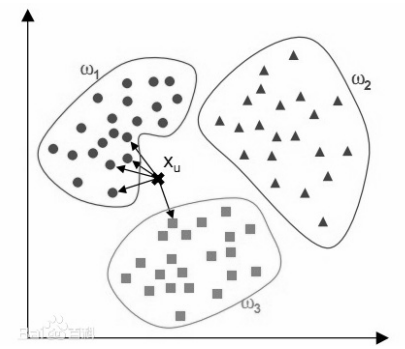


图 10.2　K 最近邻算法

所谓 K 最近邻，就是 k 个最近的邻居的意思，说的是每个样本都可以用它最接近的 k 个邻居来代表。

KNN 算法可用于分类问题和回归问题，该算法更常用于分类问题。

对于回归问题，KNN 算法输出的一般是变量平均值；对于分类问题，

KNN 算法输出的是类别值。

KNN 算法通过周围 k 个案例中的大多数情况划分新的案例。根据一个距离函数，新案例会被分配到它的 k 个近邻中最普遍的类别中去。

KNN 算法是一个相对比较简单的机器学习算法，在理论上也比较成熟。KNN 算法的核心思路是：如果一个样本在特征空间中的 k 个最相似（即特征空间中最邻近）的样本中的大多数属于某一个类别，则该样本也属于这个类别。

KNN 算法的分类是根据一个距离函数进行判别的。这些距离函数可以是欧式距离、曼哈顿距离、明式距离或者是汉明距离。前三个距离函数用于连续函数，第四个函数（汉明函数）则被用于分类变量。

需要注意的是，距离或紧密度的概念在高维度（或者大量的输入变量）中可能会失效，因为输入变量的数量对于算法性能有着很大的负面影响，这就是所谓的维度灾难。

KNN 算法的缺点是需要大量频繁的距离计算，计算成本很高。数据要进行预处理，对异常数据要去噪和清洗，此外还要对数据进行标准化处理。

在 sklearn 模块库当中，K 最近邻算法相关的机器学习算法函数位于 neighbors 模块库，主要如下。

- KNeighborsClassifier，K 最近邻算法。
- NearestNeighbors，最近邻算法。
- KNeighborsRegressor，K 最近邻回归算法。
- NearestCentroid，最近质心算法。
- Locality Sensitive Hashing Forest，局部敏感哈希森林算法。

案例 10-1：第一次惊喜——KNN 算法

前面我们说过，机器学习常见的算法模型大部分都是分类算法，幸运

的是 K 最近邻算法是一种基于数值分析的算法模型，可以用于价格预测。

案例 10-1 的文件名是 kb1001_knn_Reg.py，本案例主要介绍 sklearn 模块库当中 K 最近邻回归函数 KNeighborsRegressor 的具体应用。

案例数据源采用的是 BTC 实盘交易数据，为便于大家以后结合实盘操作，导入数据、训练数据字段是标准的 OHLC 金融数据。

前面几组代码，属于数据准备，和前面的案例 9-2 大同小异，故此处不再重复说明。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = KNeighborsRegressor()
```

调用 K 最近邻回归函数 KNeighborsRegressor 生成模型，并保存在变量 mx 当中。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

调用 fit 函数训练模型。

第 7 组代码如下：

```
#7
print('#7,predict 模型预测数据')
df_test['close_pred']=mx.predict(xtst)
zt.prDF('df',df_test)
```

模型训练完成后，调用模型内置的 predict 预测函数，生成验证数据。

第 8 组代码如下：

```
#8
print('#8,绘制对比曲线图')
df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 9 组代码如下：

```
    #9
    print('#9,验证模型预测效果')
    #9.1
    print('\n#9.1,按 5%精度验证模型')
    dacc,df=zai.ai_acc_xed2x(df_test['close_next'],df_test['close
_pred'],5,True)
    print('acc',dacc)

    #9.2
    print('\n#9.2,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

验证 K 最近邻回归函数的预测结果，以下是对应的输出信息：

```
    #9,验证模型预测效果
    #9.1,按 5%精度验证模型
    acc 55.752

    #9.2,按 1%精度验证模型
    acc: 12.39%
```

运行结果如下。

- 5%的精度下，准确度是 55.75%。
- 1%的精度下，准确度是 12.39%。

以上的准确度数据很差，根本无法用于实盘分析。

我们无须修改代码，只要把第一组代码的数据源文件由 BTC（比特币）改为上证指数数据，再运行一次。

运行结果如下。

- 在 5%的精度下，准确度是 100%。
- 在 1%的精度下，准确度是 68.09%。

这个准确率数据是本书案例当中较好的一个数据，而且没有经过任何优化，在 1%的精度下，准确度达到 68.0%，可能还无法直接用于实盘，不

过作为策略参考还是可以的。

此外，这个模型还可以通过增加数据、优化精度来进一步提高模型的准确率。

这个案例也说明，BTC 等数字货币市场的价格波动远远超出传统的金融市场。一般的日线数据根本无法用于实盘，必须采用分钟级别的分时数据作为数据源才能够获得有效的结果。

案例 10-2：KNN 分类

案例 10-2 的文件名是 kb1002_knn_cla.py，本案例主要介绍 sklearn 模块库当中，K 最近邻分类函数 KNeighborsClassifier 的具体应用。

前面几组代码，属于数据准备，和前面的案例 9-2 大同小异，故此处不再重复说明。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = KNeighborsClassifier()
```

调用 K 最近邻分类函数 KNeighborsClassifier 生成模型，并保存在变量 mx 当中。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

调用 fit 函数，训练模型。

第 7 组代码如下：

```
#7
print('#7,predict 模型预测数据')
df_test['close_pred']=mx.predict(xtst)
zt.prDF('df',df_test)
```

模型训练完成后，调用模型内置的 predict 预测函数，生成验证数据。

第 8 组代码如下：

```
    #8
    print('#8,绘制对比曲线图')
    df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 9 组代码如下：

```
    #9
    print('#9,验证模型预测效果')
    #9.1
    print('\n#9.1,按 5%精度验证模型')
    dacc,df=zai.ai_acc_xed2x(df_test['close_next'],df_test['close
_pred'],5,True)
    print('acc',dacc)

    #9.2
    print('\n#9.2,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

验证 K 最近邻分类算法模型的预测结果，对应的输出信息如下：

```
    #9,验证模型预测效果
    #9.1,按 5%精度验证模型
    acc 39.362

    #9.2,按 1%精度验证模型
    ky0=1; n_df9,94,n_dfk,37
    acc: 39.36%
```

运行结果如下。

- 在 5%的精度下，准确度是 39.36%
- 在 1%的精度下，准确度是 39.36%

以上的准确度数据很差，根本无法用于实盘分析。

第 11 章

正则化算法

正则化算法，英文全称是 Regularization Algorithms。正则化算法是其他算法（比如回归算法）的延伸，根据算法的复杂度对算法进行调整。

正则化算法基于模型复杂度对其进行“奖惩”，它喜欢相对简单的、能有更好泛化能力的模型，通常会对简单模型予以“奖励”，而对复杂模型予以“惩罚”。

正则化算法的优点是其“惩罚”会减少过拟合，可以找到最终结果。正则化算法的缺点是其“惩罚”会造成欠拟合，很难校准模型参数。

在机器学习中，无论是分类算法还是回归算法，都可能存在由于特征过多而导致的过拟合问题，通常的解决办法有两种。

- 减少特征，留取最重要的特征。
- “惩罚”不重要特征的权重。

广告行业有句名言：

我们都知道有 50%的广告预算是浪费的，但问题是，我们不知道是哪 50%。

同样，对于机器学习而言，在通常情况下，大家也不知道：

应该“惩罚”哪些特征的权重取值。

而正则化算法可以防止过拟合，提高泛化能力，解决这方面的问题，如图 11.1 所示。

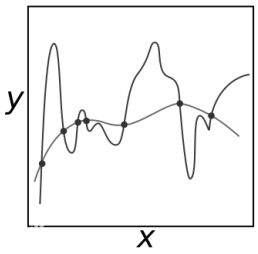


图 11.1　正则化方法

正则化算法可以保留数据原来的特征变量，但是会减小特征变量的数量级。这个方法非常有效，当模型有很多特征变量时，每一个变量都能对最终的预测结果产生一点影响。

正则化项即惩罚函数，该函数对模型向量进行“惩罚”，从而避免单纯最小二乘问题的过拟合问题。

近年来，随着神经网络、深度学习领域的发展，大数据模型的流行，数据的正则化算法的研究也越来越深入。

目前常见的正则化算法如下。

- 岭回归算法，英文全称是 Ridge Regression。
- LASSO 算法，最小收缩选择算法，又名套索回归算法，英文全称是 Least Absolute Shrinkage and Selection Operator。
- EN（弹性网络算法），英文全称是 Elastic Net。
- LAR（最小角回归）算法，英文全称是 Least-Angle Regression。
- L1 和 L2 正则化算法。

11.1　岭回归算法

岭回归算法，英文全称是 Ridge Regression，简称 RR 算法。

岭回归算法本质上是最小二乘法的改进版本，通过损失部分信息、降低精度，使最终结果更加符合实际，是一种更可靠的回归算法，如图 11.2 所示。

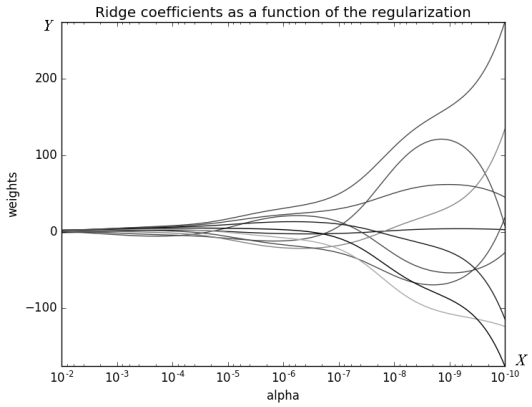


图 11.2　RR（岭回归）算法

以前我们案例的准确度（按 1%精度）最高大约有 68%，接近七成。而 RR 岭回归算法没有经过任何优化，准确度达到 73.40%。

在处理复杂数据的回归问题时，线性回归通常会遇到以下一些问题。

- 预测精度低。
- 当结果数据方差小时，容易产生过拟合。
- 当结果数据方差小时，容易产生无效结果。
- 模型的解释能力低，复杂程度高。

岭回归算法，具有严格的理论基础：

根据高斯-马尔可夫定理，多重相关性并不影响最小二乘法估计量的无偏性和最小方差性。

最小二乘法估计量在所有线性估计量中是方差最小的，但是这个不一定是最优解。

通常，还可以找到其他的有偏估计量，这个估计量虽然有较小的偏差，但它的精度却能够大大高于无偏的估计量。

岭回归算法，就是根据以上原理，通过在正规方程中引入有偏常量以获得最佳结果。

岭回归算法虽然效果不错，不过还是存在以下一些问题。

- 岭参数计算方法太多，差异太大。
- 根据岭迹图进行变量筛选，主观性太大。
- 模型变量太多，难以筛选。

为改进岭回归算法当中的这些问题，有学者又提出了 LASSO（套索）算法、LAR（最小角回归）算法等多种改进版本的算法模型，以优化模型结构，方便筛选自变量。

岭回归算法本质上是一种缩减的算法，通过把一些系数缩减成很小的值，来优化模型结果，类似于降维算法，保留更少的特征，以减少模型的复杂程度，便于理解。

实际研究表明，与简单的线性回归相比，这种缩减算法的确能够取得更好的预测效果。

案例 11-1：新高度——岭回归算法

案例 11-1 的文件名是 kb1101_rr_Reg.py，本案例主要介绍 sklearn 模块库当中，岭回归函数的具体应用。

前面几组代码，属于数据准备，和前面的案例 9-2 大同小异，故此处不再重复说明。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = linear_model.Ridge()
```

调用岭回归函数，生成模型，并保存在变量 mx 当中。

注意，岭回归函数位于 linear_model 线性模型子模块当中，这也间接

说明，岭回归函数其实也是线性模型的衍生算法。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

调用 fit 函数，训练模型。

第 7 组代码如下：

```
#7
print('#7,predict 模型预测数据')
df_test['close_pred']=mx.predict(xtst)
zt.prDF('df',df_test)
```

模型训练完成后，调用模型内置的 predict 预测函数，生成验证数据。

第 8 组代码如下：

```
#8
print('#8,绘制对比曲线图')
df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 9 组代码如下：

```
#9
print('#9,验证模型预测效果')
#9.1
print('\n#9.1,按 5%精度验证模型')
dacc,df=zai.ai_acc_xed2x(df_test['close_next'],df_test['close
_pred'],5,True)
print('acc',dacc)


#9.2
print('\n#9.2,按 1%精度验证模型')
dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

验证岭回归函数的预测结果，以下是对应的输出信息：

```
#9,验证模型预测效果
#9.1,按 5%精度验证模型
```

```
acc 100

#9.2,按 1%精度验证模型
acc: 73.4%
```

运行结果如下。

- 在 5%的精度下，准确度是 100%。
- 在 1%的精度下，准确度是 73.4%。

这是本书案例当中最好的成绩。不过，越是在快成功的时候，越是需要冷静。

我们无须修改代码，只要把第一组代码的数据源文件由上证指数数据改为比特币的实盘交易数据，再运行一次。

运行结果如下：

```
#9,验证模型预测效果
#9.1,按 5%精度验证模型
acc 64.602

#9.2,按 1%精度验证模型
ky0=1; n_df9,113,n_dfk,20
acc: 17.70%;
```

BTC 数据的运行结果如下。

- 在 5%的精度下，准确度是 64.6%。
- 在 1%的精度下，准确度是 17.7%。

由比特币数据的运算结果可以看出，岭回归算法模型还有很大的提升空间，可以通过增加数据、优化精度，进一步提高模型的准确率。

11.2　套索回归算法

套索回归算法，英文全称是 Least Absolute Shrinkage and Selection Operator，如图 11.3 所示。

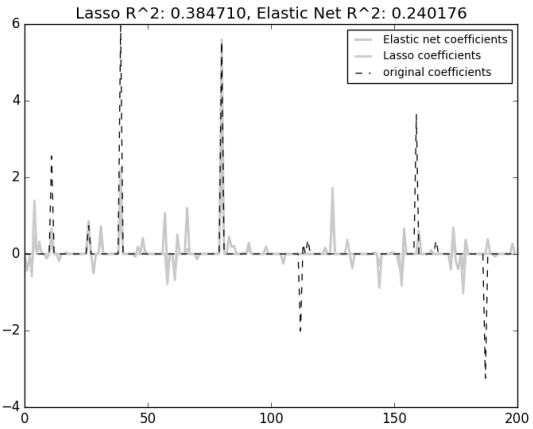


图 11.3　LASSO（套索）回归算法

岭回归算法主观性太大，很多参数的选择都要靠人工，依据经验手动完成，完全不适合大数据和机器学习思想。

为克服这些缺点，有学者对岭回归算法进行了多方面的优化，设计出套索回归算法。

套索回归算法，是 1996 年由斯坦福大学统计学教授 Robert Tibshirani 最早提出的。套索回归算法最初用于计算最小二乘法模型。套索回归算法输入简单，却揭示了很多估计量的重要性质。

- 估计量与岭回归和最佳子集选择的关系。
- 套索回归系数估计值和软阈值之间的联系。
- 当协变量共线时，套索回归系数估计值不一定唯一。

套索回归算法模型简单，便于分析，而且方便筛选参数变量。套索回归算法通过构造一个惩罚函数得到一个较为精炼的模型，压缩一些系数，同时设定一些系数为零。

套索回归算法也叫作线性回归算法的 L1 正则化，和岭回归算法的主要区别就是在正则化项，岭回归算法用的是 L2 正则化，而套索回归算法用的是 L1 正则化。

案例 11-2：套索回归算法应用

案例 11-2 的文件名是 kb1102_lasso_Reg.py，本案例主要介绍 sklearn 模块库当中，套索回归函数的具体应用。

前面几组代码，属于数据准备，和前面的案例 9-2 大同小异，故此处不再重复说明。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = linear_model.Lasso()
```

调用套索回归函数生成模型，并保存在变量 mx 当中。

注意，套索回归函数位于 linear_model（线性模型）子模块当中，这也间接说明，套索回归函数其实也是线性模型的衍生算法。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

调用 fit 函数，训练模型。

第 7 组代码如下：

```
#7
print('#7,predict 模型预测数据')
df_test['close_pred']=mx.predict(xtst)
zt.prDF('df',df_test)
```

图 11.4 是程序的输出信息。

在图 11.4 当中，注意最后两个字段。

- close_next，实盘交易真实数据。
- close_pred，训练后的模型的预测数据。

以上数据都是基于测试数据集，参见变量 df_test。

```
#7,predict模型预测数据

 df
              open     high    close      low       volume        amount   xprice  close_next  close_pred
date
2018-01-02  3314.03  3349.05  3348.33  3314.03  20227886000  227788461113  3369.11     3369.11     3350.27
2018-01-03  3347.74  3379.92  3369.11  3345.29  21383614900  258366523235  3385.71     3385.71     3372.75
2018-01-04  3371.00  3392.83  3385.71  3365.30  20695528800  243090768694  3391.75     3391.75     3385.43
2018-01-05  3386.46  3402.07  3391.75  3380.24  21306068100  248187840542  3409.48     3409.48     3391.21
2018-01-08  3391.55  3412.73  3409.48  3384.56  23616510600  286213219095  3413.90     3413.90     3407.44
2018-01-09  3406.11  3417.23  3413.90  3403.59  19148855100  238249975070  3421.83     3421.83     3411.64
2018-01-10  3414.11  3430.21  3421.83  3398.84  20909499700  254515441261  3425.34     3425.34     3417.78
2018-01-11  3415.58  3426.48  3425.34  3405.64  17381213300  218414134129  3428.94     3428.94     3419.96
2018-01-12  3423.88  3435.42  3428.94  3417.98  17406340400  215961455748  3410.49     3410.49     3426.56
2018-01-15  3428.95  3442.50  3410.49  3402.31  23200928300  286362732919  3436.59     3436.59     3409.48

              open     high    close      low       volume        amount   xprice  close_next  close_pred
date
2018-05-11  3179.80  3180.76  3163.26  3162.21  13065974900  167364290366  3174.03     3174.03     3159.52
2018-05-14  3167.04  3183.82  3174.03  3163.48  12932735300  172410691054  3192.12     3192.12     3175.08
2018-05-15  3180.42  3192.81  3192.12  3164.52  12454905100  162990790010  3169.56     3169.56     3185.66
2018-05-16  3180.23  3191.95  3169.56  3166.81  13052496800  174590979834  3154.28     3154.28     3170.40
2018-05-17  3170.01  3172.77  3154.28  3148.62  11399556700  150598842185  3193.30     3193.30     3149.93
2018-05-18  3151.08  3193.45  3193.30  3144.78  13651691800  168038057477  3213.84     3213.84     3194.75
2018-05-21  3206.18  3219.74  3213.84  3203.34  16445941300  202663464515  3214.35     3214.35     3213.30
2018-05-22  3211.25  3214.59  3214.35  3192.23  14429268400  185721667752  3168.96     3168.96     3205.19
2018-05-23  3205.44  3205.44  3168.96  3168.96  15780764800  199358101015  3154.65     3154.65     3164.64
2018-05-24  3167.94  3173.53  3154.65  3152.07  12408580000  160658185502  3154.65     3154.65     3152.67
```

图 11.4　程序输出信息

第 8 组代码如下：

```
    #8
    print('#8,绘制对比曲线图')
    df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 9 组代码如下：

```
    #9
    print('#9,验证模型预测效果')
    #9.1
    print('\n#9.1,按 5%精度验证模型')
    dacc,df=zai.ai_acc_xed2x(df_test['close_next'],df_test['close
_pred'],5,True)
    print('acc',dacc)


    #9.2
    print('\n#9.2,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

验证套索回归函数的预测结果，以下是对应的输出信息：

```
    #9,验证模型预测效果
    #9.1,按 5%精度验证模型
```

```
acc 100

#9.2,按 1%精度验证模型
acc: 73.4%
```

运行结果如下。

- 在 5%的精度下，准确度是 100%。
- 在 1%的精度下，准确度是 73.4%。

11.3　弹性网络算法

弹性网络算法，英文全称是 Elastic Net，简称 EN 算法或 ENet 算法，如图 11.5 所示。

在使用机器学习方法进行预测时，往往会出现过拟合的情况。模型在训练数据集上的效果很好，但是在测试数据和进行实盘应用时效果很差。

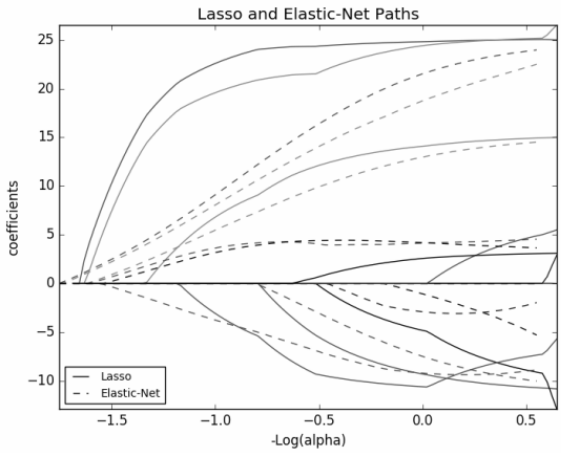


图 11.5　弹性网络算法

在线性回归模型领域，针对这种过拟合情况，产生了许多优化算法，岭回归算法、套索回归算法和弹性网络算法是其中常用的三种优化算法模型。

弹性网络算法是一种同时使用 L1 范数和 L2 范数作为正则化矩阵的线性回归模型。岭回归算法、套索回归算法，可以看作是弹性网络算法的特例。

套索回归算法常用于稀疏数据，而弹性网络算法类似套索回归算法的衍生版本，在套索回归模型上新增了一个 L2 范数正则化的“惩罚”项。当多个特征和另一个特征相关的时候，套索回归算法倾向于随机选择其中一个，而弹性网络算法更倾向于选择两个。

案例 11-3：弹性网络算法应用

案例 11-3 的文件名是 kb1103_enet_Reg.py，本案例主要介绍 sklearn 模块库当中，弹性网络回归函数的具体应用。

前面几组代码，属于数据准备，和前面的案例 9-2 大同小异，故此处不再重复说明。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = linear_model.ElasticNet()
```

调用弹性网络回归函数生成模型，并保存在变量 mx 当中。

注意，弹性网络回归函数位于 linear_model（线性模型）子模块当中，这也间接说明，弹性网络回归回归函数其实也是线性模型的衍生算法。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

调用 fit 函数，训练模型。

第 7 组代码如下：

```
#7
```

```
    print('#7,predict 模型预测数据')
    df_test['close_pred']=mx.predict(xtst)
    zt.prDF('df',df_test)
```

模型训练完成后，调用模型内置的 predict 预测函数，生成验证数据。

第 8 组代码如下：

```
    #8
    print('#8,绘制对比曲线图')
    df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 9 组代码如下：

```
    #9
    print('#9,验证模型预测效果')
    #9.1
    print('\n#9.1,按 5%精度验证模型')
    dacc,df=zai.ai_acc_xed2x(df_test['close_next'],df_test['close
_pred'],5,True)
    print('acc',dacc)

    #9.2
    print('\n#9.2,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

验证弹性网络回归函数的预测结果，以下是对应的输出信息：

```
    #9,验证模型预测效果
    #9.1,按 5%精度验证模型
    acc 100

    #9.2,按 1%精度验证模型
    acc: 73.4%
```

运行结果如下。

- 在 5%的精度下，准确度是 100%。
- 在 1%的精度下，准确度是 73.4%。

11.4 最小角回归算法

最小角回归算法，英文全称是 Least-Angle Regression，简称 LARS 算法。

LARS 算法模型特别适合处理高维、多字段数据集。LARS 算法类似于前向逐步回归。在每一次运算当中，找到与预测值最相关的值。当多个变量具有相同的线性相关时，不是沿着相同的预测变量继续，而是沿着预测变量之间的等角方向前进。

LARS 算法通过相关性选择与残存变化相关的特征。从模型上看，相关性实际上就是特征与残差之间的最小角度，这就是 LARS 名称的由来。

案例 11-4：LARS 算法应用

案例 11-4 的文件名是 kb1104_lars_Reg.py，本案例主要介绍 sklearn 模块库当中，最小角回归函数的具体应用。

前面几组代码，属于数据准备，和前面的案例 9-2 大同小异，故此处不再重复说明。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = linear_model.Lars()
```

调用最小角回归函数，生成模型，并保存在变量 mx 当中。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

调用 fit 函数，训练模型。

第 7 组代码如下：

```
    #7
    print('#7,predict 模型预测数据')
    df_test['close_pred']=mx.predict(xtst)
    zt.prDF('df',df_test)
```

第 8 组代码如下：

```
    #8
    print('#8,绘制对比曲线图')
    df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 9 组代码如下：

```
    #9
    print('#9,验证模型预测效果')
    #9.1
    print('\n#9.1,按 5%精度验证模型')
    dacc,df=zai.ai_acc_xed2x(df_test['close_next'],df_test['close
_pred'],5,True)
    print('acc',dacc)

    #9.2
    print('\n#9.2,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

验证最小角回归函数的预测结果，以下是对应的输出信息：

```
    #9,验证模型预测效果
    #9.1,按 5%精度验证模型
    acc 100

    #9.2,按 1%精度验证模型
    acc: 73.4%
```

运行结果如下。

- 在 5%的精度下，准确度是 100%。
- 在 1%的精度下，准确度是 73.4%。

第 12 章

聚类分析

聚类分析，又称为聚类算法、集簇算法，英文全称是 Clustering Algorithms，简称 CA 算法。

聚类算法通过对一组目标进行分类处理把属于同一组（即一个类）的目标划分在同一组中，与其他组目标相比，同一组目标彼此更加相似。

聚类算法就像回归算法一样，有时候人们描述的是一类问题，有时候描述的是一类算法。

通常，聚类算法首先按照中心点，或者通过分层的方式，对输入数据进行归并，把输入样本聚成围绕一些中心的“数据团”，以找到数据的内在结构，按照最大的共同点将数据进行归类。

聚类算法可以让数据变得更加有意义，不过聚类算法的结果往往难以解读，对于某些特殊数据，结果可能无法使用。

常用的聚类算法如下。

- K 均值算法。
- BIRCH 算法。
- K 最近邻算法。
- EM 算法。

- HC（分层集群）算法。

每种聚类算法，采用的模式都不相同，常用的聚类模式如下。

- 基于类心的聚类模式。
- 基于连接的聚类模式。
- 基于密度的聚类模式。
- 概率型聚类模式。
- 降维聚类模式。
- 神经网络/深度学习聚类模式

12.1　K均值算法

K 均值算法，英文全称是 K-Means Algorithm，也称为 K-Means 算法，简称 K 均值算法。

K 均值算法在本质上是一种聚类算法，把 n 个对象根据它们的属性分为 k 个类型（$k < n$），如图 12.1 所示。

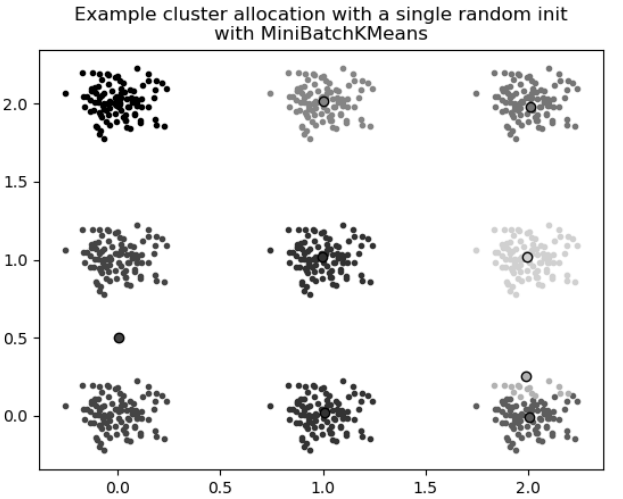


图 12.1　K 均值算法

K 均值算法涉及集群，每个集群有自己的质心。一个集群内的质心和各数据点之间距离的平方和形成了这个集群的平方值之和。同时，当所有集群的平方值之和加起来的时候，就组成了集群方案的平方值之和。

K 均值算法假设对象属性来自空间向量，目标是使各个群组内部的均方误差总和最小。

K 均值算法是一种无监督式学习算法，与处理混合正态分布的最大期望算法很相似，因为它们都试图找到数据中自然聚类的中心。

K 均值算法给每个集群选择 k 个点，这些点称为质心。每一个数据点与距离最近的质心形成一个集群，也就是 k 个集群。

K 均值算法的缺点是对于高维空间、多属性数据集效果不好，需要预先利用 PCA 等算法对数据进行降维。

案例 12-1：K 均值算法应用

案例 12-1 的文件名是 kb1201_kmean_cla.py，本案例采用 K-Means 分类算法，对上证指数的价格趋势进行分析。

分类函数使用的是 KMeans 分类函数。

前面几组代码，属于数据准备，和前面的案例 9-2 大同小异。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx =  KMeans(n_clusters=mn9, random_state=0)
```

调用 KMeans 分类函数生成模型，并保存在变量 mx 当中，调用参数：

```
n_ clusters =3
```

参数表示 KMeans 分类函数生成的类别数目，案例中数值为 3，因为我们的价格趋势采用的是类似足彩博弈的“310”（胜平负）模式，所以有三种类型。

因为股票市场波动很小，我们按收盘价千分之五的上下波动对数据进行分类，分别为“2”“1”“0”，如下所示。

- 2，波动大于 100.5%，up（上涨）模式。
- 1，波动小于 99.5%，down（下跌）模式。
- 0，波动在 0.5%之间，eq（平稳）模式。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

输入训练数据集，对模型进行训练，通过输入数据集，不断调整优化模型的内部参数。

第 7 组代码如下：

```
#7
print('#7,predict 模型预测数据')
df_test['close_pred']=mx.fit_predict(xtst)
```

模型训练完成后，调用模型内置的 fit_predict 预测函数，生成验证数据。

第 8 组代码如下：

```
#8
print('\n#8,按 1%精度验证模型')
dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

验证 KMeans 分类函数的预测结果，对应的输出信息如下：

```
#8,按 1%精度验证模型
ky0=1; n_df9,94,n_dfk,24
acc: 25.53%;  MSE:1.38, MAE:0.96,  RMSE:1.18, r2score:-0.98,
@ky0:1.00
```

运行结果表明，在 1%的精度下，模型准确度是 25.5%。

第 9 组代码如下：

```
#9
print('\n#9,绘制对比数据曲线图')
df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 10 组代码如下：

```
print('\n#10,value_counts')
#
print("\ndf_test['close_next'].value_counts()")
print(df_test['close_next'].value_counts())
print("\ndf_test['close_pred'].value_counts()")
print(df_test['close_pred'].value_counts())
```

本组代码是分类算法的附加代码，调用 pandas 的 value_counts 统计函数，查看 close_next 实盘数据字段、close_pred 预测数据字段的具体数据分布情况。或者说，用于统计实盘数据和预测数据当中数值为 2、1、0，即上涨、下跌、平稳的具体数值，对应的输出信息如下：

```
#10,value_counts
df_test['close_next'].value_counts()
0    39
2    28
1    27
Name: close_next, dtype: int64

df_test['close_pred'].value_counts()
1    45
0    25
2    24
Name: close_pred, dtype: int64
```

12.2 BIRCH算法

除了 K 均值算法外，BIRCH 算法也是一种常用的聚类算法。BIRCH 算法的全称是层次平衡迭代聚类算法，英文全称是 Balanced Iterative

Reducing and Clustering Using Hierarchies，简称 BIRCH 算法，如图 12.2 所示。

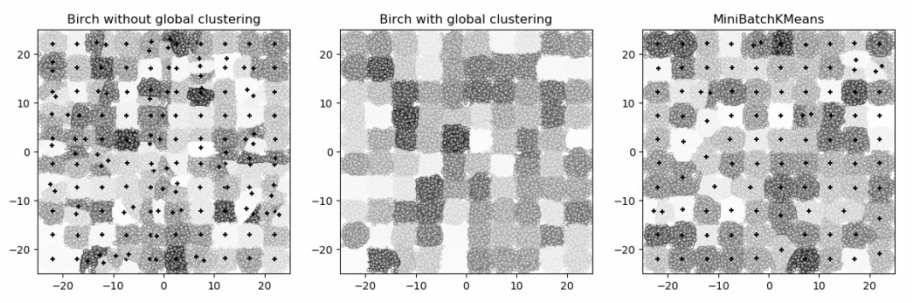


图 12.2　BIRCH 算法

BIRCH 算法运行速度很快，只需要单次扫描数据集就能进行聚类分析，适合于数据量大、类别数也比较多的情况。

BIRCH 算法通过聚类特征树（Clustering Feature Tree，简称 CF 树）来快速进行聚类分析。CF 树结构类似平衡 B+树，每个节点由若干个聚类特征（Clustering Feature，简称 CF）组成。

BIRCH 算法把所有的数据建成 CF 树，对应的输出就是若干个 CF 节点，每个节点里的样本点就是一个聚类的簇。BIRCH 算法的主要过程就是建立 CF 树的过程。

案例 12-2：BIRCH 算法应用

案例 12-2 的文件名是 kb1201_birch_cla.py，本案例介绍使用 BIRCH 分类算法对上证指数的价格趋势进行分析。

分类函数使用的是 BIRCH 分类函数。

前面 4 组代码，属于数据准备，和前面的案例 9-2 大同小异，在此处不再叙述。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx =  Birch(n_clusters=3)
```

调用 BIRCH 分类函数生成模型，并保存在变量 mx 当中。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

输入训练数据集，对模型进行训练，通过输入数据集不断调整优化模型的内部参数。

第 7 组代码如下：

```
#7
print('#7,predict 模型预测数据')
df_test['close_pred']=mx.fit_predict(xtst)
```

模型训练完成后，调用模型内置的 fit_predict 预测函数，生成验证数据。

第 8 组代码如下：

```
#8
print('\n#8,按 1%精度验证模型')
dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

验证 BIRCH 分类函数的预测结果，对应的输出信息如下：

```
#8,按 1%精度验证模型
ky0=1; n_df9,94,n_dfk,32
acc: 34.04%;  MSE:1.04, MAE:0.79,  RMSE:1.02, r2score:-0.49,
@ky0:1.00
```

运行结果表明，在 1%的精度下，模型准确度是 34%。

第 9 组代码如下：

```
#9
print('\n#9,绘制对比数据曲线图')
df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 10 组代码如下：

```
#10
print('\n#10,value_counts')
#
print("\ndf_test['close_next'].value_counts()")
print(df_test['close_next'].value_counts())
print("\ndf_test['close_pred'].value_counts()")
print(df_test['close_pred'].value_counts())
```

这个是分类案例附带的程序，用于统计实盘数据和预测数据当中数值为 2、1、0，即上涨、下跌、平稳的具体数值，对应的输出信息如下：

```
#10,value_counts

df_test['close_next'].value_counts()
0    39
2    28
1    27
Name: close_next, dtype: int64

df_test['close_pred'].value_counts()
1    45
0    26
2    23
Name: close_pred, dtype: int64
```

12.3　小结

本章以及全书的案例重点在于讲解 sklearn 集成的经典机器学习算法的使用方法。除了特定章节外，都舍弃了传统 sklearn 案例当中的数据预处理环节。

直接调用相关的机器学习函数，这样大大简化了案例流程和程序逻辑，便于初学者快速掌握机器学习的核心算法。不过，这种简化流程的代价就是准确度大大降低，使得大部分案例的准确度都低于 50%，这个属于正常

情况。

目前，金融量化交易、大数据分析等领域的 AI（人工智能）应用都是基于深度学习、神经网络来展开的，sklearn 模块库基本上只是作为辅助工具。

但是，对于初学者而言，快速、系统地掌握 sklearn 模块库当中的各种机器学习算法模型是下一步学习深度学习、神经网络的基础。这种节点式的学习方法简单高效。对于个别难于理解的节点，可以结合“黑箱模式”，以后有经验时再补充。

第 13 章

降维算法

降维算法，英文全称是 Dimensionality Reduction Algorithms，简称 DRA 算法。

DRA 算法通过无监督学习的方式，利用较少的信息来归纳或者解释数据。DRA 算法和聚类算法类似，通过分析数据的内在结构，来找到数据间的关联。

DRA 算法常用于高维数据的可视化，或者用来简化数据，以便监督学习使用。

DRA 算法的优点是可以处理大规模数据集，无须在数据上进行假设，缺点是难以分析非线性数据，难以理解结果的意义。

常用的 DRA 算法如下。

- 主成分分析
- 主成分回归
- 偏最小二乘回归
- Sammon 映射
- 多维尺度变换
- 投影寻踪

- 线性判别分析
- 混合判别分析
- 二次判别分析
- 灵活判别分析

13.1 主成分分析

主成分分析算法，英文全称是 Principal Components Analysis，简称 PCA 算法。

PCA 算法属于统计学的方法，通过正交变换将一组可能存在相关性的变量转换为一组线性不相关的变量，转换后的这组变量称为主成分，如图 13.1 所示。

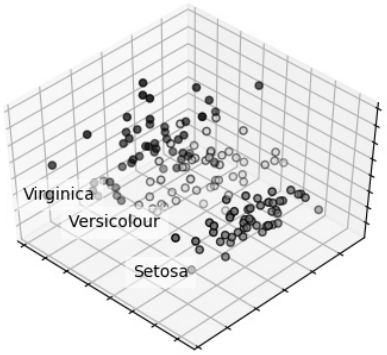


图 13.1 主成分分析

PCA 算法是最重要的降维方法之一，在数据压缩、消除冗余和数据噪声消除等领域都有广泛的应用。一般我们提到降维最容易想到的算法就是 PCA 算法，下面我们就对 PCA 算法的原理做一个总结。

PCA 算法可以用来降低算法计算开销、去除噪声，以及使结果易于展示与理解等。

PCA 算法就是找出数据里最主要的方面，用数据里最主要的方面来代

替原始数据。简单来说，PCA 算法类似于作文里面的主题思想，或者说新闻报道的摘要、大纲，通过更加简短的核心数据，反映整体内容。

PCA 算法的主要应用领域包括数据压缩、简化数据、数据可视化等。

PCA函数接口

sklearn 模块库有个 decomposition 子模块，用于处理降维方面的算法函数。

其中收录的 PCA 算法函数的 API 接口如下：

```
    PCA(n_components=None, copy=True, whiten=False, svd_solver
= 'auto', tol=0.0, iterated_power='auto', random_state=None)
```

相对其他机器学习算法模型而言，PCA 算法函数的 API 接口调用的参数算是非常少的了。

其中主要参数如下。

- n_components，表示保留下来的特征个数 *n*。如果设置为'mle'，将自动选取特征个数 *n*，使得满足所要求的方差百分比。
- copy，默认为 True。表示在运行算法时，是否将原始训练数据复制一份。若为 True，则在运行 PCA 算法后，原始训练数据的值不会有任何改变，因为是在原始数据的副本上进行运算的；若为 False，则在运行 PCA 算法后，原始训练数据的值会改变，因为是在原始数据上进行降维计算的。
- whiten，默认为 False，表示对数据进行白化处理。所谓白化，就是对降维后的数据的每个特征进行归一化，让方差都为 1。对于 PCA 降维本身来说，一般不需要白化处理。
- svd_solver，表示指定奇异值分解（SVD）的方法。由于特征分解是奇异值分解（SVD）的一个特例，一般的 PCA 库都是基于 SVD 实现的。有 4 个可以选择的值：auto、full、arpack、randomized。randomized 一般适用于数据量大、数据维度多同时主成分数目比例又较低的 PCA 降维，它使用了一些加快 SVD 的随机算法。full 则是传统意义上的 SVD，

使用了 scipy 库对应的实现。arpack 和 randomized 的适用场景类似，区别是 randomized 使用的是 scikit-learn 自己的 SVD 实现，而 arpack 直接使用了 scipy 库的 sparse SVD 实现。默认是 auto，即 PCA 算法会自动在前面讲到的三种算法里面去权衡，选择一个合适的 SVD 算法来降维。一般来说，使用默认值就够了。

PCA 函数返回值是一个 PCA 对象，主要包括以下属性。

- components_，返回具有最大方差的成分。
- explained_variance_，降维后的各主成分的方差值。
- explained_variance_ratio_，返回值 *n* 个特征各自的方差百分比比例越大越重要。
- n_components_，返回所保留的特征个数 *n*。
- mean_，返回值的平均值。

案例 13-1：主成分分析的应用

案例 13-1 的文件名是 kb1301_pca.py，本案例主要介绍 sklearn 模块库中主成分分析函数的具体应用。

在以往的机器学习案例中，前面几组代码都是数据准备环节，只有个别细节不同，所以大同小异。

但在进行实盘分析时，往往把前面这些环节压缩到一个 init 函数当中，本书的案例采用的是分解模式。

第 1 组代码如下：

```
#1
print('#1,rd data & init')
np.set_printoptions(suppress=True)
pd.set_option('display.width', 450)
#
mlst=['ma_5','ma_10','ma_15','ma_30']
```

```
clst4=['open','high','low','close']
clst=clst4+['avg']
clst=clst4+mlst
clst=clst4
#
fss='data/sh2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
xdf['avg']=xdf[clst4].mean(axis=1)
xdf=zta.tax_mul(zta.MA,xdf,'close','ma',[5,10,15,30])
zt.prDF('xdf',xdf)
```

第 1 组代码不再只是读取数据，而是增加了一个 init 初始化环节，用于设置运行环境，初始化相关变量参数。

设置运行环境，主要是以下语句：

```
np.set_printoptions(suppress=True)
pd.set_option('display.width', 450)
```

分别设置如下环节。

- 小数显示不采用科学计数法，更加直观。
- pandas 的表格输出宽度设置为 450，以免经常换行。

初始化参数有两段，一段是数字 clst、mlst 列表变量，这一般是标准 OHLC 金融数据字段，用于设置数据字段名称。不过在进行实盘分析时，往往需要加入其他金融指标参数，如 AVG（均值）、MA（均线数据）等。

其中 mlst 对应的 MA 字段名称需要和下面的 tax_mul 函数调用参数匹配。

另外一段是具体数据的设置，重点是以下语句：

```
xdf['avg']=xdf[clst4].mean(axis=1)
xdf=zta.tax_mul(zta.MA,xdf,'close','ma',[5,10,15,30])
```

前面一句是 pandas 的内置 mean 函数，注意 axis=1 参数的使用，采用行模式求均值，一般初学者很容易在这里出问题。

第二句调用 TOP 极宽版的 TA-Lib 金融函数库的 tax_mul 函数，批量生成多组 MA 数据，有关细节，请参看有关代码。

第 2 组代码如下：

```
#2
print('#2,xed data')
xdf['xprice']=xdf['close'].shift(-1)
xdf['close_next']=xdf['xprice']
xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf',xdf)
```

设置相关数据，对应的输出信息如图 13.2 所示。

```
#2,xed data

 xdf
             open    high   close     low     volume      amount     avg    ma_5   ma_10  ma_15  ma_30  xprice  close_next
date
1994-01-03  837.70  840.65  833.90  831.66  101005600  1048326000  835.98     NaN     NaN    NaN    NaN  832.69      832.69
1994-01-04  835.97  836.97  832.69  829.89   65274300   692748000  833.88     NaN     NaN    NaN    NaN  846.98      846.98
1994-01-05  829.30  847.05  846.98  823.10   89412100   975053000  836.61     NaN     NaN    NaN    NaN  869.33      869.33
1994-01-06  850.78  869.33  869.33  850.78  184511700  1970032000  860.06     NaN     NaN    NaN    NaN  879.64      879.64
1994-01-07  875.18  883.99  879.64  873.01  168688400  1752262000  877.96  852.51     NaN    NaN    NaN  900.30      900.30
1994-01-10  891.99  900.30  900.30  889.73  187595100  1896704000  895.58  865.79     NaN    NaN    NaN  891.79      891.79
1994-01-11  903.54  907.09  891.79  884.00  136850000  1590901000  896.60  877.61     NaN    NaN    NaN  888.04      888.04
1994-01-12  891.83  900.23  888.04  885.88  127429900  1306063000  891.50  885.82     NaN    NaN    NaN  897.46      897.46
1994-01-13  889.02  899.14  897.46  888.80   80792100   838602000  893.60  891.45     NaN    NaN    NaN  849.23      849.23
1994-01-14  900.13  900.99  849.23  842.62  242925400  2505558000  873.24  885.36  868.94    NaN    NaN  859.28      859.28

             open    high    close      low       volume        amount      avg     ma_5    ma_10    ma_15    ma_30   xprice  close_next
date
2018-05-11  3179.80  3180.76  3163.26  3162.21  13065974900  167364290366  3171.51  3158.99  3122.53  3115.28  3130.77  3174.03     3174.03
2018-05-14  3167.04  3183.82  3174.03  3163.48  12932735300  172410691054  3172.09  3166.47  3132.43  3119.05  3131.02  3192.12     3192.12
2018-05-15  3180.42  3192.81  3192.12  3164.52  12454905100  162990790010  3182.47  3172.60  3143.42  3127.09  3133.35  3169.56     3169.56
2018-05-16  3180.23  3191.95  3169.56  3166.81  13052496800  174590979834  3177.14  3174.68  3152.26  3133.86  3133.65  3154.28     3154.28
2018-05-17  3170.01  3172.77  3154.28  3148.62  11399556700  150598842185  3161.42  3170.65  3157.60  3135.55  3133.16  3193.30     3193.30
2018-05-18  3151.08  3193.45  3193.30  3144.78  13651691800  168038057477  3170.65  3176.66  3167.83  3140.57  3134.16  3213.84     3213.84
2018-05-21  3206.18  3219.74  3213.84  3203.34  16445941300  202663464515  3210.77  3184.62  3175.55  3149.83  3136.74  3214.35     3214.35
2018-05-22  3211.25  3214.59  3214.35  3192.23  14429268400  185721667752  3208.10  3189.07  3180.83  3158.64  3139.51  3168.96     3168.96
2018-05-23  3205.44  3205.44  3168.96  3168.96  15780764800  199358101015  3187.20  3188.95  3181.81  3164.49  3140.53  3154.65     3154.65
2018-05-24  3167.94  3173.53  3154.65  3152.07  12408580000  160658185502  3162.05  3189.02  3179.84  3168.07  3139.35  3154.65     3154.65
```

图 13.2　初始化数据输出信息

由图 13.2 可以看出，经过初始化处理的数据，增加了 AVG 和多组 MA 数据等字段。

第 3 组代码如下：

```
#3.1
print('#3,cut data')
tim0str,tim9str='2010-10-10','2017-12-31'
df_train=zdat.df_kcut8tim(xdf,'',tim0str,tim9str)
zt.prDF('#3.1 df_train\n',df_train)
#
#3.2
df_test=xdf[xdf.index>'2018'] #.tail(100)
zt.prDF('#3.2 df_test\n',df_test)
```

把原始数据集拆分为 train 训练数据集和 test 测试数据集。

在本书其他案例当中，第 5 组代码一般是调用机器学习的模型函数，而 PCA 等降维算法属于数据预处理环节，程序代码应该位于模型调用函数之前。

为了和本书其他案例的分组匹配，我们对第 4 组 AI 数据处理进行了增强，额外进行了分组，把有关代码尽量插入在第 4 组当中。

第 4.1 组代码如下：

```
#4
print('#4.1,准备 AI 数据')
x=df_train[clst].values
y=df_train['close_next'].values
#
xtst=df_test[clst].values
```

按 sklearn 模块库要求，把数据集从 DataFrame 格式转换为 NumPy 标准的 ndarray 数组格式。

第 4.2 组代码如下：

```
print('#4.2,数据处理 ext')
drx = PCA(n_components=3, copy=True, whiten=False)
drx2 = PCA(n_components=2, copy=True, whiten=False)
drx4 = PCA(n_components=4, copy=True, whiten=False)
```

ext 额外的数据处理代码，调用 PCA 函数，生成对应的变量。调用参数分别是 2、3、4，把数据集压缩为 2、3、4 个字段。

通常，保留的数据字段越多，原始数据损失越少，准确度越高，但在有些情况下，为了达到运算速度和模型可视化的要求，必须牺牲部分数据字段。

这时，就需要通过 PCA 等降维函数进行预先分析，在降维、减少数据字段的同时，尽量避免准确度的降低。

第 4.3 组代码如下：

```
print('#4.3,数据处理 ext#2')
drx.fit(x)
drx2.fit(x)
drx4.fit(x)
```

ext 额外数据代码，调用 PCA 算法对应的变量，用 train 训练数据集的属性数据集 x 进行计算，分析其中的主要成分。

第 4.4 组代码如下：

```
print('#4.4,检查数据处理结果')
print('\npca',drx.explained_variance_ratio_)
#
print('\npca#2',drx2.explained_variance_ratio_)
#
print('\npca#4',drx4.explained_variance_ratio_)
```

ext 额外数据代码，检查 PCA 函数的分析结果，重点是降维后各种数据成分的方差百分比 explained_variance_ratio，比例越高，就越重要。

对应的输出信息是：

```
#4.4,检查数据处理结果
pca [0.9988668  0.00059772 0.0004723 ]
pca#2 [0.9988668  0.00059772]
pca#4 [0.9988668  0.00059772 0.0004723  0.00006319]
```

在案例当中，我们调用参数部分是 3、2、4，表示降维后的数据种类。

从输出信息当中可以看出，第一组数据最重要，与之对应的是 OHLC 数据集当中的开盘价。当然这种结论还需要进一步地分析验证。

第 4.5 组代码如下：

```
print('\n#4.5,对比数据处理结果')
xtst=drx.transform(xtst)
#
print('\n@x.sr',x)
x=drx.transform(x)
print('\n@x.new',x)
```

按训练好的 PCA 模型对原始数据集进行降维处理，注意在进行案例程序运行、实盘分析时，train 训练数据集和 test 测试数据集都要进行转换。

以下是对应的输出信息：

```
#4.5,对比数据处理结果
```

```
@x.sr [[2755.026 2823.597 2755.026 2806.942]
 [2795.764 2843.428 2782.341 2841.407]
 [2842.138 2862.035 2826.221 2861.361]
 ...
 [3302.461 3307.08  3270.349 3275.783]
 [3272.291 3304.096 3263.728 3296.385]
 [3295.246 3308.225 3292.77  3307.172]]

@x.new [[  48.79076279  -21.11189947   29.91820907]
 [ 109.90785612  -20.48587571   23.67350147]
 [ 174.06773699  -14.18073413   -1.76116903]
 ...
 [1055.92769127    8.50289666  -28.7874454 ]
 [1046.39967932  -20.80206711   -7.95282124]
 [1079.61903647  -22.78596205  -27.75809516]]
```

在以上输出信息当中，原始数据@x.sr 对应的 train 训练数据集是标准的 OHLC 金融数据字段，为 4 个数据字段。

PCA 函数调用参数值为 3，降维后的数据字段为 3 个，参见@x.new 的输出信息部分。

案例 13-2：PCA 算法的上证戏法

案例 13-2 的文件名是 kb1302_pca_reg.py，本案例介绍 PCA（主成分分析）函数结合 K 最近邻算法在上证指数预测方面的应用。

本节案例的开头，与前面的案例 13-1 类似。

第 1 组代码如下：

```
#1
print('#1,rd data & init')
np.set_printoptions(suppress=True)
pd.set_option('display.width', 450)
```

```
#
mlst=['ma_5','ma_10','ma_15','ma_30']
clst4=['open','high','low','close']
clst=clst4+['avg']
clst=clst4+mlst
clst=clst4
#
fss='data/sh2018.csv'
xdf=pd.read_csv(fss,index_col=0)
xdf=xdf.sort_index()
xdf['avg']=xdf[clst4].mean(axis=1)
xdf=zta.tax_mul(zta.MA,xdf,'close','ma',[5,10,15,30])
#xdf=zta.tax_mul(zta.MA,xdf,'open','ma',[5,10,15,30])
zt.prDF('xdf',xdf)
```

第 1 组代码不再只是读取数据，而是增加了一个 init 初始化环节，用于设置运行环境，初始化相关变量参数。

第 2 组代码如下：

```
#2
print('#2,xed data')
xdf['xprice']=xdf['close'].shift(-1)
xdf['close_next']=xdf['xprice']
xdf.fillna(method='pad',inplace=True)
zt.prDF('xdf',xdf)
```

设置相关数据。

第 3 组代码如下：

```
#3.1
print('#3,cut data')
tim0str,tim9str='2010-10-10','2017-12-31'
df_train=zdat.df_kcut8tim(xdf,'',tim0str,tim9str)
zt.prDF('#3.1 df_train\n',df_train)
#
#3.2
df_test=xdf[xdf.index>'2018'] #.tail(100)
```

```
zt.prDF('#3.2 df_test\n',df_test)
```

把原始数据集拆分为 train 训练数据集和 test 测试数据集。

第 4 组代码如下：

```
#4
print('#4.1,准备 AI 数据')
x=df_train[clst].values
y=df_train['close_next'].values
#
xtst=df_test[clst].values


print('#4.2,数据处理 ext')
drx = PCA(n_components=3, copy=True, whiten=False)

print('#4.3,数据处理 ext#2')
drx.fit(x)

print('#4.4,检查数据处理结果')
print('\npca',drx.explained_variance_ratio_)
#
print('\n#4.5,对比数据处理结果')
x=drx.transform(x)
xtst=drx.transform(xtst)
```

准备 AI 机器学习算法所需的数据，并调用 PCA 函数进行降维处理。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = KNeighborsRegressor()
```

调用 K 最近邻回归函数 KNeighborsRegressor 生成模型，并保存在变量 mx 当中。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
```

```
    mx.fit(x,y)
```

按惯例，调用 fit 函数，训练模型。

第 7 组代码如下：

```
    #7
    print('#7,predict 模型预测数据')
    df_test['close_pred']=mx.predict(xtst)
    zt.prDF('df',df_test)
```

当模型训练完成后，调用模型内置的 predict 预测函数，生成验证数据。

第 8 组代码如下：

```
    #8
    print('#8,绘制对比数据图')
    df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 9 组代码如下：

```
    #9
    print('#9,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

验证 K 最近邻回归函数的预测结果，以下是对应的输出信息：

```
    #9,按 1%精度验证模型
    acc: 67.2%
```

在调用 PCA 函数进行降维处理时，n_components 调用参数不同的数值表示不同保留维数，对模型最终结果会有所影响。

- *n*=2，准确度是 63.83%。
- *n*=3，准确度是 67.02%。
- *n*=4，准确度是 68.09%。

当数据源采用标准的 OHLC 格式时，输入数据属性字段是 4 个字段，当 *n*=4 时，其实并没有进行降维处理，所以最终的准确度结果是 68.09%，这与直接调用 K 最近邻回归函数完全一样。

13.2　奇异值分解算法

主成分分析算法其实就是一种特殊情况下的奇异值分解算法。

奇异值分解算法，英文全称是 Singular Value Decomposition，简称 SVD 算法。

奇异值分解是线性代数中一种重要的矩阵分解方法，如图 13.3 所示。

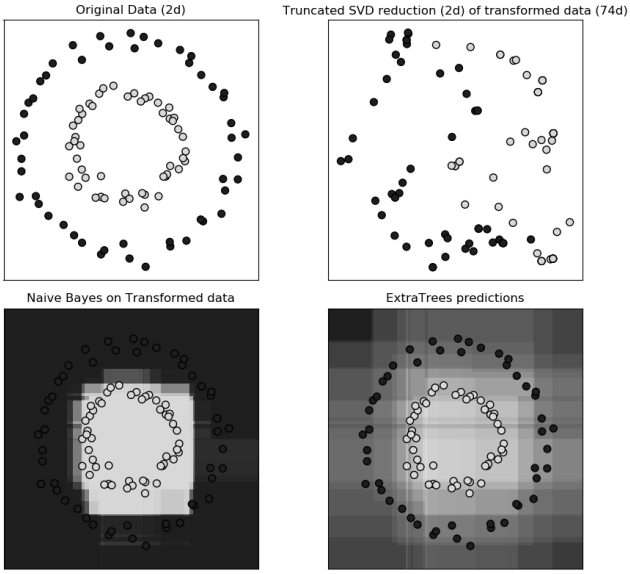


图 13.3　SVD（奇异值分解）算法

SVD函数接口

sklearn 模块库中的 decomposition 子模块用于处理降维方面的算法函数。其中收录的奇异值分解函数的 API 接口如下：

```
    TruncatedSVD(n_components=2, algorithm='randomized', n_iter=5,
random_state=None, tol=0.0)
```

SVD 函数的 API 接口的主要参数如下。

- n_components，保留下来的特征个数 n。如果设置为'mle'，将自动选取特征个数 n，使得满足所要求的方差百分比。

- algorithm，函数算法模式，默认是 randomized 模式，还有一种是 arpack 模式。

案例 13-3：奇异果传说：SVD

看到 SVD 算法第一眼就联想到奇异果。

案例 13-3 的文件名是 kb1303_sva_reg.py，本案例主要介绍 sklearn 模块库当中 TruncatedSVD（奇异值分解）函数与结合 K 最近邻算法在上证指数预测方面的应用。

本案例与 PCA 算法案例基本相同，只是下面这条代码有所不同：

```
print('#4.2,数据处理 ext')
drx = TruncatedSVD(3)
```

在程序的 4.2 组代码中，本案例调用 TruncatedSVD 函数对数据进行降维处理。

案例的运行结果如下：

```
#9,按 1%精度验证模型
acc: 67.02%;
```

使用 TruncatedSVD 函数，对数据集进行降维处理后，使用 K 最近邻回归函数的预测结果，准确度是 67.02%。

第 14 章

集成算法

集成算法，也称为模型融合算法，英文全称是 Ensemble Algorithms，简称 EA 算法。

EA 算法功能非常强大，也非常流行，是当前机器学习四大主流课题之一，也是行业的研究热点。

EA 算法通过使用一些相对较弱的学习模型，彼此独立地对同样的样本进行训练，然后把结果整合起来进行整体预测，以提高系统的最终效果。EA 算法由于采用了投票平均的方法组合多个分类器，可以减少单个分类器的误差，从而提高算法模型的准确度。

EA 算法消除了偏置的影响，减小了预测的方差：多个模型聚合后的预测结果比单一模型的预测结果更稳定。

在金融界，这被称为多样化：多个股票的混合波动总是远小于单个股票的波动。

EA 算法不容易产生过拟合：如果单个模型会产生过拟合，那么将每个模型的预测结果简单地组合（取均值、加权平均、逻辑回归），就可以轻松解决过拟合问题。

EA 算法通常分为两大类型：

- 平均法，Averaging Methods。通过构建几个独立的学习器，然后把它们的预测值平均化。一般而言，组合的学习器要比任何一个单个的学习器好，因为它降低了方差。代表算法有 Bagging（装袋）算法、RF（随机森林）算法。
- 迭代法，Boosting Methods。通过迭代逐步构建学习器，以降低组合学习器的偏差，类似目前神经网络常用的 BP 反向传播迭代模式。代表算法有 AdaBoost 迭代算法、迭代决策树算法。

也有学者又提出了一种新的分类：stacking 模式。

将训练好的所有基模型对训练集进行预测，第 j 个基模型对第 i 个训练样本的预测值将作为新的训练集中第 i 个样本的第 j 个特征值，最后基于新的训练集进行训练。

同理，预测的过程也要先经过所有基模型的预测形成新的测试集，最后再对测试集进行预测。

集成算法的主要难点如下。

- 究竟集成哪些独立的、较弱的学习模型。
- 如何把学习结果整合起来。
- 需要大量的维护工作。
- 各种模型之间的权重参数优化。

本质上，EA 算法不算是一种机器学习算法，而是一种优化手段。通过结合多个简单的弱机器学习算法去做更可靠的决策。比如单个分类器的分类是不可靠的，但是如果使用多个分类器投票，可靠度就会高很多。

EA 算法类似在传统政府和议会当中，就各种问题举行的代表投票，单个代表就相当于一种学习算法，整个议会就是一个 EA 算法模型。

常见的 EA 算法如下。

- Boosting 迭代提升算法。
- Bagging（装袋）算法。

- AdaBoost 迭代算法。
- RF（随机森林）算法。
- Blending 层叠泛化算法。
- GBM（梯度推进机）。
- GBRT 算法。

决策树算法大部分也都是 EA 算法。

14.1　sklearn内置集成算法

sklearn 机器学习模块库内置的 EA 算法基本都收录在 Ensemble 算法子模块中，主要算法模型如下。

- AdaBoostClassifier，AdaBoost 迭代分类算法。
- AdaBoostRegressor，AdaBoost 迭代回归算法。
- BaggingClassifier，Bagging（装袋）分类算法。
- BaggingRegressor，Bagging（装袋）回归算法。
- ExtraTreesClassifier，极端树分类算法。
- ExtraTreesRegressor，极端树回归算法。
- GradientBoostingClassifier，梯度迭代分类算法。
- GradientBoostingRegressor，梯度迭代回归算法。
- IsolationForest，孤立森林（异常监测）算法。
- RandomForestClassifier，RF（随机森林）分类算法。
- RandomForestRegressor，RF（随机森林）回归算法。
- RandomTreesEmbedding，RF（随机森林）集成算法。
- VotingClassifier，投票分类算法。

14.2 装袋算法

装袋算法是一种根据均匀概率分布从数据中重复抽样的技术。

装袋算法也叫自助聚集算法，与随机森林算法类似，通常用于决策树模型。

装袋算法的基本建模思路一般分为三个步骤。

- 步骤一，把数据集分为 *n* 个小组，类似把一大堆水果装到 *n* 个袋子里面，这也是装袋算法的一种形象的比喻。
- 步骤二，对各小组的数据进行分析，找出每组的最佳答案。相当于在每个袋子里挑选最符合要求的水果。注意，不是最大的水果，是最符合要求的水果。
- 步骤三，把 *n* 个小组挑选出的结果进行汇总，再挑选出现频率最高的结果作为最终答案，类似统计学的众数模式。对于水果模式而言，如果按大小作为标准，不是挑选最大或最小的水果，而是按大小分配后出现最多的一类水果。

案例 14-1：装袋回归算法

案例 14-1 的文件名是 kb1401_bag_Reg.py，本案例主要介绍 sklearn 模块库中 BaggingRegressor 函数的具体应用。

前面几组代码属于数据准备，和前面的案例 9-2 大同小异，故此不再重复说明。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = ensemble.BaggingRegressor()
```

调用 BaggingRegressor 函数，生成模型，并保存在变量 mx 当中。

第 6 组代码如下：

```
    #6
    print('#6,fit 训练模型')
    mx.fit(x,y)
```

按惯例，调用 fit 函数，训练模型。

第 7 组代码如下：

```
    #7
    print('#7,predict 模型预测数据')
    df_test['close_pred']=mx.predict(xtst)
    zt.prDF('df',df_test)
```

当模型训练完成后，调用模型内置的 predict 预测函数，使用测试数据集 df_test，生成验证数据。

第 8 组代码如下：

```
    #8
    print('#8,绘制对比数据图')
    df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 9 组代码如下：

```
    #9
    print('#9,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

验证 BaggingRegressor 函数的预测结果，以下是对应的输出信息：

```
    #9,按 1%精度验证模型
    ky0=1; n_df9,94,n_dfk,56
    acc: 59.57%;  MSE:2047.02, MAE:33.18,  RMSE:45.24, r2score:0.89,
@ky0:1.00
```

运行结果表明在 1%的精度下，准确度是 59.57%。

案例 14-2：装袋分类算法

案例 14-2 的文件名是 kb1402_bag_cla.py，本案例主要介绍 sklearn 模块库中 BaggingClassifier 函数的具体应用。

前面几组代码属于数据准备，和前面的案例 9-2 大同小异，故此不再重复说明。

因为是分类算法，所以数据准备阶段采用的是分类模式，大家注意相关代码。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = ensemble.BaggingClassifier()
```

调用 BaggingClassifier 算法函数，生成模型，并保存在变量 mx 当中。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

按惯例，调用 fit 函数，训练模型。

第 7 组代码如下：

```
#7
print('#7,predict 模型预测数据')
df_test['close_pred']=mx.predict(xtst)
zt.prDF('df',df_test)
```

当模型训练完成后，调用模型内置的 predict 预测函数，生成验证数据。

第 8 组代码如下：

```
#8
print('#8,绘制对比数据图')
```

```
    df_test[['close_next','close_pred']].plot(linewidth=3)
    #print(predicted)
```

第 9 组代码如下：

```
    #9
    print('#9,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

验证 BaggingClassifier（装袋分类）算法模型的预测结果，对应的输出信息如下：

```
    #9,按 1%精度验证模型
    ky0=1; n_df9,113,n_dfk,56
    acc: 49.56%
```

运行结果表明在 1%的精度下，准确度是 49.56%。

第 10 组代码如下：

```
    #10
    print('\n#10,value_counts')
    print("\ndf_test['close_next'].value_counts()")
    print(df_test['close_next'].value_counts())
    print("\ndf_test['close_pred'].value_counts()")
    print(df_test['close_pred'].value_counts())
```

本组代码是分类算法的附加代码，调用 pandas 的 value_counts 统计函数，查看 close_next（实盘数据字段）、close_pred（预测数据字段）的具体数据分布情况。

对应的输出信息是：

```
    #10,value_counts

    df_test['close_next'].value_counts()
    2    54
    1    51
    0     8
    Name: close_next, dtype: int64
```

```
df_test['close_pred'].value_counts()
2    97
1    15
0     1
Name: close_pred, dtype: int64
```

以上数据是采用 BTC（比特币）的测试数据，如果我们改用上证指数作为数据源，则在第一组代码当中修改数据源文件：

```
fss='data/btc2018.csv'
xdf=pd.read_csv(fss,index_col=0)
```

运行结果表明在 1%的精度下，准确度是 38.3%。

本案例是少有的 BTC（比特币）数据准确度高于上证指数的案例，其内在原因大家慢慢研究。

此外，从第 10 组附加代码的输出信息中也可以看出，BaggingClassifier（装袋分类）算法模型的 close_pred（预测数据字段）的数据分布较为分散，更加客观。

14.3 AdaBoost迭代算法

AdaBoost 算法是一种迭代算法，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），如图 14.1 所示。

在针对二元分类所开发的 Boosting 迭代算法中，AdaBoost 迭代算法是第一个成功的算法。

在金融量化分析领域，往往需要有很多数据对未来的走势进行预测，这种基于大数据的高预测分析，经常会使用到 GBM 和 AdaBoost 两种 Boosting 算法。

Boosting 算法是一种集成学习算法，它结合了建立在多个基础估计值上的预测结果，以此来提高单个估计值的可靠程度。Boosting 算法通过给训练数据集重新分配权重来优化最终的计算结果。

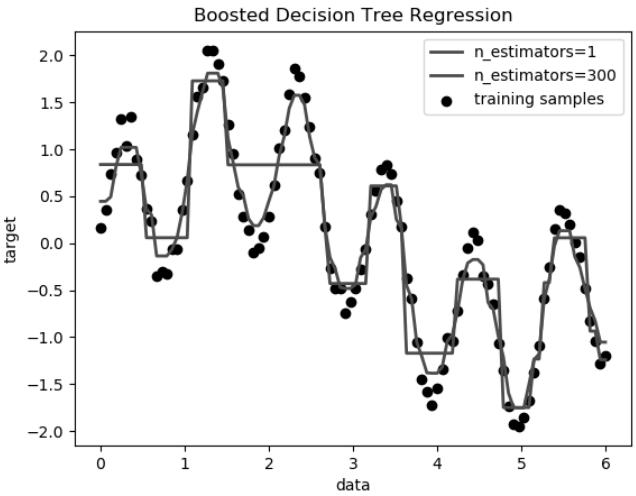


图 14.1　AdaBoost 迭代算法

AdaBoost 算法本身是通过改变数据分布来实现的，它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。

AdaBoost 算法将修改过权值的新数据集再发送给下层分类器进行训练，最后将每次训练得到的分类器融合起来，作为最后的决策分类器。

通常，AdaBoost 算法与决策树一起工作。在第一个决策树创建后，决策树在每个训练实例上的性能都被用来衡量下一个决策树针对该实例所应分配的权重。难以预测的训练数据被赋予更大的权重，而容易预测的数据则被赋予更小的权重。模型依次被创建，每次更新训练实例的权重，都会影响到序列中下一个决策树的学习性能。在所有决策树完成后，即可对新输入的数据进行预测，而每个决策树的性能将由它在训练数据上的准确度来决定。

案例 14-3：AdaBoost 迭代回归算法

案例 14-3 的文件名是 kb1403_ada_Reg.py，本案例主要介绍 sklearn 模

块库中 AdaBoostRegressor 函数的具体应用。

前面几组代码属于数据准备，和前面的案例 9-2 大同小异，故此不再重复说明。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = ensemble.AdaBoostRegressor()
```

调用 AdaBoostRegressor 函数生成模型，并保存在变量 mx 当中。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

按惯例，调用 fit 函数，训练模型。

第 7 组代码如下：

```
#7
print('#7,predict 模型预测数据')
df_test['close_pred']=mx.predict(xtst)
zt.prDF('df',df_test)
```

当模型训练完成后，调用模型内置的 predict 预测函数，使用测试数据集 df_test，生成验证数据 close_pred。

第 8 组代码如下：

```
#8
print('#8,绘制对比数据图')
df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 9 组代码如下：

```
#9
print('#9,按 1%精度验证模型')
dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

验证 AdaBoostRegressor 函数的预测结果，以下是对应的输出信息：

```
#9,按 1%精度验证模型
ky0=1; n_df9,94,n_dfk,56
acc: 51.06%
```

运行结果表明在 1%的精度下，准确度是 51.06%。

案例 14-4：AdaBoost 迭代分类算法

案例 14-4 的文件名是 kb1404_ada_cla.py，本案例主要介绍 sklearn 模块库中 AdaBoostClassifier 函数的具体应用。

前面几组代码属于数据准备，和前面的案例 9-2 大同小异，故此不再重复说明。

因为是分类算法，所以数据准备阶段采用的是分类模式，大家注意相关代码。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = ensemble.AdaBoostClassifier()
```

调用 AdaBoostClassifier 算法函数生成模型，并保存在变量 mx 中。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

按惯例，调用 fit 函数，训练模型。

第 7 组代码如下：

```
#7
print('#7,predict 模型预测数据')
df_test['close_pred']=mx.predict(xtst)
zt.prDF('df',df_test)
```

当模型训练完成后，调用模型内置的 predict 预测函数，生成验证数据。

第 8 组代码如下：

```
    #8
    print('#8,绘制对比数据图')
    df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 9 组代码如下：

```
    #9
    print('#9,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

验证 AdaBoostClassifier 算法模型的预测结果，对应的输出信息如下：

```
    #9,按 1%精度验证模型
    ky0=1; n_df9,113,n_dfk,53
    acc: 46.7%
```

运行结果表明在 1%的精度下，准确度是 46.7%。

第 10 组代码如下：

```
    #10
    print('\n#10,value_counts')
    print("\ndf_test['close_next'].value_counts()")
    print(df_test['close_next'].value_counts())
    print("\ndf_test['close_pred'].value_counts()")
    print(df_test['close_pred'].value_counts())
```

本组代码是分类算法的附加代码，调用 pandas 的 value_counts 统计函数，查看 close_next（实盘数据字段）、close_pred（预测数据字段）的具体数据分布情况。

对应的输出信息是：

```
    #10,value_counts


    df_test['close_next'].value_counts()
    2    54
```

```
1    51
0     8
Name: close_next, dtype: int64

df_test['close_pred'].value_counts()
2    105
1      6
0      2
Name: close_pred, dtype: int64
```

第 15 章

支持向量机

15.1 支持向量机算法

支持向量机算法是神经网络、深度学习复兴以前最流行的机器学习算法。

支持向量机，英文全称是 Support Vector Machine，简称 SVM 算法，如图 15.1 所示。

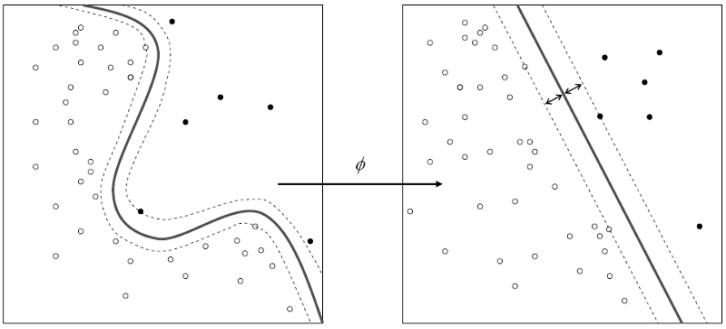


图 15.1　SVM 算法

SVM 算法设计思想是：对于二元可分的模式，存在线性可分离模式的最佳超平面。

对于不可二元分类（线性分离）的数据，SVM 算法使用内核函数将原始数据转换为新空间。

SVM 算法基于超平面使分离超平面（Hyperplane）的边界最大化，可以轻松地对数据群进行分类。SVM 算法在高维空间中表现良好，如果维度大于取样的数量，SVM 算法仍然有效。目前 SVM 模型主要用于二进制分类。

SVM 算法的优点如下。

- 在非线性可分问题上表现优秀。
- 擅长在变量 X 与其他变量之间进行二元分类操作，无论其关系是否是线性的。

SVM 算法的缺点如下。

- 非常难以训练。
- 很难理解与解释。

支持向量机算法将训练事例表示为空间中的点，它们被映射到一幅图中，由一条明确的、尽可能宽的间隔线分开，以区分两个类别。

超平面是输入变量空间内的一条分割线。在支持向量机中，超平面可以通过类别（0 类或 1 类）最佳分割输入变量空间。

超平面与最近的数据点之间的距离被称为边距，在分离两个类上具有最大边距的超平面被称为最佳超平面。超平面的确定只与这些点及分类器的构造有关。这些点被称为支持向量，它们支持并定义超平面。

平行超平面间的距离或差距越大，分类器的总误差越小。

SVM 算法在本质上是一种二分类算法。在 N 维空间中给定两类点，支持向量机生成一个（N–1）维的超平面将这些点分为两类。比如在纸上有两类线性可分的点，支持向量机会寻找一条直线将这两类点区分开来，并且与这些点的距离都尽可能远。

支持向量机算法是一种监督式学习的方法，主要用于以下领域：模式

识别、统计分类、回归分析、展示广告分析、人体结合部位识别、基于图像的性别检查、大规模图像分类等。

本质上，SVM 算法也是一种基于核函数的机器学习算法。

基于核函数的算法思路，是把输入数据映射到一个高阶的向量空间，在这些高阶向量空间里，有些分类或者回归问题能够更容易地解决。

其他常见的基于核函数的算法如下。

- RBG（径向基函数）算法，Radial Basis Function。
- LDA（线性判别分析）算法，Linear Discriminate Analysis。

15.2 SVM函数接口

在 sklearn 模块库中，有独立的 SVM 模块 sklearn.svm，其中相关的机器学习算法函数如下。

- SVC，C-支持向量机算法。
- LinearSVC，线性向量分类算法。
- LinearSVR，线性向量回归算法。
- NuSVC，Nu 支持向量算法。
- SVR，SVR（TEpsilon）支持向量算法。
- NuSVR，Nu 支持 SVR 向量算法。
- OneClassSVM，无监督离群点检测算法。
- l1_min_c，辅助函数，返回边界参数。

其中与本章案例相关的两个函数如下。

- LinearSVR，线性向量回归算法。
- LinearSVC，线性向量分类算法。

LinearSVR（线性向量回归算法），函数接口定义如下：

```
    svm.LinearSVR(epsilon=0.0, tol=0.0001, C=1.0,
loss='epsilon_insensitive', fit_intercept=True,
intercept_scaling=1.0, dual=True, verbose=0, random_state=None,
max_iter=1000)
```

LinearSVC（线性向量分类算法），函数接口定义如下：

```
    svm.LinearSVC(penalty='l2', loss='squared_hinge', dual=True,
tol=0.0001, C=1.0, multi_class='ovr', fit_intercept=True,
intercept_scaling=1, class_weight=None, verbose=0,
random_state=None, max_iter=1000)
```

两个函数的主要调用参数如下。

- C，惩罚系数 C，默认为 1，一般需要通过交叉验证来选择一个合适的 C。一般来说，如果噪声点较多时，C 需要小一些。
- penalty，正则化参数，仅用于线性拟合，可以选择‘l1’即 L1 正则化或者‘l2’即 L2 正则化，默认是 L2 正则化。
- epsilon，距离误差。
- loss，损失函数度量，默认值是 epsilon_insensitive，可选值是 squared_epsilon_insensitive。

案例 15-1：SVM 回归算法

案例 15-1 的文件名是 kb1501_svm_Reg.py，本案例主要介绍 sklearn 模块库中 LinearSVR 函数的具体应用。

前面几组代码属于数据准备，和前面的案例 9-2 大同小异，故此不再重复说明。

第 5 组代码如下：

```
    #5
    print('#5,模型设置')
    mx =svm.LinearSVR()
```

调用 LinearSVR 函数生成模型，并保存在变量 mx 当中。

在 sklearn 模块库中，SVM 算法相关的函数全部收录在 SVM 子模块中。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

按惯例，调用 fit 函数，训练模型。

第 7 组代码如下：

```
#7
print('#7,predict 模型预测数据')
df_test['close_pred']=mx.predict(xtst)
zt.prDF('df',df_test)
```

当模型训练完成后，调用模型内置的 predict 预测函数，使用测试数据集 df_test 生成验证数据，对应的输出信息如图 15.2 所示。

```
#7,predict模型预测数据

 df
              open     high    close      low       volume        amount   xprice  close_next  close_pred
date
2018-01-02  3314.03  3349.05  3348.33  3314.03  20227886000  227788461113  3369.11     3369.11     3350.04
2018-01-03  3347.74  3379.92  3369.11  3345.29  21383614900  258366523235  3385.71     3385.71     3371.20
2018-01-04  3371.00  3392.83  3385.71  3365.30  20695528800  243090768694  3391.75     3391.75     3385.89
2018-01-05  3386.46  3402.07  3391.75  3380.24  21306068100  248187840542  3409.48     3409.48     3391.49
2018-01-08  3391.55  3412.73  3409.48  3384.56  23616510600  286213219095  3413.90     3413.90     3409.04
2018-01-09  3406.11  3417.23  3413.90  3403.59  19148855100  238249975070  3421.83     3421.83     3412.85
2018-01-10  3414.11  3430.21  3421.83  3398.84  20909499700  254515441261  3425.34     3425.34     3420.37
2018-01-11  3415.58  3426.48  3425.34  3405.64  17381213300  218414134129  3428.94     3428.94     3423.19
2018-01-12  3423.88  3435.42  3428.94  3417.98  17406340400  215961455748  3410.49     3410.49     3427.85
2018-01-15  3428.95  3442.50  3410.49  3402.31  23200928300  286362732919  3436.59     3436.59     3409.79

              open     high    close      low       volume        amount   xprice  close_next  close_pred
date
2018-05-11  3179.80  3180.76  3163.26  3162.21  13065974900  167364290366  3174.03     3174.03     3160.92
2018-05-14  3167.04  3183.82  3174.03  3163.48  12932735300  172410691054  3192.12     3192.12     3174.30
2018-05-15  3180.42  3192.81  3192.12  3164.52  12454905100  162990790010  3169.56     3169.56     3189.56
2018-05-16  3180.23  3191.95  3169.56  3166.81  13052496800  174590979834  3154.28     3154.28     3169.39
2018-05-17  3170.01  3172.77  3154.28  3148.62  11399556700  150598842185  3193.30     3193.30     3151.81
2018-05-18  3151.08  3193.45  3193.30  3144.78  13651691800  168038057477  3213.84     3213.84     3195.17
2018-05-21  3206.18  3219.74  3213.84  3203.34  16445941300  202663464515  3214.35     3214.35     3213.42
2018-05-22  3211.25  3214.59  3214.35  3192.23  14429268400  185721667752  3168.96     3168.96     3210.36
2018-05-23  3205.44  3205.44  3168.96  3168.96  15780764800  199358101015  3154.65     3154.65     3166.22
2018-05-24  3167.94  3173.53  3154.65  3152.07  12408580000  160658185502  3154.65     3154.65     3153.16
```

图 15.2　输出数据

第 8 组代码如下：

```
#8
print('#8,绘制对比数据图')
df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 9 组代码如下：

```
#9
print('#9,按 1%精度验证模型')
```

```
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

验证 LinearSVR 函数的预测结果，以下是对应的输出信息：

```
    #9,按 1%精度验证模型
    ky0=1; n_df9,94,n_dfk,68
    acc: 72.34%
```

需要注意的是，SVM（支持向量机）算法类似神经网络算法，内部采用了随机数作为种子，因此每次运行的结果都不一定相同。

以下是第二次运行的结果信息：

```
    #9,按 1%精度验证模型
    ky0=1; n_df9,94,n_dfk,55
    acc: 58.51%;
```

以下是第三次运行结果信息：

```
    #9,按 1%精度验证模型
    ky0=1; n_df9,94,n_dfk,55
    acc: 58.51%
```

案例 15-2：SVM 分类算法

案例 15-2 的文件名是 kb1502_svm_cla.py，本案例主要介绍 sklearn 模块库中 LinearSVC 函数的具体应用。

前面几组代码属于数据准备，和前面的案例 9-2 大同小异，故此不再重复说明。

因为是分类算法，所以数据准备阶段采用的是分类模式，大家注意相关代码。

第 5 组代码如下：

```
    #5
    print('#5,模型设置')
    mx =svm.LinearSVC()
```

调用 LinearSVC 函数生成模型，并保存在变量 mx 当中。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

按惯例，调用 fit 函数，训练模型。

第 7 组代码如下：

```
#7
print('#7,predict 模型预测数据')
df_test['close_pred']=mx.predict(xtst)
zt.prDF('df',df_test)
```

当模型训练完成后，调用模型内置的 predict 预测函数，生成验证数据。

第 8 组代码如下：

```
#8
print('#8,绘制对比数据图')
df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 9 组代码如下：

```
#9
print('#9,按 1%精度验证模型')
dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

对应的输出信息如下：

```
#9,按 1%精度验证模型
ky0=1; n_df9,94,n_dfk,45
acc: 47.87%
```

运行结果表明在 1%的精度下，准确度是 47.89%。

第 10 组代码如下：

```
#10
print('\n#10,value_counts')
print("\ndf_test['close_next'].value_counts()")
print(df_test['close_next'].value_counts())
```

```
print("\ndf_test['close_pred'].value_counts()")
print(df_test['close_pred'].value_counts())
```

本组代码是分类算法的附加代码，调用 pandas 的 value_counts 函数，查看 close_next（实盘数据字段）、close_pred（预测数据字段）的具体数据分布情况。

对应的输出信息是：

```
#10,value_counts

df_test['close_next'].value_counts()
0    39
2    28
1    27
Name: close_next, dtype: int64

df_test['close_pred'].value_counts()
0    67
1    27
Name: close_pred, dtype: int64
```

SVM（支持向量机）算法类似神经网络算法，内部采用了随机数作为种子，每次运行的结果都不一定相同。

以下是第二次运行的结果信息：

```
#9,按 1%精度验证模型
ky0=1; n_df9,94,n_dfk,28
acc: 29.79%;
```

以下是第三次运行的结果信息：

```
#9,按 1%精度验证模型
ky0=1; n_df9,94,n_dfk,38
acc: 40.43%;
```

第 16 章

人工神经网络算法

人工神经网络算法，英文全称是 Artificial Neural Network Algorithms，简称 ANN 算法。

ANN 算法属于类模式匹配算法，模拟生物神经网络，采用类似于大脑神经突触连接的结构进行信息处理，通常用于解决分类和回归问题。

ANN 算法是 20 世纪 40 年代后出现的。模型由众多的神经元可调的连接权值连接而成，具有大规模并行处理，分布式信息存储，良好的自组织、自学习能力等特点。

在神经网络算法当中，比较经典的 BP（反向传播）算法是人工神经网络中的一种监督式的学习算法。

反向传播算法在理论上可以逼近任意函数，基本的结构由非线性变化单元组成，具有很强的非线性映射能力。而且网络的中间层数、各层的处理单元数及网络的学习系数等参数可根据具体情况设定，灵活性很大。在数据优化、信号处理、模式识别、智能控制、故障诊断等领域都有广泛的应用前景，如图 16.1 所示。

ANN 算法是机器学习的一个庞大分支，目前热门的深度学习就是由神经网络算法发展而来的。

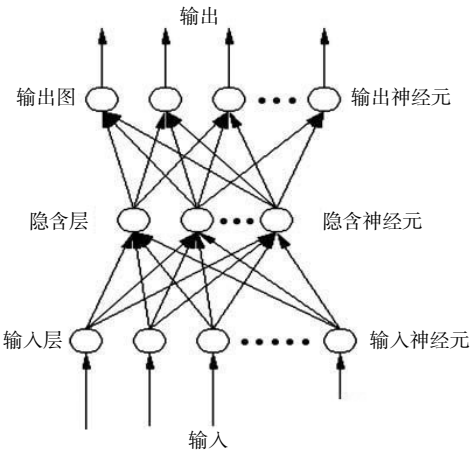


图 16.1　神经网络算法

ANN 算法模型里面的各种节点被称为“神经元”或“单元”，这些节点之间相互连接构成的网络就是“神经网络”。

ANN 算法模型需要进行训练，训练的过程就是网络进行训练学习的过程，通过训练学习过程来改变各个节点的连接权重参数值。

ANN 算法模型的主要缺点如下。

- 算法模型收敛速度慢。
- 计算量巨大。
- 训练时间长。
- 缺乏足够的理论基础。

常见的 ANN 算法如下。

- PNN（感知器神经网络）算法。
- BP（反向传播）算法。
- Hopfield 反馈网络算法。
- Elman 反馈网络算法。
- SOM（自组织映射）算法。
- RBF（径向基）算法。

- Boltzmann 随机神经网络算法。
- Hamming 网络算法。
- LVQ（学习矢量化）算法。

在 sklearn 0.17 版以前，对于神经网络算法的支持很少，通常都是使用其兼容模块库 sknn 代替，或者使用其他的 Python 模块库，比如 Theano、PyBrain、FFNN、Pylearn 2 等。

自 sklearn 0.18 版开始，sklearn 强化了这方面的功能，目前 neural_network 模块提供了以下三种算法函数。

- BernoulliRBM，伯努利受限玻尔兹曼机神经网络算法，简称 RBM 算法。
- MLPClassifier，多层感知器神经网络算法，简称 MLP 分类算法。
- MLPRegressor，MLP 回归算法。

多层感知器

多层感知器，又称多层神经网络，英文全称是 Multi-layer Perceptron，简称 MLP，是一种前向结构的人工神经网络，映射一组输入向量到一组输出向量。

MLP 是常见的神经网络算法，它由一个输入层、一个输出层和一个或多个隐藏层组成，如图 16.2 所示。

在 MLP 模型中，所有神经元都差不多，每个神经元都有几个输入（连接前一层）神经元和输出（连接后一层）神经元，该神经元会将相同值传递给与之相连的多个输出神经元。

MLP 模型可以被看作一个有向图，由多个节点层组成，每一层全连接到下一层。除了输入节点外，其他每个节点都是一个带有非线性激活函数的神经元（或称处理单元）。反向传播算法的监督学习方法常被用来训练 MLP。MLP 模型是感知器的推广，克服了感知器不能对线性不可分数据进行识别的弱点。

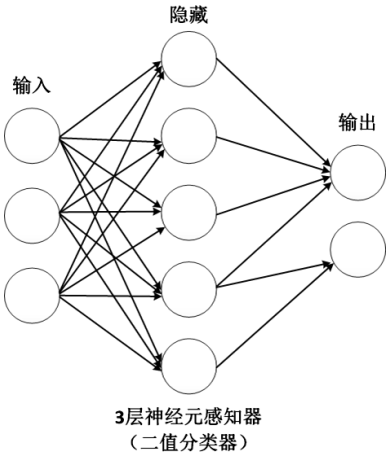


图 16.2　多层感知器

若每个神经元的激活函数都是线性函数，那么，任意层数的 MLP 都可被简化成一个等价的单层感知器。

实际上，MLP 模型本身可以使用任何形式的激活函数，比如阶梯函数或逻辑乙形函数（Logistic Sigmoid Function），但为了使反向传播算法进行有效学习，激活函数必须限制为可微函数。由于具有良好的可微性，很多乙形函数，尤其是双曲正切函数（Hyperbolic Tangent）及逻辑乙形函数常被用为激活函数。

常被 MLP 模型用来学习的反向传播算法，在模式识别的领域中算是标准监督学习算法，并在计算神经学及并行分布式处理领域中，持续成为被研究的课题。多层感知器已被证明是一种通用的函数近似方法，可以被用来拟合复杂的函数或解决分类问题。

案例 16-1：多层感知器回归算法

案例 16-1 的文件名是 kb1601_mlp_Reg.py，本案例主要介绍 sklearn 模块库中多层感知器回归算法的具体应用。

前面几组代码属于数据准备，和前面的案例 9-2 大同小异，故此不再

重复说明。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = neural_network.MLPRegressor()
```

调用 MLPRegressor（多层感知器回归）函数生成模型，并保存在变量 mx 中。

sklearn 模块库重点是传统的机器学习算法，对于神经网络新一代深度学习算法的支持比较少，相关的函数全部收录在 neural_network 子模块中。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

按惯例，调用 fit 函数，训练模型。

第 7 组代码如下：

```
#7
print('#7,predict 模型预测数据')
df_test['close_pred']=mx.predict(xtst)
zt.prDF('df',df_test)
```

当模型训练完成后，调用模型内置的 predict 预测函数，使用测试数据集 df_test，生成验证数据。

对应的输出信息如图 16.3 所示。

第 8 组代码如下：

```
#8
print('#8,绘制对比数据图')
df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 9 组代码如下：

```
#9
print('#9,按 1%精度验证模型')
dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
```

```
['close_pred'],1,True)
```

```
#7,predict模型预测数据

 df
              open     high    close      low       volume        amount   xprice  close_next  close_pred
date
2018-01-02  3314.03  3349.05  3348.33  3314.03  20227886000  227788461113  3369.11     3369.11     3363.94
2018-01-03  3347.74  3379.92  3369.11  3345.29  21383614900  258366523235  3385.71     3385.71     3394.52
2018-01-04  3371.00  3392.83  3385.71  3365.30  20695528800  243090768694  3391.75     3391.75     3412.28
2018-01-05  3386.46  3402.07  3391.75  3380.24  21306068100  248187840542  3409.48     3409.48     3424.18
2018-01-08  3391.55  3412.73  3409.48  3384.56  23616510600  286213219095  3413.90     3413.90     3432.82
2018-01-09  3406.11  3417.23  3413.90  3403.59  19148855100  238249975070  3421.83     3421.83     3443.74
2018-01-10  3414.11  3430.21  3421.83  3398.84  20909499700  254515441261  3425.34     3425.34     3449.91
2018-01-11  3415.58  3426.48  3425.34  3405.64  17381213300  218414134129  3428.94     3428.94     3451.28
2018-01-12  3423.88  3435.42  3428.94  3417.98  17406340400  215961455748  3410.49     3410.49     3460.49
2018-01-15  3428.95  3442.50  3410.49  3402.31  23200928300  286362732919  3436.59     3436.59     3457.20

              open     high    close      low       volume        amount   xprice  close_next  close_pred
date
2018-05-11  3179.80  3180.76  3163.26  3162.21  13065974900  167364290366  3174.03     3174.03     3203.79
2018-05-14  3167.04  3183.82  3174.03  3163.48  12932735300  172410691054  3192.12     3192.12     3204.06
2018-05-15  3180.42  3192.81  3192.12  3164.52  12454905100  162990790010  3169.56     3169.56     3212.87
2018-05-16  3180.23  3191.95  3169.56  3166.81  13052496800  174590979834  3154.28     3154.28     3210.28
2018-05-17  3170.01  3172.77  3154.28  3148.62  11399556700  150598842185  3193.30     3193.30     3193.56
2018-05-18  3151.08  3193.45  3193.30  3144.78  13651691800  168038057477  3213.84     3213.84     3201.33
2018-05-21  3206.18  3219.74  3213.84  3203.34  16445941300  202663464515  3214.35     3214.35     3242.67
2018-05-22  3211.25  3214.59  3214.35  3192.23  14429268400  185721667752  3168.96     3168.96     3238.55
2018-05-23  3205.44  3205.44  3168.96  3168.96  15780764800  199358101015  3154.65     3154.65     3221.16
2018-05-24  3167.94  3173.53  3154.65  3152.07  12408580000  160658185502  3154.65     3154.65     3194.49
```

图 16.3　预测数据

验证 MLPRegressor（多层感知器回归）函数的预测结果，以下是对应的输出信息：

```
#9,按 1%精度验证模型
ky0=1; n_df9,94,n_dfk,51
acc: 54.26%;
```

运行结果表明在 1%的精度下，准确度是 59.57%。

需要注意的是，MLP（多层感知器）算法属于神经网络算法，内部采用了随机数作为种子，因此，每次运行的结果都不一定相同。

以下是第二次运行的结果信息：

```
#9,按 1%精度验证模型
ky0=1; n_df9,94,n_dfk,63
acc: 67.02%
```

以下是第三次运行的结果信息：

```
#9,按 1%精度验证模型
ky0=1; n_df9,94,n_dfk,65
acc: 69.15%
```

案例 16-2：多层感知器分类算法

案例 16-2 的文件名是 kb1602_mlp_cla.py，本案例主要介绍 sklearn 模块库中多层感知器分类算法的具体应用。

前面几组代码属于数据准备，和前面的案例 9-2 大同小异，故此不再重复说明。

因为是分类算法，所以数据准备阶段采用的是分类模式，大家注意相关源码。

第 5 组代码如下：

```
#5
print('#5,模型设置')
mx = neural_network.MLPClassifier()
```

调用 MLPClassifier（多层感知器分类）函数生成模型，并保存在变量 mx 中。

第 6 组代码如下：

```
#6
print('#6,fit 训练模型')
mx.fit(x,y)
```

按惯例，调用 fit 函数，训练模型。

第 7 组代码如下：

```
#7
print('#7,predict 模型预测数据')
df_test['close_pred']=mx.predict(xtst)
zt.prDF('df',df_test)
```

当模型训练完成后，调用模型内置的 predict 预测函数，生成验证数据，对应的输出信息，如图 16.4 所示。

```
#7,predict模型预测数据

 df
              open     high      low    close          vol           cap   xprice      kpr  close_next  close_pred
time
2018-01-01  14112.2  14112.2  13154.7  13657.2  10291200000  236725000000  14982.1  1097.01           2           2
2018-01-02  13625.0  15444.6  13163.6  14982.1  16846600000  228579000000  15201.0  1014.61           2           2
2018-01-03  14978.2  15572.8  14844.5  15201.0  16871900000  251312000000  15599.2  1026.20           2           2
2018-01-04  15270.7  15739.7  14522.2  15599.2  21783200000  256250000000  17429.5  1117.33           2           2
2018-01-05  15477.2  17705.2  15202.8  17429.5  23840900000  259748000000  17527.0  1005.59           2           2
2018-01-06  17462.1  17712.4  16764.6  17527.0  18314600000  293091000000  16477.6   940.13           1           2
2018-01-07  17527.3  17579.6  16087.7  16477.6  15866000000  294222000000  15170.1   920.65           1           2
2018-01-08  16476.2  16537.9  14208.2  15170.1  18413900000  276612000000  14595.4   962.12           1           2
2018-01-09  15123.7  15497.5  14424.0  14595.4  16660000000  253935000000  14973.3  1025.89           2           2
2018-01-10  14588.5  14973.3  13691.2  14973.3  18500800000  244981000000  13405.8   895.31           1           2

              open     high      low    close         vol           cap   xprice      kpr  close_next  close_pred
time
2018-04-14  7874.67  8140.71  7846.00  7986.24  5191430000  133682000000  8329.11  1042.93           2           2
2018-04-15  7999.33  8338.42  7999.33  8329.11  5244480000  135812000000  8058.67   967.53           1           2
2018-04-16  8337.57  8371.15  7925.73  8058.67  5631310000  141571000000  7902.09   980.57           1           2
2018-04-17  8071.66  8285.96  7881.72  7902.09  6900880000  137070000000  8163.42  1033.07           2           2
2018-04-18  7944.43  8197.80  7886.01  8163.42  6529910000  134926000000  8294.31  1016.03           2           2
2018-04-19  8159.27  8298.69  8138.78  8294.31  7063210000  138591000000  8845.83  1066.49           2           2
2018-04-20  8286.88  8880.23  8244.54  8845.83  8438110000  140777000000  8895.58  1005.62           2           2
2018-04-21  8848.79  8997.57  8652.15  8895.58  7548550000  150337000000  8802.46   989.53           1           2
2018-04-22  8925.06  9001.64  8779.61  8802.46  6629900000  151651000000  8930.88  1014.59           2           2
2018-04-23  8794.39  8958.55  8788.81  8930.88  6925190000  149448000000  8930.88  1014.59           0           2
```

图 16.4　预测数据

第 8 组代码如下：

```
    #8
    print('#8,绘制对比数据图')
    df_test[['close_next','close_pred']].plot(linewidth=3)
```

第 9 组代码如下：

```
    #9
    print('#9,按 1%精度验证模型')
    dacc,df,xlst=zai.ai_acc_xed2ext(df_test['close_next'],df_test
['close_pred'],1,True)
```

验证 BaggingClassifierKNN 算法模型的预测结果，对应的输出信息如下：

```
    #9,按 1%精度验证模型
    ky0=1; n_df9,113,n_dfk,51
    acc: 45.13%
```

结果表明在 1%的精度下，准确度是 49.56%。

第 10 组代码如下：

```
    #10
    print('\n#10,value_counts')
    print("\ndf_test['close_next'].value_counts()")
    print(df_test['close_next'].value_counts())
```

```
print("\ndf_test['close_pred'].value_counts()")
print(df_test['close_pred'].value_counts())
```

本组代码是分类算法的附加代码，调用 pandas 的 value_counts 函数，查看 close_next（实盘数据字段）、close_pred（预测数据字段）的具体数据分布情况。

对应的输出信息是：

```
#10,value_counts

df_test['close_next'].value_counts()
2    54
1    51
0     8
Name: close_next, dtype: int64

df_test['close_pred'].value_counts()
1    74
2    39
Name: close_pred, dtype: int64
```

多层感知器算法属于神经网络算法，内部采用了随机数作为种子，每次运行的结果都不一定相同。

以下是第二次运行的结果信息：

```
#9,按 1%精度验证模型
ky0=1; n_df9,113,n_dfk,48
acc: 42.48%;
```

以下是第三次运行的结果信息：

```
#9,按 1%精度验证模型
ky0=1; n_df9,113,n_dfk,58
acc: 51.33%;
```

附录 A

sklearn 常用模块和函数

本附录主要用于查询 sklearn 人工智能、机器学习的相关函数。

1．模块库清单

sklearn 人工智能、机器学习模块，常用的子模块库如下。

- sklearn.base：基础模块库。
- sklearn.utils：常用工具模块库。
- sklearn.cluster：聚类模块库。
- sklearn.cluster.bicluster：双聚类模块库。
- sklearn.covariance：协方差评估模块库。
- sklearn.model_selection：模型选择模块库。

  该模块库包括：

  - Splitter Classes：分割器类。
  - Splitter Functions：分割器函数。
  - Hyper-parameter：超参数优化。
  - optimizers Model validation：优化模型验证。

- sklearn.datasets：数据集模块库。
- sklearn.decomposition：矩阵分解模块库。
- sklearn.dummy：伪变量模块库。
- sklearn.ensemble：集成算法模块库。
- sklearn.exceptions；异常和警告模块库。
- sklearn.feature_extraction：特征提取模块库。
- sklearn.feature_selection；特征选择模块库。
- sklearn.gaussian_process：高斯过程模块库。
- sklearn.isotonic：保序回归模块库。
- sklearn.kernel_approximation；核近似模块库。
- sklearn.kernel_ridge；核岭回归模块库。
- sklearn.discriminant_analysis：判别分析模块库。
- sklearn.linear_model：线性模型模块库。
- sklearn.manifold：流形学习模块库。
- sklearn.metrics：结果测度模块库。

  该模块库包括：

  - Model Selection Interface：模型选择界面。
  - Classification metrics：分类度量。
  - Regression metrics：回归度量。
  - Multilabel ranking metrics：多类别排名度量。
  - Clustering metrics：聚类度量。
  - Biclustering metrics：双聚类度量。
  - Pairwise metrics：配对度量。
- sklearn.mixture：高斯混合模型模块库。
- sklearn.multiclass：多分类模块库。

- sklearn.multioutput：多输出回归和分类模块库。
- sklearn.naive_bayes：朴素贝叶斯算法模块库。
- sklearn.neighbors：近邻算法模块库。
- sklearn.neural_network：神经网络算法模块库。
- sklearn.calibration：校准概率模块库。
- sklearn.cross_decomposition：交叉分解算法模块库。
- sklearn.pipeline：管道数据记录模块库。
- sklearn.preprocessing：预处理和归一化模块库。
- sklearn.random_projection：随机投影算法模块库。
- sklearn.semi_supervised；半监督学习算法模块库。
- sklearn.svm：向量机算法模块库。
- sklearn.tree：决策树算法模块库。

2．Base（基础）模块

Base 基础模块库 sklearn.base，包括 Base 基类定义和工具函数，常用类定义如下。

- BaseEstimator：测试评估基类。
- ClassifierMixin：混合分类器。
- ClusterMixin：混合聚类等基类。
- RegressorMixin：混合回归评估基类。
- TransformerMixin：混合转换基类。

sklearn.base 常用函数是 base.clone，即复制一个新的基类变量。

3．Utils（常用工具）模块

sklearn.utils：常用工具模块库，包括常用的工具函数和各种实用程序。

- check_random_state：生成 Randomstate 实例。
- estimator_checks.check_estimator：检查评估参数。

- resample：数组或稀疏矩阵，重采样。
- shuffle：数组或稀疏矩阵混洗整理。

4．Exceptions（异常和警告）

sklearn.exceptions：异常和警告模块库，包括 sklearn 模块库所有的警告、出错信息和处理函数。

- NotFittedError：如果分类器在拟合前就使用，则抛出异常。
- ChangedBehaviorWarning：警告类通知，用户有任何行为上的变更。
- ConvergenceWarning：自定义警告，来捕捉收敛问题
- DataConversionWarning：警告通知，用于隐式数据转换代码。
- DataDimensionalityWarning：自定义警报，通知用户数据维度的潜在问题。
- EfficiencyWarning：警告通知用户，计算效率偏低。
- FitFailedWarning：警告类，用于拟合评估时存在的误差。
- NonBLASDotWarning：当没使用 BLAS 就计算点运算时，出现警告。
- Undefinedmetricwarning：无效度量时出现警告。

5．Cluster（聚类）模块

sklearn.cluster：无监督聚类算法模块库，主要的类定义如下。

- AffinityPropagation：仿射传播聚类算法。
- AgglomerativeClustering：凝聚聚类算法。
- BIRCH：聚类算法。
- DBSCAN：矢量聚类算法（根据矩阵距离）。
- FeatureAgglomeration：团簇特征聚类算法。
- KMeans：K-均值聚类算法。
- MiniBatchKMeans：迷你批次 K-均值聚类算法。

- MeanShift：均值偏移聚类算法。
- SpectralClustering：投影归一化（拉普拉斯算子）聚类算法。主要的函数如下。
- estimate_bandwidth：评估均值偏移算法使用的带宽。
- k_means：K-means 聚类算法。
- ward_tree：沃德基于聚类的特征矩阵。
- affinity_propagation：仿射传播聚类算法。
- dbscan：DBSCAN 密度聚类算法。
- mean_shift：均值偏移聚类算法。
- spectral_clustering：投影归一化（拉普拉斯算子）聚类算法。

6．cluster.bicluster（双聚类）

sklearn.cluster.bicluster：双聚类模块库，主要包括以下算法。

- SpectralBiclustering：Kluger 双谱聚类算法。
- SpectralCoclustering：Dhillon 同谱聚类算法。

7．Covariance（协方差评估）

sklearn.covariance：协方差评估模块，包括一组稳健特征协方差评估算法。精度矩阵定义为逆的协方差，协方差评估与高斯图形模型理论密切相关。包括以下函数。

- EmpiricalCovariance：最大似然评估量协方差。
- EllipticEnvelope：用于检测对象的离群值的高斯分布数据集。
- GraphLasso：稀疏逆协方差评估算法，使用 L1 惩罚评估器。
- GraphLassoCV：稀疏逆协方差交叉验证，使用 L1 惩罚评估器。
- LedoitWolf：估计量。
- MinCovDet：最小协方差行列式（MCD），稳健评估量的协方差。

- OAS：Oracle 逼近收缩评估量。
- ShrunkCovariance：协方差收缩评估算法。
- empirical_covariance：计算最大似然评估量协方差。
- ledoit_wolf：协方差矩阵评估算法。
- shrunk_covariance：矩阵对角线协方差计算。
- OAS：中心评估协方差与 Oracle 逼近收缩算法。
- graph_lasso：L1 处罚协方差评估算法。

8．model_selection（模型选择）

sklearn.model_selection：模型选择模块库，包括以下几大部件。

- Splitter Classes：分割器类。
- Splitter Functions：分割器函数。
- Hyper-parameter：超参数优化。
- optimizers Model validation：模型验证优化。

（1）Splitter Classes（分割器类）

- KFold：K-折交叉验证。
- GroupKFold：K-折迭代变量的非重叠组。
- StratifiedKFold：分层 K-折交叉验证算法。
- LeavePGroupsOut：留一组交叉验证算法。
- leavepgroupsoutN：留 P 组交叉验证程序。
- LeaveOneOut：留一法交叉验证程序。
- LeavePOut：留 P 法交叉验证算法。
- ShuffleSplit：随机置换交叉验证算法。
- GroupShuffleSplit：混洗组交叉迭代验证算法。
- StratifiedShuffleSplit：分层混洗分割交叉验证程序。

- PredefinedSplit：预定义分割交叉验证程序。
- TimeSeriesSplit：时间序列交叉验证程序。

（2）Splitter Functions（分割函数）

- train_test_split：训练和测试数据集分割函数。
- check_cv：建立交叉验证输入检查器。

（3）Hyper-parameter（超参数优化）

- GridSearchCV：网格搜索交叉验证。
- RandomizedSearchCV：随机搜索交叉验证。
- ParameterGrid：对每一个离散值进行参数网格搜索。
- ParameterSampler：指定分布按参数抽样生成器。
- fit_grid_point：按一组参数运行拟合。

（4）optimizers Model validation（模型验证优化）

- cross_val_score：交叉验证评估得分。
- cross_val_predict：对每一个输入数据点生成交叉验证评估器。
- permutation_test_score：评估置换交叉验证测试得分的重要性。
- learning_curve：学习曲线。
- validation_curve：验证曲线。

9．Datasets（数据集）模块

sklearn.datasets：数据集模块库模块，包括一些常用的测试数据，以及数据的加载函数，同时还提供一些人工数据生成器函数。

（1）Loader（数据加载）

- clear_data_home：清除所有加载的数据。
- get_data_home：返回 scikit-learn 数据 dir 路径。
- fetch_20newsgroups：抓取 20 组新闻数据集的文件名和数据。

- fetch_20newsgroups_vectorized：抓取 20 组新闻数据集的 TF-IDF 向量化数据。
- load_boston：加载波士顿房屋价格数据集（回归）。
- load_breast_cancer：加载威斯康星州乳腺癌数据集（分类）。
- load_diabetes：加载糖尿病数据集（回归）。
- load_digits：加载数字 OCR 识别图像数据集（分类）。
- load_files：从类别子文件夹加载文本文件。
- load_iris：加载 Iris（爱丽丝）经典数据集（分类）。
- fetch_lfw_pairs：加载 LFW 人脸两两数据集。
- fetch_lfw_people，LFW 人脸数据集。
- load_linnerud：加载 linnerud 多元回归数据集。
- mldata_filename：从 mldata.org 网站抓取数据集文件。
- fetch_mldata：从 mldata.org 网站抓取数据集。
- fetch_olivetti_faces：加载 Olivetti 面部数据集。
- fetch_california_housing：加载加州房屋数据集。
- fetch_covtype：加载并下载 covertype 数据集。
- fetch_kddcup99：加载 kddcup99 数据集。
- fetch_rcv1：加载并下载 RCV1 多标签数据集。
- load_mlcomp：因 mlcomp.org 数据网站将在 2017 年 3 月关闭，本加载函数在 0.19 和 0.21 版将被删除。
- load_sample_imageimage_name：采用 NumPy 数组加载单个图像样本。
- load_sample_images：加载 NumPy 数组格式的样本图像。
- fetch_species_distributions：加载物种分布数据集，来自 Phillips 等机构。
- load_svmlight_file：从文件加载 SVMLight/libsvm 文件格式的的数据，并转换为 CSR 稀疏矩阵格式。

- load_svmlight_files：从多个文件加载 SVMLight/libsvm 文件格式的数据，并转换为 CSR 稀疏矩阵格式。
- dump_svmlight_file：把数据集转为 SVMLight/libsvm 文件格式，并保存。

（2）样本数据生成器

- make_blobs：生成同性高斯斑点 blob 聚类数据。
- make_classification：产生随机 *N* 类分类数据。
- make_circles：生成二维包含指定的多个小圆的大圆圈数据。
- make_friedman1：生成的“弗里德曼#1”回归数据集。
- make_friedman2：生成的“弗里德曼#2”回归数据集。
- make_friedman3：生成的“弗里德曼#3”回归数据集。
- make_gaussian_quantiles：生成同性高斯样本分位数和标签。
- make_hastie_10_2：生成 Hastie 等二进制分类数据。
- make_low_rank_matrix：生成具有钟形奇异值的低秩矩阵。
- make_moons：生成两个交错半圆。
- make_multilabel_classification：生成随机多标记分类数据。
- make_regression：生成随机回归数据。
- make_s_curve：生成 S 曲线数据集。
- make_sparse_coded_signal：生成稀疏组合信号的字典数据。
- make_sparse_spd_matrix：生成 SPD 稀疏对称正定矩阵。
- make_sparse_uncorrelated：生成随机不相关稀疏数据集。
- make_spd_matrix：生成 SPD 随机对称正定矩阵。
- make_swiss_roll：生成瑞士卷数据集。
- make_biclusters：生成双聚类块对角结构矩阵。

- make_checkerboard：生成块结构的双聚类棋盘矩阵。

10．Decomposition（矩阵分解）

sklearn.decomposition：矩阵分解模块库，包括矩阵分解模块算法，例如 PCA、ICA 或 NMF 等算法，该模块可作为降维使用。

- PCA：主成分分析。
- IncrementalPCA：增量主成分分析。
- KernelPCA：核主成分分析（KPCA）。
- FactorAnalysis：因子分析（FA）。
- FastICA：一种快速独立分量分析算法。
- TruncatedSVD：使用截断奇异值分解降维（又称作 LSA）。
- NMF：非负矩阵分解。
- SparsePCA：稀疏主成分分析。
- MiniBatchSparsePCA：小批量稀疏主成分分析。
- SparseCoder：稀疏编码。
- DictionaryLearning：学习词典。
- MiniBatchDictionaryLearning：小批量学习词典。
- LatentDirichletAllocation：潜在狄利克雷分配与变分贝叶斯网络算法。
- Fastica：快速独立分量分析。
- dict_learning：解决学习字典矩阵分解问题。
- dict_learning_online：在线解决学习字典矩阵分解问题。
- sparse_encode：稀疏编码。

11．Dummy（伪变量）

sklearn.dummy：伪变量模块库。

- DummyClassifier：简单规则预测算法。

- DummyRegressor：简单规则回归预测算法。

12．Ensemble（集成）算法

sklearn.ensemble：集成算法模块，包括集成分类算法、回归算法和异常检测算法。

- AdaBoostClassifier：AdaBoost 迭代分类算法，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）。
- AdaBoostRegressor：AdaBoost 回归算法。
- BaggingClassifier：Bagging 算法，相当于多个专家投票表决，对于多次测试，每个样本返回的是多次预测结果较多的那个。
- BaggingRegressor：Bagging 回归算法。
- ExtraTreesClassifier：完全随机树算法。
- ExtraTreesRegressor：完全随机树回归算法。
- GradientBoostingClassifier：梯度增强分类算法。
- GradientBoostingRegressor：梯度增强回归算法。
- IsolationForest：隔离森林算法。
- RandomForestClassifier：随机森林算法
- RandomTreesEmbedding：完全随机树算法。
- RandomForestRegressor：随机森林回归算法。
- VotingClassifier：投票算法，多种机器学习算法的集成式算法。

部分依赖 Tree 决策树的算法如下。

- partial_dependence.partial_dependence：局部变量依赖算法。
- partial_dependence.plot_partial_dependence：局部变量依赖算法绘图。

13．feature_selection（特征选择）

sklearn.feature_selection：特征选择模块，包括各种特征选择算法，如

单变量特征选择算法、滤波器特征选择算法、递归特征消除算法等。

- GenericUnivariateSelect：通用单变量特征选择器的配置策略。
- SelectPercentile：按百分比最高分选择特征。
- SelectKBest：按最佳 k 值选择特征。
- SelectFpr：滤波器，在 FPR 测试中选择低于 alpha 的 p 值。
- SelectFdr：滤波器，为评估错误发现率选择 p 值。
- SelectFromModel：元数据转换选择特征算法，基于输入的权重数据。
- SelectFwe：过滤器，根据多重比较错误率选择 p 值。
- RFE：递归特征消除算法的特征排序。
- RFECV：递归特征消除算法的交叉验证特征排序。
- VarianceThreshold：特征选择算法，除去所有低方差特征。
- chi2：计算卡方统计每个非负特征和类。
- f_classif：计算样品的方差分析 F-值。
- f_regression：一元线性回归测试。
- mutual_info_classif：评估离散目标变量的互信息。
- mutual_info_regression：评估连续目标变量的互信息。

14．feature_extraction（特征提取）

sklearn.feature_extraction：特征提取模块，主要用于从原始数据提取特征数据，包括提取数值、文本、图像等数据的特征提取。

- DictVectorizer：变换的特征值映射为向量。
- FeatureHasher：把特征值采用 Hash（哈希）函数进行散列变换。

（1）图像特征提取

sklearn.feature_extraction.image：从图像中提取特征。

- image.img_to_graph：像素-像素梯度连接图表。

- image.grid_to_graph：像素-像素连接网格图表。
- image.extract_patches_2d：把 2D 图像重塑为小块模块容器。
- image.reconstruct_from_patches_2d：从小块模块容器重建图像。
- image.PatchExtractor：从图像容器中抽取小块。

（2）文本特征提取

sklearn.feature_extraction.text 模块：从文本信息建立特征向量。

- text.CountVectorizer：将文本集合转换为 token 关键词统计向量矩阵。
- text.HashingVectorizer：采用哈希算法，将文本转换为向量矩阵。
- text.TfidfTransformer：把统计矩阵转换为归一化 tf 或 tf-idf 表征。
- text.TfidfVectorizer：把原文档转换为 tf-idf 特征矩阵。

15．gaussian_process（高斯过程）

sklearn.gaussian_process 模块实现了基于高斯过程回归和分类算法。

- GaussianProcessRegressor：高斯过程回归（GPR）算法。
- GaussianProcessClassifier：高斯过程分类（GPC）算法。

Kernels 子模块功能：

- kernels.Kernel：核的基类。
- kernels.Sum：k1、k2 两个核求和运算。
- kernels.Product：k1、k2 两个核乘积运算。
- kernels.Exponentiation：乘方运算，根据给定的指数运算。
- kernels.ConstantKernel：核常量。
- kernels.WhiteKernel：白色核。
- kernels.RBF：核半径函数，径向基核函数（又名平方指数内核）。
- kernels.Matern：母体核。
- kernels.RationalQuadratic：有理二次核。

- kernels.ExpSineSquared：正弦平方核。
- kernels.DotProduct：点积核。
- kernels.PairwiseKernel：sklearn.metrics.pairwise 内的核封装。
- kernels.CompoundKernel：由一组其他核构建的核。
- kernels.Hyperparameter：命名元组形式的超参数内核规范。

16．Isotonic（保序）回归模块库。

sklearn.isotonic 模块：保序回归算法。

- IsotonicRegression：保序回归模型。
- isotonic_regression：保序回归模型求解算法。
- check_increasing(x, y)：y 与 x 的单调相关确认算法。

17．kernel_approximation（核近似）模块

sklearn.kernel_approximation 模块收录了核近似算法，基于傅里叶变换特征映射内核近似算法。

- AdditiveChi2Sampler：添加卡方核近似特征映射。
- Nystroem：奈斯特龙 Nystroem 内核近似算法，使用训练数据的一个映射子集。
- RBFSampler：RBF 内核近似算法，基于蒙特卡洛傅里叶变换的 RBF 函数映射。
- SkewedChi2Sampler：近似特征映射的“斜卡方”内核，蒙特卡洛傅里叶变换算法。

18．kernel_ridge（核岭回归）模块库。

klearn.kernel_ridge 模块：实现核岭回归算法。

- KernelRidge：内核岭回归算法。

19．discriminant_analysis（判别分析）

sklearn.discriminant_analysis 模块：判别分析算法。

- LinearDiscriminantAnalysis：线性判别分析算法。
- QuadraticDiscriminantAnalysis：二次判别分析算法。

20．linear_model（线性模型）

sklearn.linear_model 模块库：通用线性模型算法，包括岭回归算法、套索回归算法、贝叶斯算法、弹性算法与最小二乘回归算法等，它还实现了相关的随机梯度下降算法。

- ARDRegression：ARD 贝叶斯算法。
- BayesianRidge：贝叶斯岭回归算法。
- ElasticNet：使用 L1 和 L2 正则的线性回归算法。
- ElasticNetCV：正则路径迭代拟合的交叉验证弹性网络。
- HuberRegressor：稳健离群值的线性回归算法。
- Lars：拉斯最小角回归算法。
- LarsCV：交叉验证最小角回归算法。
- Lasso：采用 L1 正则训练数据的线性算法（又称 Lasso（套索））。
- LassoCV：正则路径迭代拟合的交叉验证 Lasso（套索）线性算法。
- LassoLars：套索与最小角度拟合回归算法。
- LassoLarsCV：交叉验证套索与最小角度拟合回归算法。
- LassoLarsIC：套索与最小角度拟合回归算法，使用 AIC 或 BIC 选择模型。
- LinearRegression：普通最小二乘线性回归算法。
- LogisticRegression，逻辑回归算法（又称 logit.MaxEnt）。
- LogisticRegressionCV：交叉验证逻辑回归算法（又称 logit.MaxEnt）。

- MultiTaskLasso：多任务 Lasso（套索）算法，使用 L1/L2 混合正则。
- MultiTaskElasticNet：多任务弹性网络算法，使用 L1/L2 混合正则。
- MultiTaskLassoCV：交叉验证 Lasso（套索）算法。
- MultiTaskElasticNetCV：交叉验证多任务弹性网络算法。
- OrthogonalMatchingPursuit：正交匹配追踪（OMP）算法。
- OrthogonalMatchingPursuitCV：交叉验证正交匹配追踪（OMP）算法。
- PassiveAggressiveClassifier：被动的分类算法。
- PassiveAggressiveRegressor：被动侵略性回归算法。
- Perceptron：感知器算法。
- RandomizedLasso：随机套索算法。
- RandomizedLogisticRegression：随机逻辑回归算法。
- RANSACRegressor：RANSAC（RANdom SAmple Consensus）算法，统一随机采样算法。
- Ridge：岭回归算法。
- RidgeClassifier：岭回归分类算法。
- RidgeClassifierCV：交叉验证岭回归分类算法。
- RidgeCV：交叉验证岭回归算法。
- SGDClassifier：线性分类算法（支持向量机，逻辑回归等），使用随机梯度下降算法算法训练数据。
- SGDRegressor：正则化线性拟合算法，采用最小化经验损失与随机梯度下降算法。
- TheilSenRegressor：Theil-Sen 算法，采用稳健多变量回归模型。
- lars_path：最小角回归算法，使用套索路径或 LARS 拉斯算法。
- lasso_path：计算索套路径坐标下降数据。
- lasso_stability_path：基于随机套索稳定性评估的稳定路径算法。

- logistic_regression_path：计算逻辑回归模型，用于正则化参数列表。
- orthogonal_mp：正交匹配追踪（OMP）算法。
- orthogonal_mp_gram：gram 正交匹配追踪（OMP）算法。

21．Manifold（流形学习）

sklearn.manifold 模块：流形学习算法、数据嵌入技术等。

- LocallyLinearEmbedding：局部线性嵌入算法。
- Isomap：映射嵌入算法。
- MDS：多维缩放算法。
- SpectralEmbedding：非线性频谱嵌入算法。
- TSNE：*t* 分布随机近邻嵌入算法。
- locally_linear_embedding：执行局部线性嵌入算法分析数据。
- spectral_embedding：基于图拉普拉斯第一特征向量的投影样本算法。
- SMACOF：多维度计算算法。

22．Mixture（高斯混合）模型

sklearn.mixture 模块：高斯建模算法。

- GaussianMixture：高斯混合算法。
- BayesianGaussianMixture：高斯贝叶斯混合算法。

23．Multiclass（多分类）

- OneVsRestClassifier：简称 OvR（1-VS-rest），一对余的多分类/多标签策略算法。
- OneVsOneClassifier：简称 OvO，一对一多分类策略算法。
- OutputCodeClassifier（纠错码）：输出多分类策略算法。

24．Multioutput（多输出）回归和分类

sklearn.multioutput 模块：实现多输出回归和分类算法。

- MultiOutputRegressor：多目标回归算法。
- MultiOutputClassifier：多目标分类算法。

25．naive_bayes（朴素贝叶斯）算法

sklearn.naive_bayes 模块，包括朴素贝叶斯（Bayes）算法，这些是在监督学习方法的基础上，运用贝叶斯定理和强（朴素）特征独立性假设而开发的算法。

- GaussianNB：高斯朴素贝叶斯算法。
- MultinomialNB：多项式朴素贝叶斯算法。
- BernoulliNB：多元伯努利（Bernoulli）朴素贝叶斯算法。

26．Neighbors（近邻）算法

sklearn.neighbors 模块：实现了 K-近邻算法。

- NearestNeighbors：无监督学习邻居搜索算法。
- KNeighborsClassifier：K-近邻算法。
- RadiusNeighborsClassifier：给定半径内的近邻投票算法。
- KNeighborsRegressor：K-近邻回归算法。
- RadiusNeighborsRegressor：固定半径内的近邻回归算法。
- NearestCentroid：最近质心算法。
- BallTree：BallTree 快速广义 N 点问题算法。
- KDTree：KDTree 快速广义 N 点问题算法。
- LSHForest：基于 LSH 森林模型的近邻算法。
- DistanceMetric：距离度量类。
- KernelDensity：核密度估计。
- LocalOutlierFactor：LOF 算法，局部异常因子的无监督异常检测算法。
- kneighbors_graph：计算（加权）为点 x 的 k 近邻图。

- radius_neighbors_graph：计算（加权）点半径为 X 的近邻图。

27．neural_network（神经网络）

sklearn.neural_network 模块：包括基于神经网络模型算法。

- BernoulliRBM：RBM 算法，伯努利受限玻尔兹曼机算法，也称有限伯努利机算法。
- MLPClassifier：多层感知算法。
- MLPRegressor：多层感知回归算法。

28．Calibration（校准概率）

sklearn.calibration 模块：包括预测概率的校准算法。

- CalibratedClassifierCV：保序回归或 sigmoid 概率校准算法。
- calibration_curve：计算真实预测概率的校正曲线。

29．cross_decomposition（交叉分解）算法

sklearn.cross_decomposition 模块：交叉分解算法。

- PLSRegression：PLS 最小二乘回归算法。
- PLSCanonical：PLS-C2A 算法。
- CCA：典型相关分析算法。
- PLSSVD：偏最小二乘奇异分解算法。

30．Pipeline（管道数据记录）

sklearn.pipeline 模块：用于记录 sklearn 机器学习算法的各个流程。

- Pipeline：管道转换数据。
- FeatureUnion：串接多个转换器对象的结果。
- make_pipeline：按给定转换器构建管道。
- make_union：按给定转换器构建联合特征。

31．Preprocessing（预处理和归一化）

sklearn.preprocessing 模块：包括数据缩放、中心化、正则化、二元化及插补方法。

- Binarizer：按阈值对数据二元化（特征设为 0 或 1）。
- FunctionTransformer：从任意调度中构建转换器。
- Imputer：完成缺失数据插补。
- KernelCenterer：中心化核矩阵。
- LabelBinarizer：在一对剩余范式中二元化标签。
- LabelEncoder：用 0 到 n_class-1 对标签编码。
- MultiLabelBinarizer：对迭代器和多标签格式之间的转换器。
- MaxAbsScaler：对每一个特征按最大绝对值缩放。
- MinMaxScaler：对每一个特征按给定范围值缩放。
- Normalizer：分别对数据单位范数。
- OneHotEncoder：使用热码的整数编码分类特征。
- PolynomialFeatures：生成多项式和交互特征。
- RobustScaler：使用统计缩放特征，对离群值较稳健。
- StandardScaler：标准缩放特征，通过删除均值来缩放至单位方差。
- add_dummy_feature：通过添加伪特征增强数据集。
- binarize：布尔阈值化的数组或 SciPy 稀疏矩阵。
- label_binarize：在一对多范式中二元化标签。
- maxabs_scale：不破坏稀疏性的情况下对每个特征缩放至[-1,1]。
- minmax_scale：对每个特征按给定范围值缩放。
- normalize：缩放输入向量至单位范数。
- robust_scale：沿任意轴标准化数据。

- scale：沿任意轴标准化数据。

32．random_projection（随机投影）算法

sklearn.random_projection 模块：包括数据缩放、归一化、二值化和估算方法。

- GaussianRandomProjection：随机高斯投影降维算法。
- SparseRandomProjection：稀疏随机投影降维算法。
- johnson_lindenstrauss_min_dim：找到“安全”数目的分量随机项目。

33．semi_supervised（半监督学习）算法

sklearn.semi_supervised 模块实现了半监督学习算法，这些算法利用少量的标记数据和大量的未标记的数据分类任务，包括标签传播算法。

- LabelPropagation：标签传播算法。
- LabelSpreading：半监督学习的标签传播算法。

34．SVM（向量机）算法

sklearn.svm 模块包括支持向量机算法。

- SVC：C-支持向量分类算法。
- LinearSVC：线性支持向量分类算法。
- NuSVC：Nu 支持向量分类算法。
- SVR：Epsilon 支持向量回归算法。
- LinearSVR：线性向量回归算法。
- NuSVR：Nu 支持向量回归算法。
- OneClassSVM：无监督离群点检测。
- l1_min_c：返回最低边界的 C(l1_min_C,infinity)，保证模型为非空。

35．Tree（决策树）算法

sklearn.tree 模块：包括决策树分类和回归算法模型。

- DecisionTreeClassifier：决策树分类算法。
- DecisionTreeRegressor：决策树回归算法。
- ExtraTreeClassifier：完全随机树分类算法。
- ExtraTreeRegressor：完全随机树回归算法。
- export_graphviz：采用 DOT 格式输出决策树图。

36．sklearn.metrics（结果测度）

sklearn.metrics 度量模块，包括模型、算法评估等功能，有评分功能、性能指标和对指标和距离计算。包括以下运行部件。

- Model Selection Interface：模式选择界面。
- Classification metrics：分类度量。
- Regression metrics：回归度量。
- Multilabel ranking metrics：多标签排序度量。
- Clustering metrics：聚类度量。
- Biclustering metrics：双聚类度量。
- Pairwise metrics：成对度量。

（1）Model Selection Interface（模式选择界面）

- make_scorer：按性能测试或损失函数评分。
- get_scorer：获取性能测试评分。

（2）Classification metrics（分类度量）

- accuracy_score：准确性得分。
- AUC：采用梯形法计算曲线下面积。
- average_precision_score：从预测评分计算平均精确度（AP）。
- brier_score_loss：计算 Brier（布莱尔）分数。Brier 分数是评价频率预测准确度的一个指标，适用的前提是只关心某个事件是否发生（0~1

频度），而不关心事件发生的强度影响。

- classification_report：建立一个主要分类指标的文本报告。
- cohen_kappa_score：科恩 Kappa 评分，统计测量注释协议。
- confusion_matrix：计算混淆矩阵来评估分类准确性。
- f1_score：计算 F1 得分，又称为平衡 F 或 F-分数度量。
- fbeta_score：计算 F-beta 得分。
- hamming_loss：计算平均汉明损失。
- hinge_loss：平均连接损失（未正则化）。
- jaccard_similarity_score：Jaccard 相似性系数评分。
- log_loss：log 损失，逻辑损失，又称作交叉熵损失。
- matthews_corrcoef：计算二分类的马修斯相关系数（MCC）。
- precision_recall_curve：对不同概率阈值精度计算-召回率。
- precision_recall_fscore_support：计算准确率、查全率、F-measure 等参数。
- precision_score：计算精度。
- recall_score：计算召回率。
- roc_auc_score：计算曲线下面积（AUC）的预测评分。
- roc_curve：计算 ROC 受试者操作特征评分。
- zero_one_loss：零分类损失。

（3）Regression metrics（回归度量）

- explained_variance_score：解释方差回归得分函数。
- mean_absolute_error：平均绝对误差回归损失。
- mean_squared_error：回归均值平方误差损失。
- mean_squared_log_error：对数回归均值平方误差损失。
- median_absolute_error：中位数绝对误差回归损失。

- r2_score：R 方确定系数的回归得分函数。

（4）Multilabel ranking metrics（多标签排序度量）

- coverage_error：覆盖误差测量。
- label_ranking_average_precision_score：计算平均精度等级。
- label_ranking_loss：计算排序度量损失。

（5）Clustering metrics（聚类度量）

- sklearn.metrics 模块：对于分析结果的度量评估，采用以下两种评估形式。
- 监督：采用每个样本的真实类别。
- 无监督：没有采用每个样本的真实类别或测量模型本身的“质量”数据。相关函数如下。
- adjusted_mutual_info_score：两个簇的调整型互信息。
- adjusted_rand_score：概率型随机索引调整。
- calinski_harabaz_score：计算 Calinski 和 Harabaz 评分。
- completeness_score：给定真实值对簇标签的完整性度量。
- fowlkes_mallows_score：测量两组点集簇的相似性。
- homogeneity_completeness_v_measure：一次性计算同质性和完整性度量。
- homogeneity_score：给定真实值对簇标签的同质性度量。
- mutual_info_score：两个簇间的互信息。
- normalized_mutual_info_score：两个簇间的归一化互信息。
- silhouette_score：计算所有样品的平均轮廓系数。
- silhouette_samples：计算各样本的轮廓系数。
- v_measure_score：给定真实值的簇标签度量。

（6）Biclustering metrics（双聚类度量）

- metrics.consensus_score：双聚类的相似性得分。

（7）Pairwise metrics（配对度量）

配对度量，也称为两两度量。

- pairwise.additive_chi2_kernel：计算观测 X 和 Y 间的附加卡方核数据。
- pairwise.chi2_kernel：计算 X 和 Y 的卡方核指数。
- pairwise.distance_metrics：成对有效距离度量。
- pairwise.euclidean_distances：计算两两向量间的距离矩阵。
- pairwise.kernel_metrics：两两核的有效性度量。
- pairwise.linear_kernel：计算 X 和 Y 之间的线性核。
- pairwise.manhattan_distances：计算各向量间的 L1 距离。
- pairwise.pairwise_distances：计算向量数组 X 和可选数组 Y 的距离矩阵。
- pairwise.pairwise_kernels：计算数组 X 和可选数组 Y 之间的核。
- pairwise.polynomial_kernel：计算 X 和 Y 之间的多项式核。
- pairwise.rbf_kernel：计算 X 和 Y 之间的高斯（RBF）核。
- pairwise.sigmoid_kernel：计算 X 和 Y 之间的 Sigmoid 核。
- pairwise.cosine_similarity：计算 X 和 Y 之间的余弦相似度。
- pairwise.cosine_distances：计算 X 和 Y 之间的余弦距离。
- pairwise.laplacian_kernel：计算 X 和 Y 之间的拉普拉斯核。
- pairwise_distances：计算向量数组 X 和可选 Y 之间的距离矩阵。
- pairwise_distances_argmin：计算一个点集和一组点集的最小距离。
- pairwise_distances_argmin_min：计算一个点集和一组点集的最小距离。
- pairwise.paired_euclidean_distances：计算 X 和 Y 之间的欧氏距离。
- pairwise.paired_manhattan_distances：计算 X 和 Y 之间的 L1 距离。
- pairwise.paired_cosine_distances：计算 X 和 Y 之间的余弦距离。
- pairwise.paired_distances：计算 X 和 Y 之间的距离。

附录 B

量化分析常用指标

量化交易属于新兴产业，TA-Lib 金融软件包收录的函数虽然多，但大部分是传统技术指标，对于量化交易、量化分析领域有很大缺失。本节主要介绍量化领域新出现的一些技术指标。

量化分析本质上是一种风险和收益的综合平衡分析，其经典三大指标是夏普比率、詹森指数和特雷诺指数。

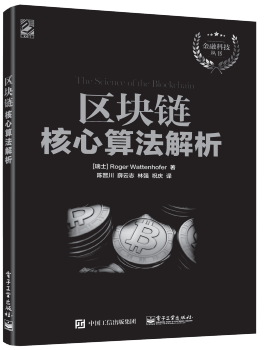
- 夏普比率（Sharpe Ratio），也称夏普指数，缩写是 SR。简单地说，它指的是投资回报与风险的比例。夏普比率代表投资人每多承担一分风险，就可以拿到几分报酬，若为正值，代表基金报酬率高过波动风险；若为负值，代表基金操作风险大于报酬率。通常，这个比例越高，投资组合越佳。
- 詹森指数（Jensen），又称为阿尔法值，是衡量基金超额收益大小的一种指标。这个指标综合考虑了基金收益与风险因素，比单纯的考虑基金收益大小要更科学。
- 特雷诺指数（Treynor Ratio），缩写是 TR，是每单位风险获得的风险溢价。特雷诺指数越大，单位风险溢价越高，绩效越好；相反，特雷诺指数越小，单位风险溢价越低，绩效越差。

在量化领域，其他常用的指标如下。

- 策略收益（Total Returns），策略实际收益额。
- 基准收益（Benchmark Returns），策略最低收益额。
- 基准收益率（Benchmark Yield），也称基准折现率，投资项目最低标准的收益水平，即选择特定的投资机会或投资方案必须达到的预期收益率。
- 阿尔法（Alpha），是高于预期收益率的超额收益率。阿尔法策略基于 CAPM 模型。传统阿尔法策略是在基金经理建立了 β 部位的头寸后，通过衍生品对冲 β 部位的风险，从而获得正的阿尔法收益。
- 按照 CAPM 模型的规定，β 系数是用以度量一项资产系统风险的指数，是用来衡量一种证券或一个投资组合相对总体市场的波动性（Volatility）的一种风险评估工具。
- 索提诺比率（Sortino Ratio），缩写是 SR，与夏普比率类似，所不同的是它区分了波动的好坏，因此在计算波动率时它所采用的不是标准差，而是下行标准差。这其中的隐含条件是投资组合的上涨（正回报率）符合投资人的需求，不应计入风险调整。和夏普比率类似，这个比率越高，表明基金承担相同单位下行风险能获得更高的超额回报率。索提诺比率可以看作夏普比率在衡量对冲基金/私募基金时的一种修正方式。
- 信息比率（Information Ratio），也称为绩效评估比率（Appraisal Ratio），缩写是 IR。信息比率是从主动管理的角度描述风险调整后的收益，它不同于夏普比率从绝对收益和总风险角度来描述。信息比率越大，说明所获得的超额收益越高，策略收益持续优于大盘的程度越高。计算方式是：将基金报酬率减去同类基金或者大盘报酬率（剩下的值为超额报酬），再除以该超额报酬的标准差。
- 策略波动率（Algorithm Volatility），是指回报率在策略执行期间内所表现出的波动率。

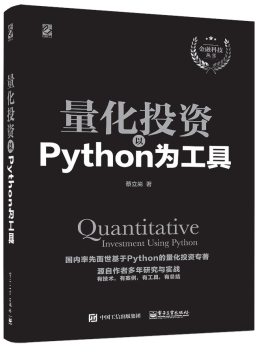
- 基准波动率（Benchmark Volatility），基准指数的波动范围，用来测量基准的风险性，波动越大代表基准风险越高。
- 下行风险（Downside Risk），是指由于市场环境变化，未来价格走势有可能低于分析师或投资者所预期的目标价位。下行风险是投资可能出现的最坏的情况，也是投资者可能需要承担的损失。
- 最大回撤率（Drawdown），是指该金融产品历史上一段时间的最大跌幅。最大回撤率用来描述买入产品后可能出现的最糟糕的情况。最大回撤是一个重要的风险指标，对于对冲基金和量化策略交易来说，该指标比波动率还重要。

金融科技丛书



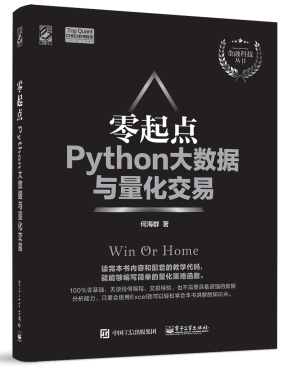
《区块链核心算法解析》
ISBN 978-7-121-31328-8
定价：59.00元

本书着眼于区块链的核心问题——拜占庭共识，针对不同的应用场景，介绍了适用的分布式共识算法。书中包含了很多算法及证明，深入剖析了共识算法的核心思想。



《量化投资：以Python为工具》
ISBN 978-7-121-30514-6
定价：99.00元

量化投资优质工具书！有技术、有案例，有工具、有总结，专家解读量化投资（以Python 语言），引导读者进入引人入胜的学术与实务领域！



《零起点Python大数据与量化交易》
ISBN 978-7-121-30659-4
定价：99.00元

无须任何编程、交易经验，也不需要具备超强的数据分析能力，只要会使用 Excel 就可以轻松学会本书讲解的知识点。读完本书内容和配套的教学代码，就能够编写简单的量化策略函数。

《零起点TensorFlow与量化交易》
ISBN 978-7-121-33584-6
定价：99.00元

一本让你轻松了解神经网络与量化投资的实战教程。图书 + 开发平台 + 成套的教学案例，系统讲解，逐步深入。

反侵权盗版声明