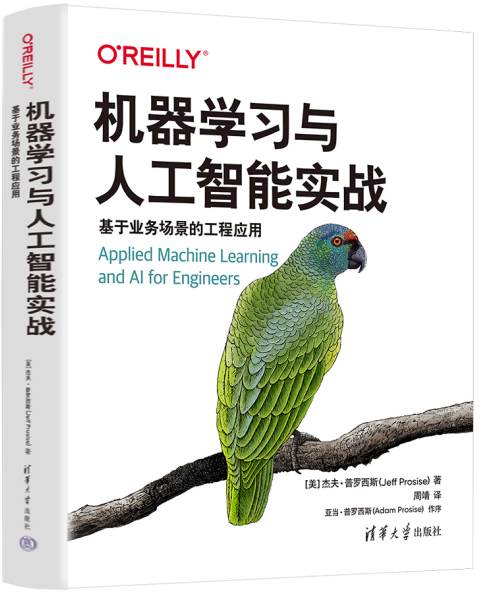
机器学习与人工智能实战：基于业务场景的工程应用

Applied Machine Learning and AI for Engineers——Solve Business Problems That Can't Be Solved Algorithmically

Jeff Prosise（杰夫·普罗西斯） 著

周 靖 译



中文试读版 1~5 章。翻译原稿，仅供参考，更多精彩内容，请购买正版。

购买链接：京东 淘宝

配套资源和试读下载： ys168 网盘>> 百度网盘>>

您还可以访问中文版博客，提交评论和勘误

清华大学出版社

本书简介

虽然许多 AI 入门指引其实都是变相的微积分教科书，但这本书大多避开了数学。相反，Jeff Prosise 帮助工程师和软件开发人员建立对 AI 的直观理解以解决商业问题。需要创建一个系统来检测雨林中非法伐木的声音，分析文本的情感，或者预测旋转机械装置的早期故障？这本实用的小书教给你将 AI 和机器学习应用于自己的公司所需的全部技能。

本书提供了 Prosise 在世界各地的公司和研究机构教授的 AI 和 ML 课程的例子和插图。没有华丽的辞藻，也没有可怕的方程式，有的只是工程师和软件开发人员的一个快速入门机会，并提供了完整的动手操作实例。

本书将帮助你：

- 了解何谓机器学习和深度学习，以及它们能做什么。
- 了解流行的学习算法是如何工作的，以及在什么情况下应用它们。
- 用 Scikit-Learn 在 Python 中构建机器学习模型，用 Keras 和 TensorFlow 构建神经网络。
- 训练回归、二分类和多分类模型，并对它们的效果进行打分。
- 构建面部识别模型和目标检测模型。
- 构建语言模型，响应自然语言查询，并将文本翻译成其他语言。
- 使用认知服务将 AI 集成到自己的应用中。

“如果想了解 AI 和 ML 幕后真正的工作原理，以及这些技术是如何发展和形成的，请务必看一下本书。”

——Todd Fine，Atmosera 公司首席战略官

“读这本书的时候，你会忍不住想去搞点什么东西出来。”

——Doug Turnure，Microsoft Azure 专家

作者简介

Jeff Prosise 是一名工程师，他酷爱向其他工程师和软件开发人员介绍 AI 和机器学习的奇迹。他是 Wintellect 公司的联合创始人，写过 9 本书和数百篇杂志文章，在 Microsoft 培训过数千名开发人员，并在全球最大规模的一些软件会议上发言。在另一段人生中，Jeff 在美国橡树岭国家实验室和劳伦斯利弗莫尔国家实验室从事高功率激光系统和聚变能源研究。在业余时间，他建造并飞行大型遥控喷气机，并经常到全球最好的一些地方去玩潜

水。2021 年他的公司被收购后，Jeff 在 Atmosera 公司担任首席学习官，帮助客户将 AI 集成到他们的产品中。

本书的誉美之辞

这本书是机器学习和 AI 算法的绝佳指南。内容既简洁又全面，附带的代码实例展示了如何将理论付诸于实践。

——Mark Russinovich，Microsoft Azure 首席技术官和技术院士

当 Jeff Prosise 对某件事情充满激情时（无论技术，他的黄颈亚马逊鹦鹉 Hawkeye，还是他建造和飞行遥控喷气机的爱好），你肯定想听一听他在说什么。他将这种激情与清晰的解释相结合，能比我所认识的任何人更好地交流和教授复杂的主题。他把你带到了一个学习和理解的私人旅程中。现在，Jeff 将这些技能带到了当前正在发生的机器学习和 AI 的“技术海啸”中（他的原话）。在这本新书中，他从基础开始建立你的理解，始终强调一种直观的方法，并将概念和解决方案与现实世界联系起来。如果想了解 AI 和 ML 幕后真正的工作原理，以及这些技术是如何发展和形成的，请务必看一下本书。

——Todd Fine，Atmosera 公司首席战略官

Jeff 将多年的 AI/ML 知识提炼成一本实用且易懂的指南，供各层次的从业人员参考。

——Ken Muse，4x Azure MVP 和高级 DevOps 架构师，GitHub

我初次接触机器学习时有这本书就好了。它对于 ML 工程师新手来说是一个很好的入门指引，对于那些更有经验的人来说也是一个很好的参考。它现在是我刷新和加强对各种 ML 技术及其适用性理解的首选资料。我喜欢 Jeff 用真实世界的例子和数据集来展示每个工具集和方法。如同神秘莫测的神经网络一样，我不知道 Jeff 的灵感是怎么来的，但就是那么有价值。

——Brent Rector，Amazon 首席技术项目经理；Wise Owl Consulting, LLC、Wise Owl Aviation Services, LLC 和 Rector Aviation Law PC 的创始人

我认识 Jeff 几十年了，他总是有本事将复杂的概念说得清清楚楚。本书的风格同样如此。大量例子、类比和彩色插图使初级和高级读者都很容易理解。

——Jeffrey Richter，Microsoft 软件架构师，《Windows 核心编程》和《CLR via C#》作者[①]

这本书填补了工程师和科学家的一个极其重要的空白，他们希望运用 AI 的力量来解决他们最具挑战性的数据分析问题。本书在技术深度和实际应用之间取得了一个恰到好处的平衡。通过工具和许多实例，读者能在自己的应用领域（包括商业、科学和其他数据丰富的领域）成为一名高效的 AI 从业人员。

——Shaun S. Gleason 博士，美国橡树岭国家实验室 Cyber Resilience and Intelligence 主任

本书有潜力成为 ML 和 AI 爱好者的首选读物。它与众不同的地方在于，在当今科技界快速采用机器学习的情况下，它所针对的问题直击人心！对于新手和专业人士来说，这是一本必读的书！

——Lipi Deepaakshi Patnaik，Zeta Suite 软件开发工程师

Jeff 总是能将深刻的技术概念包裹在伟大的故事中，这使学习过程变得轻松有趣。这或许是他迄今为止最好的一本书，也或许是和他最相关的主题。读这本书的时候，你会忍不住想去搞点什么东西出来。

——Doug Turnure，Microsoft Azure 专家

这是一本构建有用的机器学习系统的实用手册。它通俗易懂地讲解了如何应用最先进的 AI 算法来解决当代的商业问题。

——Brian Spiering，Metis 数据科学讲师

这本书对于想要了解机器学习的工程师和软件开发人员来说是一个完美的起点。它能帮你打下良好的 ML 基础，并迅速让你开始处理各种由数据而非算法驱动的问题。

——Manjeet Dahiya 博士，Ecom Express 副总裁兼 AI 和机器学习主管

[①] 这两本书的信息请参见 https://bookzhou.com/。

无论你是应用 ML 的新手，还是正在寻找一本参考书的老手，这本书都是首选的。它为所有主要的机器学习算法类别提供了最新和最全面的指南，并配有干净的代码实现，能真正加强你的理解。

——Goku Mohandas，Made With ML 创始人

这是为 AI/ML 初级到中级读者提供的一本书出色的参考书。本书建立了一个从传统 ML 到深度学习的舒适流程，还贴心地讲解了云端的 AI 实现，这使其成为任何热心读者或从业人员的完整端到端指南。

——Satyarth Praveen，美国劳伦斯伯克利国家实验室计算科学工程师

Jeff Prosise 是我遇到过的最好的老师之一，无论在什么平台上（课堂、博客、杂志文章、网络研讨会、书籍等），他都有一种特殊的才能，就是再复杂的话题，他也能让我们其他人理解。在这本书中，Jeff 不仅仅是简单地提供了对支撑机器学习的基本概念的清晰理解，还提供了简单易懂的例子，在当前环境中演示这些概念。他对机器学习中的各种主题进行了很好的介绍/概述，并对每个主题如何（以及应该）使用提供了明确的指导。Jeff 是为数不多能以一种易于吸收和应用的方式，将这一复杂的主题加以阐述的人之一。对于那些希望使用机器学习来提高自己技能的工程师和其他问题解决者来说，这是一本必读的书。

——Larry Clement，美国加州浸会大学计算机、软件和数据科学助理教授、前系主任，曾任波音公司 C-17 项目的高级工程师-科学家

多年来，Jeff Prosise 标志性的教学风格帮助了数以千计的开发人员，本书围绕复杂的主题展开，提供了易于理解的例子和教程。强烈推荐给那些希望将机器学习概念和技能纳入其工作范围的工程师。

——Vani Mandava，美国华盛顿大学电子科学研究所科学软件工程中心工程主管

献给我过去和现在的 Wintellect 家人。

目录

推荐序 **15**

译者序 **17**

前言 **18**

本书面向的读者 *18*

我为什么写这本书 *18*

运行本书的代码示例 *19*

本书导航 *21*

本书采用的约定 *21*

使用代码示例 *21*

如何联系我们 *22*

致谢 **23**

封面插图 **24**

第 1 部分 用 Scikit-Learn 进行机器学习 **25**

第 1 章 机器学习 **25**

1.1 什么是机器学习？ *25*

1.1.1 机器学习与人工智能 28

1.1.2 监督和无监督学习 30

1.2 使用 k-means 聚类算法的无监督学习 *30*

1.2.1 将 k-means 聚类算法应用于客户数据 34

1.2.2 使用两个以上的维度对客户进行细分 37

1.3 监督学习 *40*

1.3.1 k-近邻 42

1.3.2 使用 k-近邻对花卉进行分类 45

1.4 小结 *49*

第 2 章 回归模型 **50**

2.1 线性回归 *50*

2.2 决策树 *54*

2.3 随机森林 *57*

2.4 梯度提升机 *59*

2.5 支持向量机 *62*

2.6 回归模型的精度测量 *62*

2.7 使用回归来预测车费 *66*

2.8 小结 *70*

第 3 章 分类模型 **71**

3.1 逻辑回归 *71*

3.2 分类模型的准确率度量 *73*

3.3 分类数据 *77*

3.4 二分类 *79*

3.4.1 对泰坦尼克号乘客进行分类 79

3.4.2 检测信用卡欺诈 83

3.5 多分类 *88*

3.6 构建数字识别模型 *89*

3.7 小结 *93*

第 4 章 文本分类 **94**

4.1 准备用于分类的文本 *94*

4.2 情感分析 *98*

4.3 朴素贝叶斯 *101*

4.4 垃圾邮件过滤 *104*

4.5 推荐系统 *107*

4.5.1 余弦相似性 108

4.5.2 构建一个电影推荐系统 109

4.6 小结 *111*

第 5 章 支持向量机 **113**

5.1 支持向量机的工作原理 *113*

5.1.1 核 115

5.1.2 核技巧 116

5.2 超参数调整 *118*

5.3 数据归一化 *121*

5.4 管道化 *125*

5.5 使用 SVM 进行面部识别 *127*

5.6 小结 *132*

第 6 章 主成分分析 **133**

6.1 理解主成分分析 *133*

6.2 噪声过滤 *140*

6.3 数据匿名化 *142*

6.4 可视化高维数据 *144*

6.5 异常检测 *148*

6.5.1 使用 PCA 检测信用卡欺诈 149

6.5.2 使用 PCA 来预测轴承故障 152

6.5.3 多变量异常检测 157

6.6 小结 *158*

第 7 章 机器学习模型的操作化 159

7.1 从 Python 客户端使用 Python 模型 160

7.2 .pkl 文件的版本管理 162

7.3 从 C#客户端使用 Python 模型 163

7.4 容器化机器学习模型 165

7.5 使用 ONNX 来桥接不同的语言 166

7.6 用 ML.NET 在 C#中构建 ML 模型 169

7.6.1 用 ML.NET 进行情感分析 170

7.6.2 保存和加载 ML.NET 模型 173

7.7 为 Excel 添加机器学习功能 173

7.8 小结 177

第 2 部分 用 Keras 和 TensorFlow 进行深度学习 178

第 8 章 深度学习 179

8.1 了解神经网络 179

8.2 训练神经网络 184

8.3 小结 186

第 9 章 神经网络 187

9.1 用 Keras 和 TensorFlow 构建神经网络 187

9.1.1 设定神经网络的大小 191

9.1.2 使用神经网络来预测车费 192

9.2 用神经网络进行二分类 196

9.2.1 进行预测 197

9.2.2 训练神经网络来检测信用卡欺诈 199

9.3 用神经网络进行多分类 202

9.4 训练神经网络进行面部识别 205

9.5 Dropout *207*

9.6 保存和加载模型 *209*

9.7 Keras 回调 *210*

9.8 小结 *212*

第 10 章 用卷积神经网络进行图像分类 **214**

10.1 理解 CNN *215*

10.1.1 使用 Keras 和 TensorFlow 来构建 CNN 218

10.1.2 训练 CNN 来识别北极野生动物 222

10.2 预训练 CNN *226*

10.3 使用 ResNet50V2 对图像分类 *228*

10.4 转移学习 *230*

10.5 通过转移学习来识别北极野生动物 *232*

10.6 数据增强 *236*

10.6.1 用 ImageDataGenerator 进行图像增强 236

10.6.2 使用增强层进行图像增强 239

10.6.3 将图像增强应用于北极野生动物 239

10.7 全局池化 *242*

10.8 用 CNN 进行音频分类 *243*

10.9 小结 *249*

第 11 章 面部检测和识别 **251**

11.1 人脸检测 *251*

11.1.1 用 Viola-Jones 算法进行人脸检测 252

11.1.2 使用 Viola-Jones 的 OpenCV 实现 254

11.1.3 用卷积神经网络检测人脸 255

11.1.4 从照片中提取人脸 259
11.2 面部识别 261
11.2.1 将迁移学习应用于人脸识别 262
11.2.2 用任务特定的权重强化转移学习 265
11.2.3 ArcFace 268
11.3 综合运用：检测和识别照片中的人脸 268
11.4 处理未知人脸：闭集和开集分类 274
11.5 小结 275
第 12 章 目标检测 276
12.1 R-CNN 276
12.2 Mask R-CNN 279
12.3 YOLO 285
12.4 YOLOv3 和 Keras 287
12.5 自定义目标检测 291
12.5.1 用自定义视觉服务训练自定义目标检测模型 292
12.5.2 使用导出的模型 298
12.6 小结 300
第 13 章 自然语言处理 302
13.1 文本准备 302
13.2 词嵌入 305
13.3 文本分类 306
13.3.1 自动化文本矢量处理 309
13.3.2 在情感分析模型中使用 TextVectorization 310
13.3.3 将词序纳入预测的因素 312
13.3.4 循环神经网络(RNN) 313

13.3.5 使用预训练模型进行文本分类 315

13.4 神经机器翻译 316

13.4.1 LSTM 编码器-解码器 317

13.4.2 Transformer 编码器-解码器 319

13.4.3 构建基于 Transformer 的 NMT 模型 321

13.4.4 使用预训练模型来翻译文本 328

13.5 基于变换器的双向编码器(BERT) 329

13.5.1 构建基于 BERT 的答题系统 330

13.5.2 调优 BERT 以进行情感分析 334

13.6 小结 336

第 14 章 Azure 认知服务 338

14.1 Azure 认知服务简介 339

14.1.1 密钥和终结点 341

14.1.2 调用 Azure 认知服务 API 343

14.1.3 Azure 认知服务容器 346

14.2 计算机视觉服务 347

14.3 语言服务 355

14.4 翻译器服务 357

14.5 语音服务 359

14.6 集大成者：Contoso Travel 360

14.7 小结 364

推荐序

当你的父亲无限好奇你学的东西的时候，家就不再是一个安全港湾了😂。

我在 2018 年获得了分析学的硕士学位。通过研究生课程，我学会了如何利用机器学习、AI 和分析学来为企业增加价值，并为现实世界的挑战开发解决方案。我对这些事情充满热爱——这也是我和我父亲 Jeff Prosise 共同的热爱。事实上，我都不想告诉你有多少次他问是否能跟我一起上课（不是开玩笑），或者当我跑回父母家准备偷得半日闲的时候，他在厨房里就那么对我正在学的东西发起了新一轮的灵魂拷问。

如果你曾经在飞机上经历过乱流，而你旁边的人想跟你聊天来减压，说现代喷气式客机是多么了不起的工程，因为它们没有一个故障点，那么你就完全知道我当时的感受。

我们对数据和分析的热爱，发展成了对其提供的价值的共同热爱。使用本书概述的工具和技术，人们可以在不确定的情况下得出肯定的结论。数据科学使你能找出潜在的真相——发现真正发生的事情，以及它如何驱动行为和结果。在我们的信息经济中，使用分析法而不是直觉来窥探幕后奥秘的能力是一种值得赞赏的技能。对于在现代这种不确定性中航行的人们和机构来说，也是至关重要的。

同样重要的是将这些发现有效地传达给非技术性的受众，同时让他们对幕后发生的事情有深刻的技术理解。这种沟通能力是无法伪造的（在和父亲的厨房讨论中，我曾尝试过那么一两次）。

分析、AI 和机器学习并没有不能克服的技术或部署问题。相反，理解正在发生的事情和它们是如何工作的才是障碍，因为这种理解往往被笼罩在技术术语和行话中。这种进入障碍成了一种限制因素，阻碍了人们利用数据科学来解决问题和疑问。

这正是本书所要改变的：它扯下了外面那层朦胧的面纱，避免了行话，使这些工具和资源变得容易使用。

而且，说实话，我的父亲非常擅长以一种使学习过程变得毫不费力的方式，将本来令人望而生畏的技术课题写得和讲得清清楚楚。我一生都在见证这个“奇迹”。他把从 DOS 开始的所有课题都讲得通俗易懂，让今天商业世界的推动者和摇摆者都能理解。在过去几十年里，他使那些我无法理解的主题为几代程序员所接受。简而言之，他是最好的。

秘诀在于，在他的教学理念中，会问自己这样一个核心问题：“如果一个主题我从来没有听说过，但又很感兴趣，我希望对方怎么解释它？”。考虑到数据科学带来的独特挑战，而且这个领域的专业人士存在各种各样的观点，他的这种方法最终使这本书的可理解性达到了一个在别的很多地方都达不到的高度。

说真的，在复杂的 ML 和 AI 方面，你不可能得到比这本书更好的指导了。如果你已经很熟悉了，那么这本书也能磨练你的理解，就像它现在对我做的那样（说的就是你，第 13

章）。如果你对机器学习、AI、分析以及它们会为人类带来什么价值感兴趣，你就来对了。

无论你属于哪个阵营，都会对他所概述的主题有更深的了解，使你有能力使用这些工具，然后讲述你自己发现的故事。

通常，我最后会说一些像“我希望你喜欢这本书，并从中学到一些东西”这样的话，但在这种情况下，我不会那样说。相反，我不是*希望*——而是*知道*。我知道你会像我一样，从我父亲那里学到东西，并与他并肩作战。

Jeff Prosise 一直是我人生的榜样，而且我非常乐意与你们分享他的这一方面。或许——只是或许——他将点燃你对这些东西的热情，正如对我一样。

最后，我要和你们分享他在我出生那天抱着我对我说的话：“你来啦，孩子”（Welcome to the show, kid）。

——Adam Prosise，达美航空过程与创新专家

译者序

一个好的老师，会提出恰当的问题，激发学生的兴趣，然后引导学生一步一步解决问题。在此过程中，老师会做好所有必要的铺垫，埋下所有精彩的伏笔，知道学生在什么地方会遇到难点，然后以抽丝剥茧的方式，逐渐解开学生的疑惑。最终，学生会有“恍然大悟”的感觉，学起来就更有劲儿了。

简单地说，就是不要一开始设下一个很大的目标。而是把它分解为一系列小目标，让学生在很短的时间内一个接一个的达成这些小目标，并最终促成大目标的达成，从而打造出一个让人非常有成就感的学习过程。

作为几十年前就已封神的技术大牛，Jeff 的多本著作均沿用了这种精彩的写作风格。引用某个网友的话：“个人感觉大学读的很多计算机专业书籍都是叫 Jeff 或者 Jeffrey 的人写出来的。”实情确实如此，他们不仅仅是技术的大牛，还非常乐意分享，而且知道如何以正确的方式分享。如果还不清楚这几个 Jeff 的作品，在网上查一下《MFC 与 Windows 编程》或者《Windows 核心编程》就知道了。

本书的大目标是让工程师们快速掌握机器学习这一前沿技术。众所周知，机器学习的基础是数学，所以许多人一开始就会产生畏难情绪。Jeff 完全知道读者的想法，所以一开始就将数学从学习过程中剥离，通过区区几行代码，就调动起来了读者的学习兴趣，成功地使读者相信“我也能行”。有了动力之后，以后的一切就完全按部就班了。读者的瘾会越来越大，以至于最终不能自拔，兴趣盎然地看完整本书。

星转斗移，没想到 Jeff 居然有 20 多年没写新书了（Jeffrey Richter 也有 10 多年了）。时至 2023 年，Jeff 的新书居然给我们带来一个很大的惊喜。针对目前处于风口浪尖的机器学习主题，这本书完美诠释了什么叫“深入浅出”。

不再多说了，请赶快享受本书吧！

最后，我想感谢一下女儿周子衿（Ava Zhou），她从 Drexel 大学毕业后，刚好就进入了 Covid 疫情。在此期间，她一直陪伴着我。而且我发现了疫情期间的一个“特点”，那就是远程办公日渐红火。周子衿也正是这个时期获得了 VMware 公司的一个远程办公职位。她对这个职位甘之如饴，乐在其中。祝福她！

——周靖，2023 年 1 月于成都

前言

我这一生见证了三次伟大的技术革命：首先是个人电脑，然后是互联网，最后是智能手机。机器学习（ML）和 AI 的重要性与这三者比肩，并将对我们的生活产生同样深刻的影响。

有一天，我的信用卡公司打电话确认是不是我要购买一条 700 美元的项链，那天我首次对机器学习产生了兴趣。不是我，但我很好奇：他们怎么知道那可能不是我？我在世界各地都有使用信用卡，而且要说明的是，我*确实*时不时给我老婆买些好东西。信用卡公司一次都没有拒绝过合法的刷卡，但有几次他们正确标记了欺诈性的刷卡。在这之前的一次是巴西的某个家伙试图刷我的卡买机票。这一次不同的是，那家珠宝店离我家只有 2 英里。我试着想象，到底是怎样的一种算法，可以在商店里如此靠谱地检测到信用卡欺诈。没过多久，我就意识到有比单纯的算法更强大的东西在起作用。

事实证明，信用卡公司通过一个复杂的机器学习模型来运行每一笔交易，这个模型非常善于检测欺诈。那一刻改变了我的生活。这是一个很有说服力的例子，证明了 ML 和 AI 如何使世界变得更美好。此外，了解 ML 如何实时分析信用卡交易，挑出不良交易，同时对合法交易放行，成为我必须攀登的一座山峰。

本书面向的读者

最近，我接到一家制造公司工程主管的电话。他是这样说的：“直到上周，我都不知道ML 和 AI 是啥意思。现在，CEO 给我布置了任务，要弄清楚它们如何改善我们的业务，还要在竞争对手领先之前做到这一点。我现在要从头开始。你能帮忙吗？”

下一通电话来自一家对使用机器学习检测税务欺诈和洗钱感兴趣的政府承包公司。那里的团队对机器学习理论相当精通，但不知道如何最好地去建立他们需要的模型。

各地的专业人士都意识到，ML 和 AI 代表了一场技术海啸，他们正试图在浪潮冲垮自己之前站到浪尖上。本书就是为这些人准备的，他们包括工程师、软件开发人员、IT 经理以及其他相关人士。他们想要建立对 ML 和 AI 的实际理解，并应用这些知识来解决以前难以甚至不可能解决的问题。本书试图传授一种*直观*的理解，只有在必要时才会诉诸于方程式。无论你以前有没有听过，但真的不必成为微积分或线性代数的专家，就能建立一个系统来识别照片中的目标（object），将英语翻译成法语，或者揭露贩毒者和偷税漏税者。

我为什么写这本书

每个作者的内心都住着一个小小的精灵，它说他们可以用别人没有的方式讲故事。我在 30 多年前写了第一本计算机书，20 多年前写了最后一本，本来并不打算再写的。但现在我有一个故事要讲。这是一个重要的故事——每个工程师和软件开发人员都应该听听。我对别

人讲这个故事的方式不完全满意，所以我写了这本书，我希望在当初学习这门手艺时能有这本书。它从基础知识开始，带领你攀登 ML 和 AI 一座又一座的高峰。最后，你会明白信用卡公司如何检测欺诈行为，飞机公司如何通过机器学习对喷气发动机进行预测性维护，自动驾驶汽车如何看它周围的世界，谷歌翻译如何在不同语言之间翻译文本，以及面部识别系统如何工作。此外，你还能自己建立类似的系统或者使用现有的系统，将 AI 集成到自己编写的应用程序中。

今天，最先进的机器学习模型需要在配备了图形处理单元（GPU）或张量处理单元（TPU）的计算机上进行训练，而这通常会花费大量时间和费用。本书的一个特点是提供了一些例子，它们能在没有配备 GPU 的普通 PC 或笔记本电脑上构建。在讲到识别照片中目标的计算机视觉模型时，我会解释这种模型是如何工作的，以及它们如何在 GPU 集群上用数以百万计的图像进行训练。但是，我随后就会向你展示如何使用一种称为*迁移学习*（transfer learning）的技术来重用现有模型以解决领域特有的问题，并在普通的笔记本电脑上训练它们。

本书在很大程度上借鉴了我在世界各地的公司和研究机构教授的课程和研讨会。我喜欢教学，因为我喜欢看到电灯泡亮起[①]。我经常会在 ML 和 AI 的课程开始时说：我不是来教书的，我是来改变你的人生的！希望你的人生会比读这本书之前有一点点不同，有一点点好。

运行本书的代码示例

工程师最好通过亲自*做*来学习，而不仅仅是*读*。本书包含大量代码示例，你可以通过运行这些例子来巩固从每一章学到的知识。大多数是用 Python 编写的，并使用了流行的开源库，例如 Scikit-Learn、Keras 和 TensorFlow。所有这些都可以在我专门建立的一个 GitHub 公共存储库中找到（https://oreil.ly/applied-machine-learning-code）。这是所有代码示例的唯一事实来源，因为我随时能更新它。

一些机器学习平台允许在不写代码的情况下构建和训练模型。但是，为了了解这些平台能做什么以及具体如何做，最好的办法还是写代码。Python 是一种简单的编程语言。它很容易学习。今天的工程师必须能自如地写代码。你可以通过本书的例子来学习 Python，如果已经会了 Python（以及更常规意义的编程），那么你其实已经领先别人一步了。

要在你的 PC 或笔记本电脑上运行我的例子，需要安装 64 位的 Python 3.7 或更高版本。可以从 Python.org 下载一个 Python 运行时，或者可以安装一个 Python 发行版，例如 Anaconda（https://oreil.ly/4NCqN）。另外，还需要确保安装了以下软件包和它们的依赖项：

- 用于构建机器学习模型的 Scikit-Learn 和 TensorFlow

[①] 译注：即“电灯泡时刻”，形容突然灵光闪现、受到启发或者意识到什么。

- 用于数据处理和可视化的 Pandas、Matplotlib 和 Seaborn
- 用于图像处理的 OpenCV 和 Pillow
- 用于调用 REST API 和构建网络服务的 Flask 和 Requests
- 用于开放神经网络交换（Open Neural Network Exchange，ONNX）模型的 Sklearn-onnx 和 Onnxruntime
- 用于从音频文件生成频谱图的 Librosa
- 用于构建面部识别系统的 MTCNN 和 Keras-vggface
- 用于构建自然语言处理（Natural Language Processing，NLP）模型的 KerasNLP、Transformers、Datasets 和 PyTorch
- 用于调用 Azure 认知服务（Azure Cognitive Services）的 Azure-cognitiveservices-vision-computervision、Azure-ai-textanalytics 和 Azure-cognitiveservices-speech。

可以用 `pip install` 命令来安装其中大多数包。如果安装了 Anaconda，其中许多包已经安装好了，可以用 `conda install` 命令或类似的命令安装其余的包。

至于环境，最好是用虚拟 Python 环境来防止安装的这些包与其他包发生冲突。如果不熟悉虚拟环境，可以在 Python.org 上阅读关于它们的信息。如果使用的是 Anaconda，那么虚拟环境已经就位了。

我的大部分代码示例是为 Jupyter 笔记本（Jupyter notebooks）构建的，它为编写和执行 Python 代码提供了一个交互平台。数据科学界经常使用“笔记本”来探索数据和训练机器学习模型。可以通过安装 Notebook（https://oreil.ly/ZWQyG）或 JupyterLab（https://oreil.ly/5A3Ia）等包在本地运行 Jupyter 笔记本，也可以使用 Google Colab（https://oreil.ly/RdRBa）等云托管环境。Colab 的一个优点是不必在自己的电脑上安装任何东西，就连 Python 都不用。而且在我的例子要求 GPU 的罕见情况下，Colab 也能为你提供 GPU。

Python 开发环境的设置和维护是出了名的棘手，尤其是在 Windows 电脑上。如果不想创建这样的环境，或者如果尝试过但没成功，那么只需下载一个东西就可以了。我在 Docker 容器镜像（https://oreil.ly/wzEbA）中打包了一个完整的开发环境，适合运行本书中的所有例子。如果你的电脑上安装了 Docker 引擎（https://oreil.ly/XO5GD），那么可以用以下命令启动容器：

```
docker run -it -p 8888:8888 jeffpro/applied-machine-learning:latest
```

然后，在浏览器中访问输出结果中显示的 URL，就会进入一个完整的 Jupyter 环境，里面有我所有的代码示例和运行它们所需的一切。它们存储在一个名为 Applied-Machine-Learning 的文件夹中，该文件夹是从同名 GitHub 存储库中克隆的。不过，使用容器有一个缺点，即所做的更改默认不会保存。补救办法之一是在 `docker` 命令中使用`-v` 开关，从而绑定到一个本地目录。更多信息请参考 Docker 文档中的“绑定挂载”（https://oreil.ly/7wgda）。

本书导航

本书分为两部分：

- 第 1 部分（第 1 章~第 7 章）讲授机器学习的基础知识，并介绍了流行的学习算法，例如逻辑回归和梯度提升。
- 第 2 部分（第 8 章~第 14 章）讨论深度学习。作为如今人工智能的代名词，它使用深度神经网络使数学模型和数据拟合。

强烈建议在阅读本书时动手练习。这样会对这些材料有更深刻的理解，而且你肯定会开始思考如何修改我的代码，玩一玩“假如…？”（what if?）游戏。

本书采用的约定

本书采用了以下排版约定：

等宽字体：代码清单和段落中出现的程序元素（例如变量、函数、数据库、数据类型、环境变量、语句和关键字等）使用等宽代码字体。例如，`from skl2onnx import convert_sklearn`。

等宽字体加粗：要由用户亲自输入的命令或其他字面值使用加粗的等宽字体。例如，请输入 **`abc`**。

 这个元素代表一个提示或建议。

 这个元素代表一个常规的注意事项。

 这个元素代表一个警告或提醒。

使用代码示例

如前所述，本书的补充材料（代码实例、练习等）可从 https://oreil.ly/applied-machine-learning-code 下载。

如果有技术问题或者在使用代码示例时遇到问题，请电邮至 bookquestions@oreilly.com。

本书是为了帮助你完成自己的工作。一般来说，如果书中提供了示例代码，那么你可以在自己的程序和文档中使用。不需要联系我们以获得许可，除非要复制代码的很大一部分。例如，写一个用到了本书几处代码的程序不需要许可。销售或分发 O'Reilly 书中的示例则

需要。引用本书正文和示例代码来回答一个问题不需要许可。将本书大量示例代码纳入你的产品文档，则需要许可。

我们感谢，但一般不强求署名。如果要署名，通常应该包括标题、作者、出版商和 ISBN。例如，Applied Machine Learning and AI for Engineers by Jeff Prosise（O'Reilly）。Copyright 2023 Jeff Prosise, 978-1-492-09805-8。

如果你觉得对代码示例的使用超出了合理使用或上述许可的范畴，请随时与我们联系：permissions@oreilly.com。

如何联系我们

请将关于本书的意见和问题寄给出版商：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938（美国或加拿大）
707-829-0515（国际或本地）
707-829-0104（传真）

我们为这本书建立了一个网页，其中提供了勘误表、例子和其他相关信息：

https://oreil.ly/applied-machine-learning

有对本书的评论或者技术问题，请电邮至 bookquestions@oreilly.com 。

要获取我们的书籍和课程的新闻和资讯，请访问 https://oreilly.com。

在 LinkedIn 上找到我们：https://linkedin.com/company/oreilly-media

在 Twitter 上关注我们：https://twitter.com/oreillymedia

在 YouTube 上观看我们：https://youtube.com/oreillymedia

致谢

写作和出版一本书没有一个靠谱的团队可不行。它始于作者，但在上架之前，审稿人、策划编辑、文字编辑、美术师和生产人员都要经手一遍。

我要感谢几位朋友和家人，我一边写，他们一边帮我审阅了几章，并提供了建设性的反馈。他们是工程师、数学家、数据分析师和教授，而且恰好都是我认识的最聪明的人之一：Larry Clement，Manjeet Dahiya，Tom Marshall，Don Meyer，Goku Mohandas，Ken Muse，Lipi Deepaakshi Patnaik，Charles Petzold，Abby Prosise，Adam Prosise，Jeffrey Richter，Bruce Schecter，Vishwesh Ravi Shrimali，Brian Spiering 和 Ron Sumida。

感谢 O'Reilly 的团队，他们将我的文字变得更优美，将我的草图变成艺术品。这包括 Jill Leonard，Audrey Doyle，Gregory Hyman，David Futato，Karen Montgomery 和 Nicole Butterfield。尤其要感谢 Jon Hassell，在听到我对这本书的想法后，他马上就说：“赶紧的。”多年来，我曾有幸与一些伟大的出版团队合作，但这个团队真的是最棒的！

最后，感谢 Lori——我的妻子、旅行伙伴和这 40 多年来的死党。没有你，这本书是搞不定的。唉，我保证这真的是我写的最后一本书了！

封面插图

本书封面插图中的动物是喜庆亚马逊鹦鹉（Amazona festiva），也称为喜庆鹦鹉，生活在巴西、哥伦比亚、厄瓜多尔、秘鲁和玻利维亚等南美国家的热带森林、林地和沿海红树林。它们很少在远离水的地方被发现。

喜庆鹦鹉是一种色彩非常鲜艳——甚至有点……喜庆——的中等大小的鸟类。其羽毛主要是醒目的绿色，在翅膀边缘略微变成黄色。它们的脸上有各种各样的颜色，包括红色、蓝色，有时还有黄色或橙色。

喜庆鹦鹉是一个高度社会化的物种，通常成对或小群出现。大群的鸟儿经常在夜间聚集在一起，进行集体栖息，或围绕着一个局部的食物来源，并以极其聒噪而闻名。它们喜欢吃水果，如芒果和桃树，还有浆果、坚果、种子、花和叶芽作为补充食物。

虽然在其森林栖息地基本保持完整的地方仍然比较常见，但由于持续的森林砍伐和预测的栖息地减少，喜庆鹦鹉已被国际自然保护联盟归类为“近危”（Near Threatened，NT）。O'Reilly 书籍封面上的许多动物都处于濒危状态；所有这些动物对世界都很重要。

封面插图由 Karen Montgomery 创作，基于 Wood 的《Illustrated Natural History》一书中的一张复古雕刻线图。本书（英文版）封面字体使用的是 Gilroy Semibold 和 Guardian Sans。文本字体使用 Adobe Minion Pro；标题字体使用 Adobe Myriad Condensed；代码字体则使用 Dalton Maag 的 Ubuntu Mono。

第 1 部分 用 Scikit-Learn 进行机器学习

第 1 章 机器学习

机器学习（Machine Learning，ML）使计算机能解决短短几年前还难以解决的问题，从而拓展了可能的边界。从欺诈检测和医疗诊断，到产品推荐和能“观察”四周的汽车，机器学习每天都在影响我们的生活。在你读这本书的时候，科学家们正在使用机器学习来解开人类基因组的秘密。当我们有一天能治愈癌症时，也应该感谢机器学习使之成为可能。

机器学习是革命性的，因为它提供了一种替代用算法解决问题的方法。给定一个配方，或者说*算法*，编写一个密码加密或计算每月按揭付款的应用程序并不难。只需为算法编码，提供输入，然后接收输出即可。但是，写代码来判断照片中包含的是一只猫还是一只狗，则完全是两码事。虽然也可以尝试用算法来做，但很快就会遇到无法识别其中的猫或狗的一张照片。

机器学习采取了一种不同的方法将输入转化为输出。它不依赖你来实现一个算法，而是检查一个包含输入和输出的数据集，并学习如何在一种称为训练（training）的过程中生成自己的输出。在幕后，称为学习算法（learning algorithms）的一些特殊算法将数学模型与数据相匹配，并编订（codify）输入和输出数据之间的关系。一旦经过训练，模型可以接受新的输入，并产生与训练数据一致的输出。

所以，要使用机器学习来区分猫和狗，你不需要编码一个区分猫和狗的算法。相反，用猫和狗的照片训练一个机器学习模型即可。最终是否会成功，则取决于所用的学习算法以及训练数据的质量和数量。

为了成为一名机器学习工程师，需要熟悉各种学习算法，并培养一种直觉，知道何时应该使用一种算法，何时应该使用另一种算法。这种直觉来自于经验以及对机器学习如何使数学模型拟合到数据的理解。本章是这一旅程的第一步。它首先概述了机器学习和最常见的机器学习模型类型，最后介绍了两种流行的学习算法，并用它们来构建简单但功能齐全的模型。

1.1 什么是机器学习？

从存在意义的角度说，机器学习（Machine Learning，ML）是一种在数字中寻找模式并利用这些模式来进行预测的手段。ML 使我们能用 1 和 0 的行或序列来训练一个模型，并能从数据中学习，从而在给定一个新的序列时，模型能预测结果会是什么。*学习*使 ML 能找到一个模式来预测未来的输出，也是“机器学习”中“学习”一词的来源。

作为一个例子，考虑图 1-1 展示的 1 和 0 的表格。第 4 列的每个数字都是根据同一行中前面的三个数字来确定的。那么，缺少的那个数字是什么？

0	1	0	0
1	1	0	1
1	1	1	1
0	0	0	0
0	1	1	

图 1-1 由 0 和 1 构成的简单数据集

一个可能的解是，对于某一行，如果前三列包含的 0 比 1 多，那么第四列就包含 0。如果前三列包含的 1 比 0 多，那么答案就是 1。根据这个逻辑，空框中应该填入 1。数据科学家将包含答案的列（图中的红色列）称为标签列（label column）。其余列则称为特征列（feature columns）。对于一个预测模型来说，它的目标是在特征列的行中找到模式，使其能预测标签是什么。

如果所有数据集都这么简单，那么根本不需要机器学习了。但是，现实世界的数据集更大、更复杂。数据集完全可能包含数以百万计的行和数以千计的列，这在机器学习中很常见。在这种情况下应该怎么办？例如，如果是像图 1-2 那样的一个数据集呢？

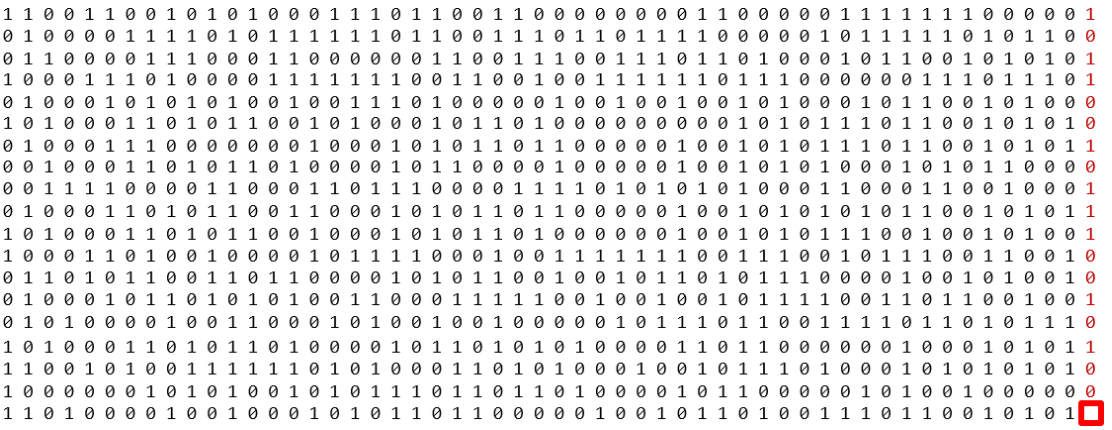


图 1-2. 一个更复杂的数据集

但凡是人，都很难检查这个数据集，并想出一套规则来预测红框里应该是 0 还是 1（不，这不是数 1 和 0 那么简单）。试想一下，如果数据集真的有几百万行和几千列，那难度该会有多大！

这就是机器学习的目的：在包含海量数字数据集中寻找模式。有 100 行还是 100 万行并不重要。甚至许多时候越多越好，因为 100 行可能无法提供足够的样本来识别模式。

我们说机器学习是通过对数字集合的模式进行数学建模以解决问题，这种说法并没有过分简化。大多数问题真的可以简化为一个数字集合。例如，今天 ML 的常见应用之一是情感分析（sentiment analysis）：检查一个文本样本，如影评或网站上的一条评价，并为其分配 0 代表负面情感（例如，“食物很普通，服务很糟糕”），或者分配 1 代表正面情感（例如，“食物和服务都很好，还会再来”）。有的评价可能是混合的。例如，“汉堡不错，但薯条太湿了”。在这种情况下，我们使用“标签为 1 的概率”作为情感分数。一条非常负面的评价可能得 0.1 分，而一条非常正面的评价可能得 0.9 分（换言之，有 90%的机率它表达的是正面情感）。

情感分析器和其他文本处理模型经常在像图 1-3 这样的数据集上进行训练。其中，每个文本样本都对应一行，每列都对应文本语料库（给定数据集中的所有词）中的一个词。在像这样一个典型的数据集中，可能包含数百万行以及 2 万或更多列。

meeting	excellent	food	meds	colleague	prescription	foolish	work	human	today	meeting	excellent	gather	lunch	mark	dog	corpus	text	document	yesterday	fuel	spreadsheet	budget	resources	virtual	mask	class	training	therefore	gasoline	consult	walk	upper	careful	brain	intimate	left	roadmap	content	airline	test	travel	fun	procedure	code	mulch	ban	least	morsel	ordinary	fossil	cream	…	sentiment
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	…	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	…	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…	1
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	…	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0

图 1-3 供情感分析的数据集

在每一行的标签列（最后一列）中，都用 0 代表负面情感，或者用 1 代表正面情感。每一行中都有词计数，即一个给定的词在单个样本中出现的次数。该数据集是稀疏的，这意味着它大部分是 0，偶尔会有一个非零的数字混入。但是，机器学习并不关心数字的构成。如果存在可供利用的模式，能判断下一个样本表达的是正面还是负面情感，它就会找出这些模式。垃圾邮件过滤器使用的就是这样的数据集，标签列中的 1 和 0 分别表示垃圾邮件和非垃圾邮件。这使现代垃圾邮件过滤器能达到惊人的准确率。另外，随着时间的推移，这些模型会变得越来越聪明，因为它们会用越来越多的真实电子邮件进行训练。

情感分析是文本分类（text classification）任务的一个例子，即分析一个文本样本，并将其

分类为正面或负面。事实证明，机器学习也善于进行图像分类（image classification）。图像分类的一个简单的例子是查看猫和狗的照片，并将每张照片分类为猫图（0）或者狗图（1）。图像分类在现实世界中的应用包括标记从装配线上下来的有缺陷的零件，识别自动驾驶汽车视野中的物体，以及识别照片中的人脸。

图像分类模型是用如图 1-4 所示的数据集来训练的。其中，每一行代表一幅图像，每一列代表一个像素值。假定一个数据集包含 1 百万张宽 200 像素、高 200 像素的图像，那么它总共就有 1 百万行和 4 万列，也就是 4 百亿个数字。如果图像是彩色而不是灰度的，那么就是 1200 亿个数字。记住，在彩色图像中，像素值由三个而不是一个数字组成。标签列包含的数字代表相应图像所属的类别或类别。在本例中，也就是照片中出现的是哪个人的脸。0 代表 Gerhard Schroeder，1 代表 George W. Bush，以此类推。

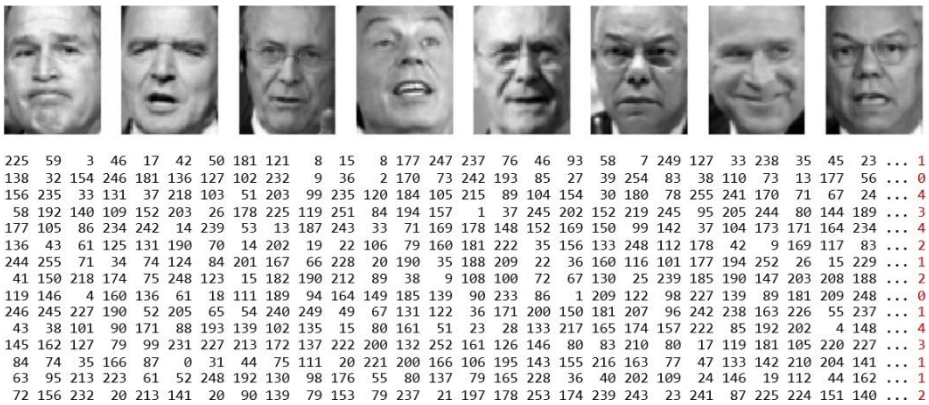


图 1-4 用于图像分类的数据集

这些面部图像来自一个著名的公共数据集，名为 Labeled Faces in the Wild（https://oreil.ly/YVIv2），简称 LFW。它是无数个带标签的数据集中的一个，这些数据集在多处发布供公众使用。如果有带标签的数据集可供使用，那么机器学习一点都不难——这些数据集是其他人（通常是研究生）辛辛苦苦花费大量时间用 1 和 0 来标记的。在现实世界中，工程师有时会将大部分时间花在生成这些数据集上。Kaggle.com 是较受欢迎的公共数据集存储库之一，它提供了许多有用的数据集，还经常举办比赛，让新晋的 ML 从业人员测试他们的技能。

1.1.1 机器学习与人工智能

今天，机器学习（ML）和人工智能（AI）这两个术语几乎可以互换着使用，但事实上，每个术语都有特定的含义，如图 1-5 所示。

从技术上说，机器学习是人工智能的一个子集，它不仅包括机器学习模型，还包括其他类型的模型，例如专家系统（expert systems，根据定义的规则做出决策的系统）和强化学习

系统（reinforcement learning systems），它们通过奖励正面的结果而惩罚负面的结果来学习行为。强化学习系统的一个例子是 AlphaGo（https://oreil.ly/uLwpd），它是第一个击败人类职业围棋选手的计算机程序。它在历史棋局上进行训练，并自行学习获胜策略。

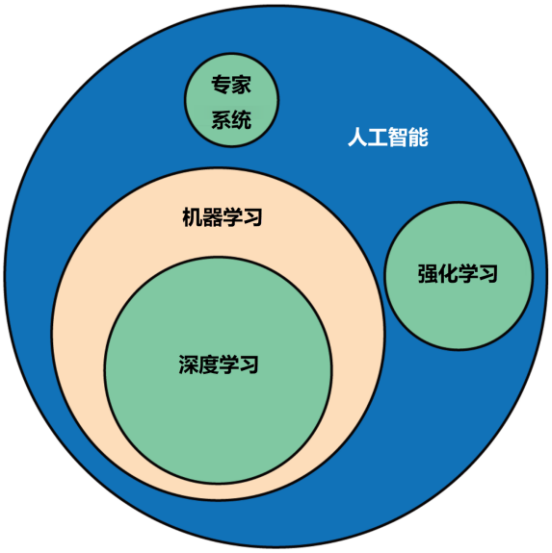


图 1-5 机器学习、深度学习和人工智能之间的关系

今天大多数人在说 AI 的时候，其实都在说深度学习，它是机器学习的一个子集。深度学习（Deep Learning）是用神经网络进行的机器学习。虽然一些形式的深度学习不涉及神经网络（例如深度波尔兹曼机），但今天绝大多数深度学习都涉及神经网络。因此，ML 模型可以分为传统模型和深度学习模型，前者使用学习算法对数据中的模式进行建模，后者使用神经网络进行建模。

AI 简史

近年来，ML 和 AI 的受欢迎程度激增。其实，20 世纪 80 年代就已经掀起了 AI 的热潮。当时，人们普遍以为计算机很快就能模仿人类的思维。但是，这股热潮很快就衰退了。如此风平浪静地过了几十年，在 2010 年以前，AI 很少成为新闻。然后，一件奇怪的事情发生了。

感谢 NVIDIA 等公司生产的图形处理单元（GPU）（https://oreil.ly/tiYd0），研究人员终于拥有了训练先进神经网络所需的引擎。这导致了技术水平的提升，人们重拾热情。这进而导致了更多的资金投入，催生了进一步的进步。突然之间，AI 又成了热门话题。自 20 世纪 50 年代以来，神经网络一直存在（至少在理论上），但研究人员缺乏算力，无法在大型数

据集上训练它们。如今，任何人都能购买一个 GPU 或者在云中建立一个 GPU 集群。AI 比以往任何时候都发展得更快。随着这种进步，现在可以在软件中做一些工程师在十年前只能梦想的事情。

随着时间的推移，数据科学家已经设计出擅长某些任务的特殊类型的神经网络，包括涉及计算机视觉的任务——例如，从图像中提取信息——以及涉及人类语言的任务，例如将英语翻译成法语。我们将从第 8 章开始深入研究神经网络。届时，你将具体了解深度学习如何将机器学习提升到一个新的高度。

1.1.2 监督和无监督学习

大多数 ML 模型都属于以下两大类别中的一类：监督学习（supervised learning）模型和无监督学习（unsupervised learning）模型。监督学习模型的目的是为了进行预测。我们用带标签的数据来训练它们，这样它们就能接受未来的输入并预测标签会是什么。今天使用的大多数 ML 模型都是监督学习模型。一个很好的例子是美国邮政总局对邮件进行分类时，用来将手写的邮政编码转化为计算机可以识别的数字的模型。另一个例子是信用卡公司对刷卡进行授权的模型。

相比之下，无监督学习模型不需要带标签的数据。它们的目的是提供对现有数据的见解（insights），或者将数据归类，并对未来的输入进行相应的分类。无监督学习的一个典型例子是检查客户从你的公司购买的产品，以及是哪些客户购买的，以确定哪些客户可能对你准备推出的新产品最感兴趣，然后建立一个针对这些客户的营销活动。

垃圾邮件过滤器是一种监督学习模型。它需要带标签的数据。根据收入、信用分和购买历史对客户进行细分的模型是一种无监督学习模型，它所消耗的数据不必加上标签。为了帮助理解这种区别，本章其余部分将更详细地探讨有监督和无监督学习。

1.2 使用 k-means 聚类算法的无监督学习

无监督学习经常采用一种称为聚类算法（clustering）的技术。聚类算法的目的是按相似性对数据进行分组。最流行的聚类算法是 k-means 聚类算法（https://oreil.ly/8duYJ），它需要获取 *n* 个数据样本，并将它们分成 *m* 个聚类，其中 *m* 由你指定。

分组是通过一个迭代过程进行的，它为每个聚类计算一个中心点，并根据样本与聚类中心点的接近程度将其分配给聚类。如果一个特定的样本到聚类 1 中心点的距离是 2.0，而同一个样本到聚类 2 中心点的距离是 3.0，那么该样本将被分配给聚类 1。在图 1-6 中，200 个样本被松散排列在三个聚类中。左边的图显示的是原始的、未分组的样本。右边的图显示了聚类中心点（红点）和按聚类着色的样本。

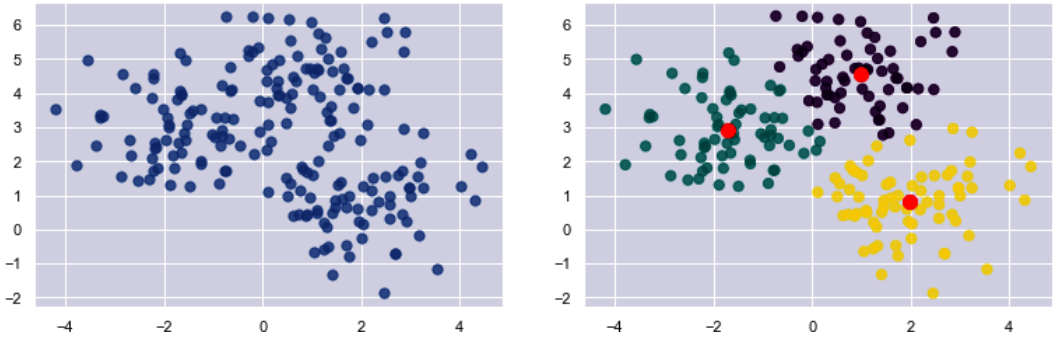


图 1-6 使用 k-means 聚类算法进行分组的数据点

那么，如何编码一个实现 k-means 聚类算法的无监督学习模型？最简单的方法是使用世界上最流行的机器学习库：Scikit-Learn（https://oreil.ly/bSQT2）。它是免费和开源的，而且是用 Python 写的。文档写得非常好，如果有任何问题，简单的一次谷歌搜索就有可能找到答案。在本书前半部分，我将使用 Scikit 完成大部分的例子。本书“前言”描述了如何安装 Scikit 并配置你的计算机来运行这些例子（或者使用一个 Docker 容器）。所以，如果还没有配置你的开发环境，现在是时候了。

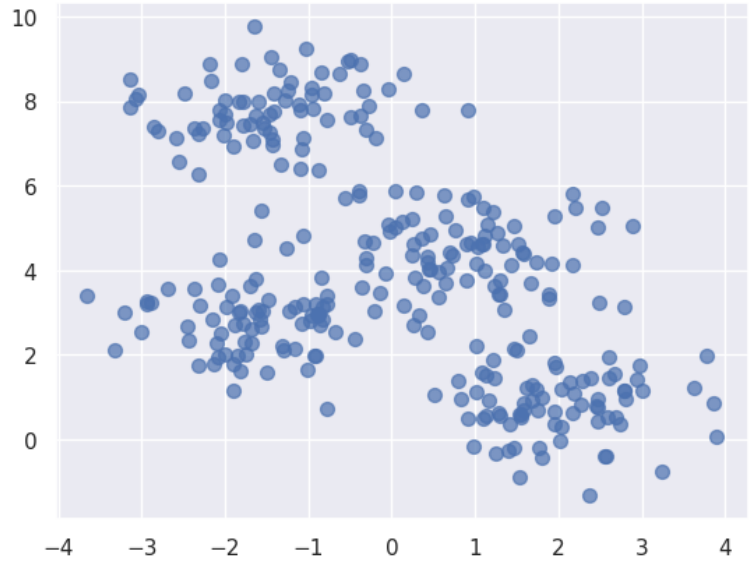
要使用 k-means 聚类算法，首先要创建一个新的 Jupyter 笔记本，并将以下语句粘贴到第一个单元格：

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
```

运行该单元格，再在下个单元格中运行以下代码，从而生成一系列半随机的 x 和 y 坐标对。这段代码使用 Scikit 的 `make_blobs` 函数（https://oreil.ly/h5sIB）来生成坐标对，并使用 Matplotlib 的 `scatter` 函数（https://oreil.ly/bnmNw）来绘制它们。

```
from sklearn.datasets import make_blobs

points, cluster_indexes = make_blobs(n_samples=300, centers=4,
                                     cluster_std=0.8, random_state=0)

x = points[:, 0]
y = points[:, 1]

plt.scatter(x, y, s=50, alpha=0.7)
```

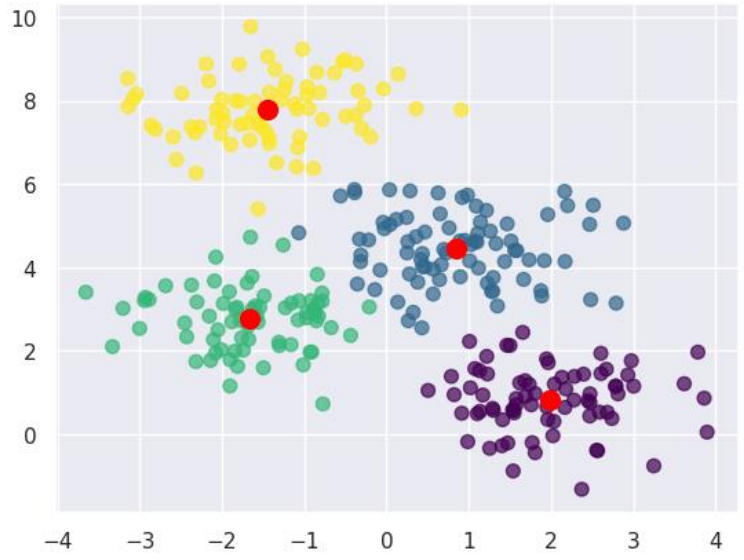
你的输出应该与我的相同，这要归功于 `random_state` 参数，它为 `make_blobs` 内部使用的随机数发生器提供了种子。



接着，使用 k-means 聚类算法将这些坐标对分为四组。然后将聚类中心点渲染成红色，并按聚类对数据点进行着色。在这里，最累的活是 Scikit 的 `KMeans` 类（https://oreil.ly/wgm9z）干的。一旦它适配了坐标对，就可以从 `KMeans` 的 `cluster_centers_`属性中获得中心点的位置。

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=4, random_state=0)
kmeans.fit(points)
predicted_cluster_indexes = kmeans.predict(points)

plt.scatter(x, y, c=predicted_cluster_indexes, s=50, alpha=0.7, cmap='viridis')

centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='red', s=100)
```

下面是结果：

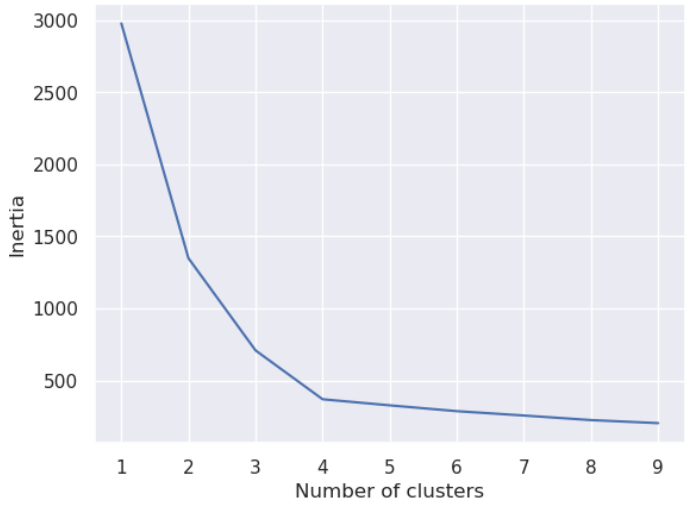


试着将 n_clusters 设为其他值，例如 3 和 5，看看在聚类数量不同的时候，这些点是如何分组的。这就引出了一个问题：你怎么知道正确的聚类数量是多少？如果仅仅是看图，那么是不容易获得答案的。如果数据有三个以上的维度，无论如何都无法绘制。

选择正确数量的一个方法是使用肘部方法（elbow method），它将从 KMeans.inertia_获得的惯性（inertias，数据点到最近聚类中心点的平方距离和）作为聚类数量的函数来绘制。以这种方式绘制惯性，可以找出曲线中弯曲幅度最大的“肘部”。

```
inertias = []

for i in range(1, 10):
    kmeans = KMeans(n_clusters=i, random_state=0)
    kmeans.fit(points)
    inertias.append(kmeans.inertia_)

plt.plot(range(1, 10), inertias)
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
```

在这个例子中，似乎 4 就是正确的聚类数量。



在现实中，肘部可能没有那么明显。但没有关系，因为以不同的方式对数据进行聚类，有时能获得其他方式无法获得的见解（insights）。

1.2.1 将 k-means 聚类算法应用于客户数据

现在，让我们运用 k-means 聚类算法来解决一个真实的问题：对客户进行细分，以确定哪些客户可以作为促销目标，从而增加其购买活动。要使用的数据集是一个名为 customs.csv 的客户细分数据集样本。首先，在你的笔记本所在的文件夹中创建一个名为 Data 的子目录，下载 customers.csv（https://oreil.ly/Md86Y），并将其复制到 Data 子目录。然后使用以下代码将数据集加载到一个 Pandas `DataFrame` 中并显示前 5 行：

```
import pandas as pd

customers = pd.read_csv('Data/customers.csv')
customers.head()
```

从输出结果可知，该数据集包含 5 列，其中两列描述了客户的年收入（Annual Income）和消费分数（Spending Score）。后者是一个 0~100 的数字。数字越大，表明该客户过去在你的公司的消费越多。

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	Male	19	15	39
1	2	Male	21	15	81
2	3	Female	20	16	6
3	4	Female	23	16	77
4	5	Female	31	17	40

现在使用以下代码来绘制年收入和消费分数。

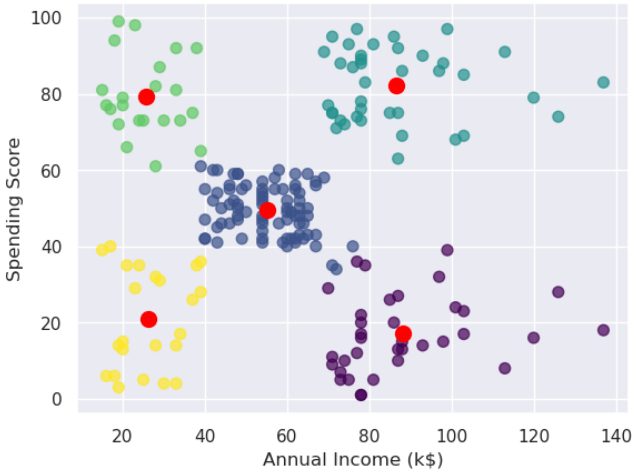
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

points = customers.iloc[:, 3:5].values
x = points[:, 0]
y = points[:, 1]

plt.scatter(x, y, s=50, alpha=0.7)
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score')
```

从结果来看，数据点大致分为五个聚类。

使用以下代码将客户分成五个聚类并进行着色：

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=5, random_state=0)
kmeans.fit(points)
predicted_cluster_indexes = kmeans.predict(points)

plt.scatter(x, y, c=predicted_cluster_indexes, s=50, alpha=0.7, cmap='viridis')
plt.xlabel('Annual Income (k$)')
plt.ylabel('Spending Score')

centers = kmeans.cluster_centers_
plt.scatter(centers[:, 0], centers[:, 1], c='red', s=100)
```

下面是结果：



图中右下角的客户可能是我们的目标客户，能通过促销来增加他们的消费。为什么？因为他们的收入很高，但消费分数很低。使用以下语句创建 `DataFrame` 的一个副本，并在其中添加一个名为 `Cluster` 的列来包含聚类的索引。

```
df = customers.copy()
df['Cluster'] = kmeans.predict(points)
df.head()
```

下面是输出：

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)	Cluster
0	1	Male	19	15	39	4
1	2	Male	21	15	81	3
2	3	Female	20	16	6	4
3	4	Female	23	16	77	3
4	5	Female	31	17	40	4

现在使用以下代码来输出那些收入高但消费分数低的客户的 ID：

```
import numpy as np

# 针对收入高但消费分数低的客户，获取其聚类索引
cluster = kmeans.predict(np.array([[120, 20]]))[0]

# 筛选 DataFrame，只包含该聚类中的客户
clustered_df = df[df['Cluster'] == cluster]

# 显示客户IDs
clustered_df['CustomerID'].values
```

一旦获得了客户 ID，就可以很容易地从客户数据库中提取姓名和电子邮件地址：

```
array([125, 129, 131, 135, 137, 139, 141, 145, 147, 149, 151, 153, 155,
       157, 159, 161, 163, 165, 167, 169, 171, 173, 175, 177, 179, 181,
       183, 185, 187, 189, 191, 193, 195, 197, 199], dtype=int64)
```

这里的关键在于，我们用聚类算法将客户按年收入和消费分数分组。一旦客户以这种方式分组，枚举每个聚类中的客户就是一件很简单的事情。

1.2.2 使用两个以上的维度对客户进行细分

上个例子很简单，因为只用到了两个变量：年收入和消费分数。即使没有机器学习的帮助，你也能做到这一点。但是，现在让我们再次对客户进行细分，这次使用除客户 ID 以外的所有东西。首先，用 1 和 0 替换 Gender（性别）列中的"Male"（男）和"Female"（女）字符串，这个过程称为标签编码（label encoding）。这是必要的，因为机器学习只能处理数字数据。

```
from sklearn.preprocessing import LabelEncoder

df = customers.copy()
encoder = LabelEncoder()
df['Gender'] = encoder.fit_transform(df['Gender'])
df.head()
```

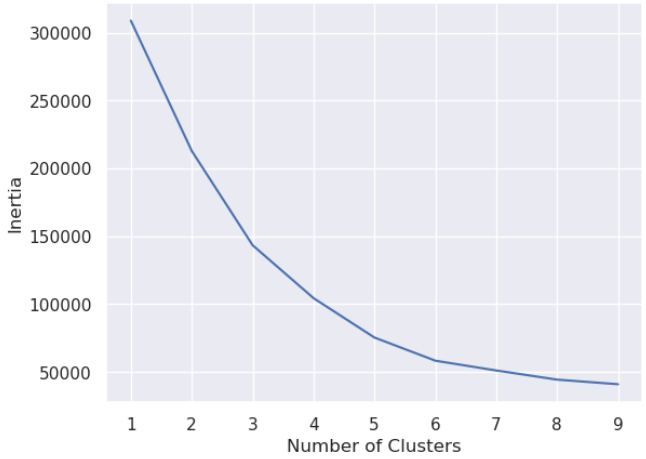
Gender 列现在包含 1 和 0：

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)
0	1	1	19	15	39
1	2	1	21	15	81
2	3	0	20	16	6
3	4	0	23	16	77
4	5	0	31	17	40

提取性别、年龄、年收入和消费分数列。然后使用肘部方法来确定基于这些特征的最佳聚类数量：

```
points = df.iloc[:, 1:5].values
inertias = []

for i in range(1, 10):
    kmeans = KMeans(n_clusters=i, random_state=0)
    kmeans.fit(points)
    inertias.append(kmeans.inertia_)

plt.plot(range(1, 10), inertias)
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia')
```

这次的肘部没有那么明显，但 5 似乎是一个合理的数字：



将客户细分成五个聚类，并添加一个名为 `Cluster` 的列，在其中包含客户被分配到的聚类（0~4）的索引：

```
kmeans = KMeans(n_clusters=5, random_state=0)
kmeans.fit(points)

df['Cluster'] = kmeans.predict(points)
df.head()
```

下面是输出：

	CustomerID	Gender	Age	Annual Income (k$)	Spending Score (1-100)	Cluster
0	1	1	19	15	39	0
1	2	1	21	15	81	4
2	3	0	20	16	6	0
3	4	0	23	16	77	4
4	5	0	31	17	40	0

现在，我们已经有了每个客户的聚类编号，但这意味着什么呢？不能像上个例子绘制年收入和消费分数那样，在一个二维图中绘制性别、年龄、年收入和消费分数。但是，可以获得每个聚类中的这些数值距离聚类中心点的平均值。以下代码新建一个 `DataFrame`，在其中包含平均年龄、平均收入等列，然后在一个表格中显示结果：

```
results = pd.DataFrame(columns = ['Cluster', 'Average Age', 'Average Income',
                                  'Average Spending Index', 'Number of Females',
                                  'Number of Males'])

for i, center in enumerate(kmeans.cluster_centers_):
    age = center[1]    # 当前聚类的平均年龄
    income = center[2] # 当前聚类的平均收入
    spend = center[3]  # 当前聚类的平均消费分数

    gdf = df[df['Cluster'] == i]
    females = gdf[gdf['Gender'] == 0].shape[0]
    males = gdf[gdf['Gender'] == 1].shape[0]

    results.loc[i] = ([i, age, income, spend, females, males])

results.head()
```

下面是输出：

	Cluster	Average Age	Average Income	Average Spending Index	Number of Females	Number of Males
0	0.0	45.217391	26.304348	20.913043	14.0	9.0
1	1.0	32.692308	86.538462	82.128205	21.0	18.0
2	2.0	43.088608	55.291139	49.569620	46.0	33.0
3	3.0	40.666667	87.750000	17.583333	17.0	19.0
4	4.0	25.521739	26.304348	78.565217	14.0	9.0

在此基础上，如果打算针对收入高但消费分数低的顾客进行促销，你会选择哪一组顾客（哪个聚类）？以男性或女性为目标有关系吗？从这一点出发，如果目标是创建一个为消费分数高的客户提供奖励的忠诚度计划，但又想优先考虑可能会成为长期忠诚客户的年轻人，那么应该怎么做？会以哪个聚类为目标？

聚类所揭示的更有趣的见解是，消费最多的是一些收入不高的年轻人（平均年龄=25.5）。这些顾客更有可能是女性而不是男性。如果你在开公司，并想更好地了解你所服务的客户，那么所有这些都是有用的信息。

k-means 或许是最常用的聚类算法，但并不是唯一。其他还有层次聚类（agglomerative clustering）（https://oreil.ly/zQpxZ），它以分层的方式对数据点进行聚类。还有 DBSCAN（https://oreil.ly/TanDh），它的全称是“具有噪声的基于密度的空间聚类方法”（density-based spatial clustering of applications with noise）。DBSCAN 不需要事先指定聚类数量。它还能识别出它所识别的聚类之外的点，这对于检测离群点（outliers）——与其他数据不一致的异常数据点——很有用。Scikit-Learn 在其 `AgglomerativeClustering`（https://oreil.ly/7CztS）和 `DBSCAN`（https://oreil.ly/D13gs）类中提供了这两种算法的实现。

那么，是否有真正的公司使用聚类算法从客户数据获取见解？事实上，他们真的这么做了。在研究生阶段，我的儿子，现在是达美航空的数据分析师，在一家宠物用品公司实习。他用 k-means 聚类算法推定，网站转化率（流量转化为销量）之所以很低，主要原因是自从他们打开网站后，销售人员首次与之接触的间隔时间太长了。因此，他的雇主在销售工作流中引入了额外的自动化措施，以确保人们一旦访问网站就会快速得到接待。这就是无监督学习的作用。这也是公司使用机器学习来改善其业务过程的一个绝佳的例子。

1.3 监督学习

无监督学习是机器学习的一个重要分支，但当大多数人听到“机器学习”这个词时，他们想到的其实是监督学习（或称“有监督学习”）。之前说过，监督学习模型可以做出预测。例如，它们能预测一笔信用卡交易是否具有欺诈性，或者一趟航班能否准时到达。另外，它们是用带标签的数据进行训练的。

监督学习模型有两种类型：回归模型（regression models）和分类模型（classification models）。回归模型的目的是预测一个数值结果，例如房屋售价或照片中一个人的年龄。相反，分类模型是从训练数据所定义的有限类别中预测一个类别。例如，信用卡交易是合法的还是欺诈性的，或者一个手写的数字代表什么。前者是一个二分类（binary classification）模型，因为只有两种可能的结果，即交易合法与否。后者是一个多分类（multiclass classification）的例子。由于阿拉伯数字系统中有 10 个数字（0~9），所以一个手写的数字可能代表 10 个类别。

图 1-7 展示了两种监督学习模型。在左边，目标是输入一个 x 并预测 y 会是什么。在右边，目标是输入一个 x 和一个 y，然后预测这个点对应的类别：三角形或椭圆。

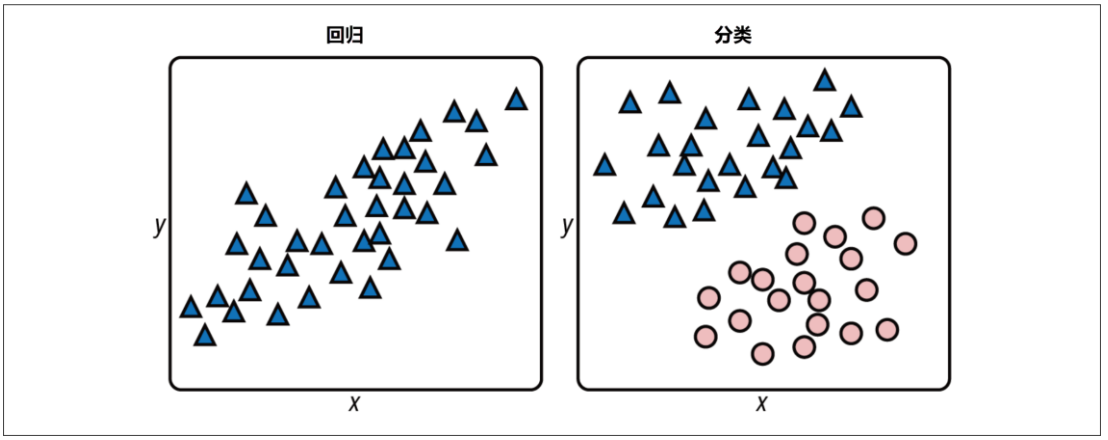


图 1-7 回归与分类

在这两种情况下，将机器学习应用于该问题的目的是建立一个预测模型。但我们不需要自己建立这个模型。相反，使用带标签的数据来训练一个机器学习模型，让它为你设计一个数学模型。

对于这些数据集，可以很容易地建立数学模型，而不必求助于机器学习。对于回归模型，可以画一条穿过数据点的线，并使用该线的方程式为给定的 x 预测 y（图 1-8）。对于分类模型，则可以画一条线，将三角形和椭圆彻底分开——数据科学家称之为分类边界（classification boundary）或决策边界（decision boundary）。然后，通过判断一个新的点是在线的上方还是下方，从而预测其所属的类别。线上方的点是三角形，线下方的点则被归类为椭圆。

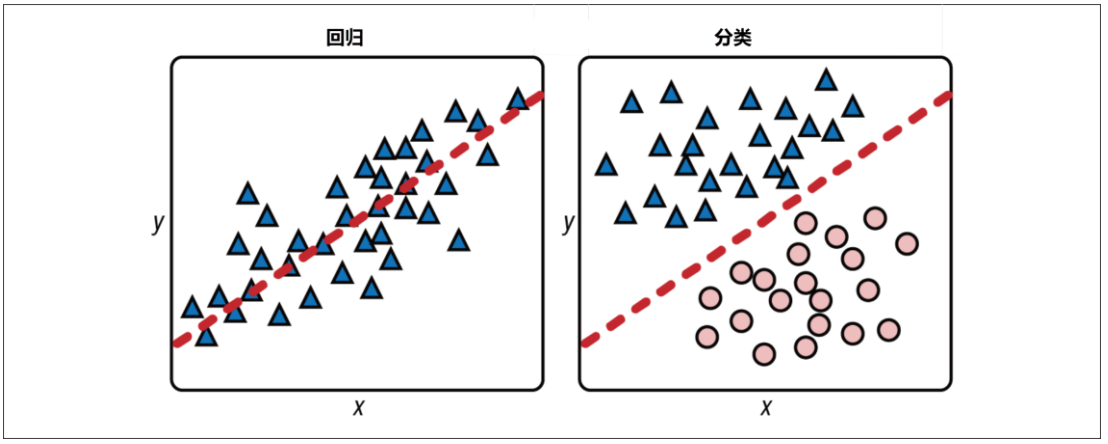


图 1-8 回归线和线性分隔边界

但是，现实世界中的数据集很少如此有序。它们看起来更像是图 1-9 那样。在左边，没有一条线能将 x 和 y 值联系起来。在右边，没有一条线能将不同类别彻底分开。因此，我们的目标是建立一个最好的模型。这意味着需要选择能生成最准确模型的学习算法。

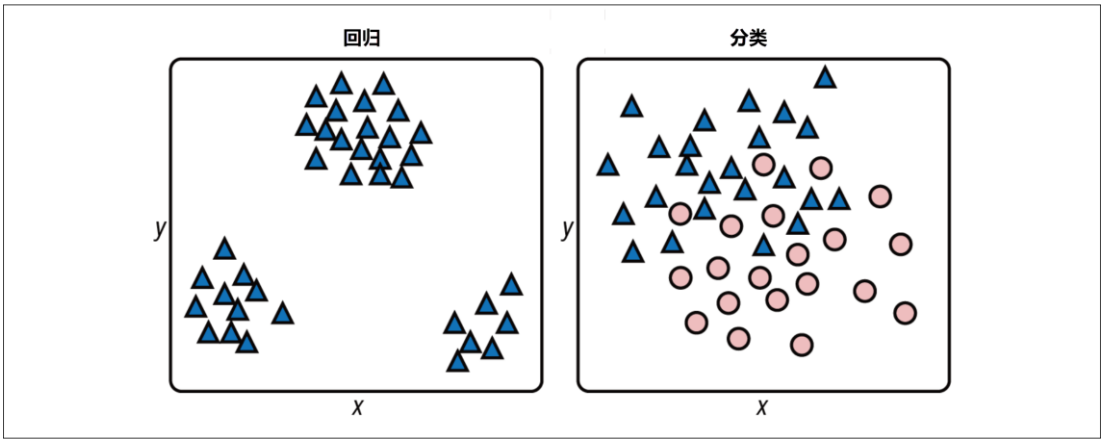


图 1-9 现实世界的数据集

目前有许多监督学习算法，其中包括线性回归、随机森林、梯度提升机（Gradient-Boosting Machine，GBM）和支持向量机（Support Vector Machine，SVM）。其中许多，但不是全部，都可以用于回归和分类。即便是经验丰富的数据科学家，也经常需要通过实验来确定哪种学习算法能生成最准确的模型。这些和其他学习算法将在后续各章介绍。

1.3.1 k-近邻

最简单的监督学习算法之一是 k-近邻（k-nearest neighbors）（https://oreil.ly/dPhKi）。它的思路是，给定一组数据点，可以通过检查离一个新点最近的点来预测其标签。在简单的回归问题中，每个数据点的特征由 x 和 y 坐标决定。这意味着给定一个 x，为了预测它的 y 是什么，可以查找 x 最近的 n 个点，然后平均它们的 y。如果是分类问题，那么要找到与想要预测的类别的点最近的 n 个点，并选择出现次数最多的类别。例如，假定 $n=5$，而且在它的近邻中，包括三个三角形和两个椭圆，那么答案就是三角形，如图 1-10 所示。

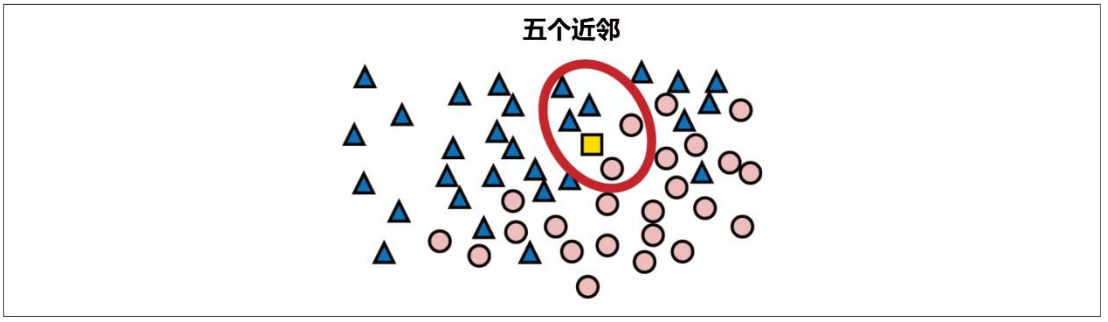


图 1-10 用 k-近邻进行分类

下面是一个涉及回归的例子。假设有 20 个数据点，描述了程序员按工作年限每年赚多少钱。在图 1-11 中，x 轴代表工作年限，y 轴则代表年收入。我们的目标是预测有 10 年从业经验的程序员有多少年薪。换言之，在 $x=10$ 的情况下，我们想预测 y 应该是多少。

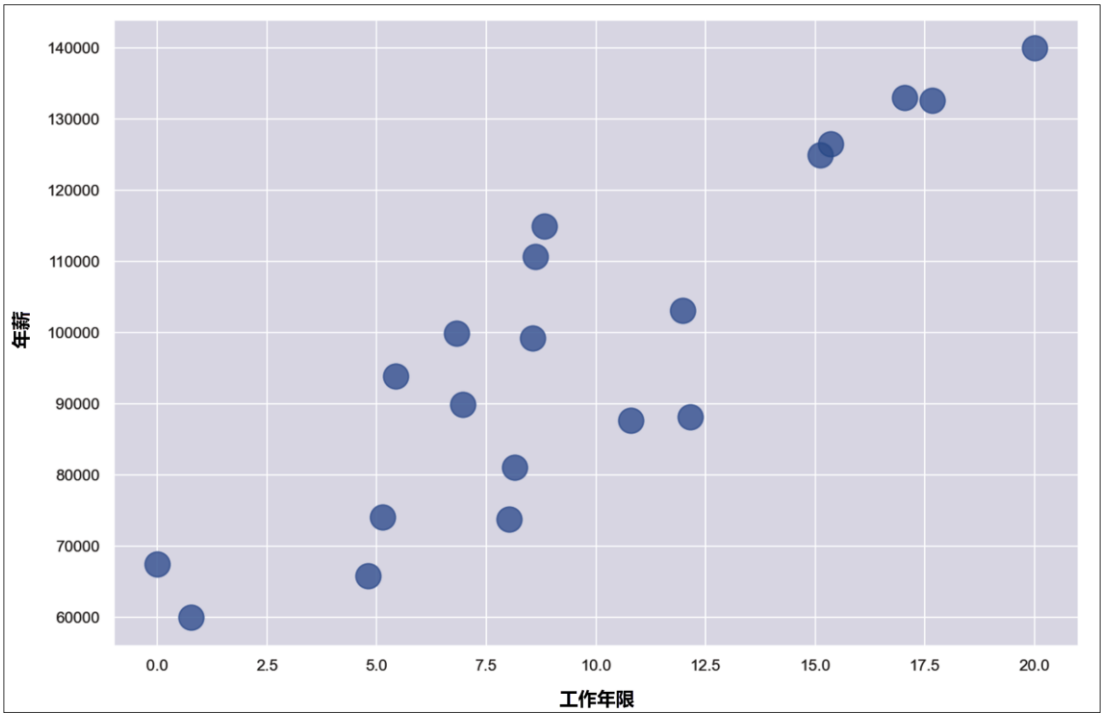


图 1-11 程序员年薪（美元）与工作年限的关系

如图 1-12 所示，在 n = 10 的情况下应用 k-近邻算法，可以找出 10 个橙色的近邻——它们的 x 坐标最接近 $x = 10$。这些点的 y 坐标的平均值为 94 838。因此，在 $n = 10$ 的情况下，k-近邻算法预测有 10 年经验的程序员的年薪为 94 838 美元，即图中的红点。

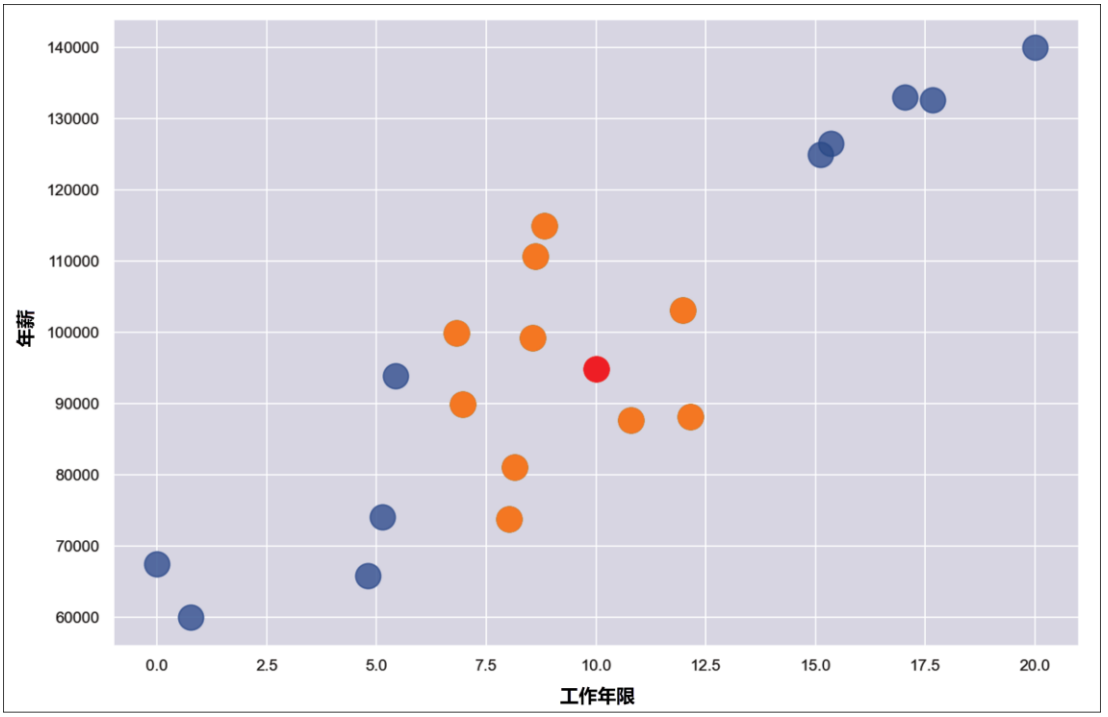


图 1-12 用 k-近邻和 $n = 10$ 进行回归

在 k-近邻算法中使用的 n 值经常会对结果产生影响。图 1-13 展示了 $n = 5$ 时的同一个求解过程。但这次的答案略有不同，因为五个近邻的平均 y 值为 98 713。

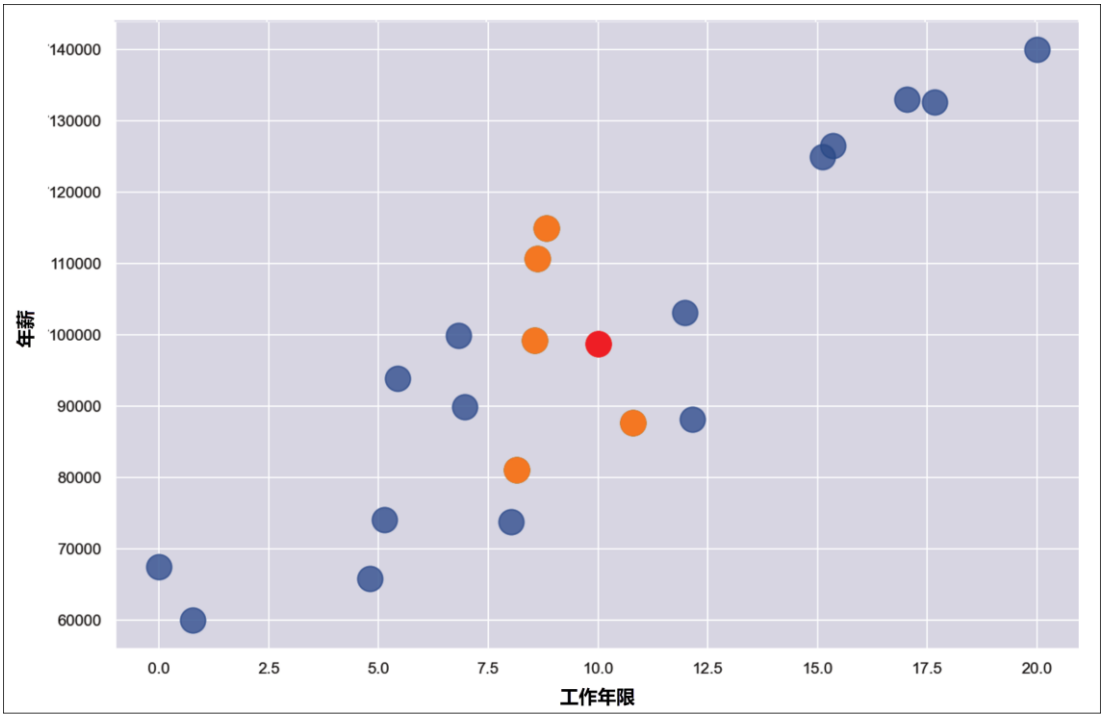


图 1-13 用 k-近邻和 $n = 5$ 进行回归

现实情况会更复杂，因为虽然数据集只有一个标签列，但它可能有几个特征列——不仅仅有 x，还有 x_1、x_2、x_3 等。可以很容易地计算 n 维空间中的距离，但为了确定一个点的近邻，可以选择几种测量距离的方法，其中包括欧氏距离、曼哈顿距离和闵可夫斯基距离（https://oreil.ly/36K7A）。甚至可以使用权重，使较近的点比较远的点对结果的贡献更大。而且，可以选择给定半径内的所有邻居，而不是找出最近的 n 个邻居，这种技术被称为半径邻居（radius neighbors）。不过，无论数据集有多少维，用什么方法来测量距离，也无论选择 n 个近邻还是选择指定半径内的所有邻居，背后的原理都是一样的：找出与目标点相似的数据点，用它们对目标点进行回归或分类。

1.3.2 使用 k-近邻对花卉进行分类

Scikit-Learn 包括名为 `KNeighborsRegressor`（https://oreil.ly/feXD0）和 `KNeighborsClassifier`（https://oreil.ly/8o7Uv）的类，帮助你使用 k-近邻学习算法训练回归和分类模型。它还包括名为 `RadiusNeighborsRegressor`（https://oreil.ly/FP3wH）和 `RadiusNeighborsClassifier`（https://oreil.ly/g1TfE）的类，接受半径而不是邻居数量作为参数。下面来看看一个使用 `KNeighborsClassifier` 对花卉进行分类的例子，它使用的是著名的鸢（yuān）尾花数据集（https://oreil.ly/lpXFR）。该数据集包括 150 个样本，每个样本代表三个鸢尾花品种中的一个。每一行都包含四个测量值——萼片长度、萼片宽度、花

瓣长度和花瓣宽度，单位都是厘米——外加一个品种标签：0 代表 Setosa 鸢尾花，1 代表 Versicolor 鸢尾花，2 代表 Virginica 鸢尾花。图 1-14 展示了每个品种的示例，并说明了花瓣（Petal）和萼片（Sepal）的区别。

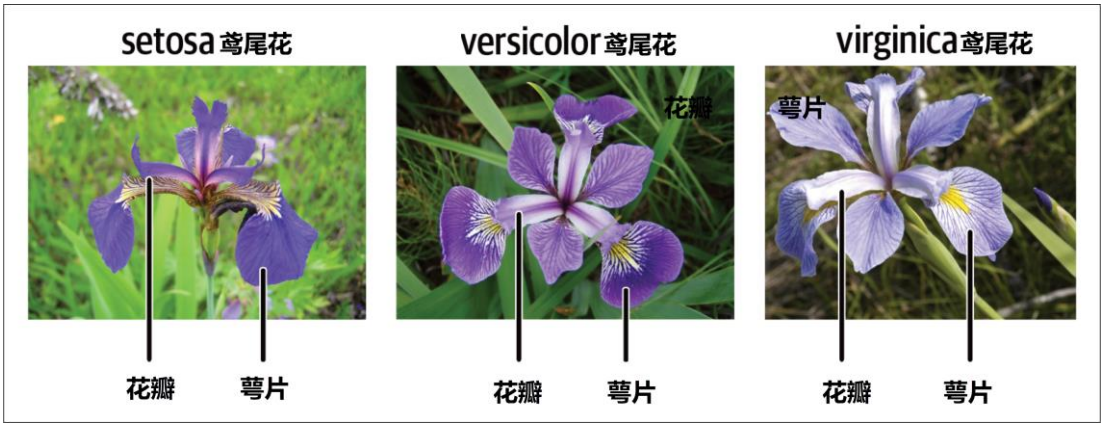


图 1-14 鸢尾花数据集

为了训练一个机器学习模型来区分基于萼片和花瓣测量值的鸢尾花品种，首先在 Jupyter 笔记本中运行以下代码来加载数据集，添加一个包含类名的列，并显示前五行。

```
import pandas as pd
from sklearn.datasets import load_iris

iris = load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['class'] = iris.target
df['class name'] = iris.target_names[iris['target']]
df.head()
```

Iris 数据集是 Scikit 自带的几个样本数据集之一（https://oreil.ly/BMQYL）。这就是为什么能直接调用 Scikit 的 `load_iris` 函数（https://oreil.ly/NmpjR）来加载它，而不必从外部文件读取的原因。下面是代码的输出：

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	class	class name
0	5.1	3.5	1.4	0.2	0	setosa
1	4.9	3.0	1.4	0.2	0	setosa
2	4.7	3.2	1.3	0.2	0	setosa
3	4.6	3.1	1.5	0.2	0	setosa
4	5.0	3.6	1.4	0.2	0	setosa

在基于数据训练机器学习模型之前，需要先将数据集分成两个数据集：一个用于训练，一个用于测试。这很重要，因为如果用以前的训练数据来测试，那么根本无法判断模型预测得有多准确。

幸好，Scikit 的 `train_test_split` 函数（https://oreil.ly/6TuKC）可以很容易地使用你指定的占比来分割一个数据集。以下语句进行 80/20 分割，即 80%的行用于训练，20%的行用于测试：

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=0.2, random_state=0)
```

现在，`x_train` 和 `y_train` 就有 120 行随机选择的测量值和标签，而 `x_test` 和 `y_test` 有剩余的 30 行。虽然对于如此小的数据集来说，进行 80/20 分割是惯例，但并没有规则说必须进行 80/20 分割。用于训练的数据越多，模型就越准确。(并非严格如此，但一般来说，我们总是希望获得尽可能多的训练数据)。而用于测试的数据越多，在衡量模型的准确率时就越有信心。对于小的数据集来说，80/20 是一个合理的起点。

下一步是训练一个机器学习模型。多亏了 Scikit，这只需要几行代码：

```
from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier()
model.fit(x_train, y_train)
```

为了在 Scikit 中创建机器学习模型，我们需要对封装了所选学习算法的一个类进行实例化。本例使用的类就是 `KNeighborsClassifier`。然后，在模型上调用 `fit`，使模型拟合到数据，以完成对它的训练。由于只有 120 行训练数据，所以训练很快就能完成。

最后一步是使用从原始数据集中分割出来的 30 行测试数据来衡量模型的准确率。在 Scikit 中，这是通过调用模型的 `score` 方法完成的。

```
model.score(x_test, y_test)
```

在这个例子中，`score` 返回 0.966667，这意味着当用 `x_test` 中的特征进行预测，并将预测的标签与 `y_test` 中的实际标签进行比较时，模型有 97%的时间是正确的。

当然，我们之所以训练一个预测模型，唯一的目的就用它进行预测。在 Scikit 中，我们通过调用模型的 `predict` 方法来进行预测。以下语句预测一株鸢尾花的品种：萼片长 5.6 厘米，萼片宽 4.4 厘米，花瓣长 1.2 厘米，花瓣宽 0.4 厘米。结果 0 代表 Setosa 鸢尾花，1 代表 Versicolor 鸢尾花，2 代表 Virginica 鸢尾花。

```
model.predict([[5.6, 4.4, 1.2, 0.4]])
```

可以在一次 `predict` 方法调用中进行多个预测。这就是为什么传给它的是一个“列表的列

表”而不是单个列表的原因。该方法返回一个长度等于你传递的列表数的列表。由于只传递了一个列表来进行预测，所以返回的是只有一个值的列表。在本例中，预测出来的品种是0，这意味着模型预测萼片长度为5.6厘米，萼片宽度为4.4厘米，花瓣长度为1.2厘米，花瓣宽度为0.4厘米的鸢尾花很可能是Setosa鸢尾花。

如果在创建KNeighborsClassifier时不指定邻居数量，那么该值默认为5。可以像下面这样指定邻居的数量：

```
model = KNeighborsClassifier(n_neighbors=10)
```

试着用n_neighbors = 10对模型进行拟合（训练）和打分。该模型的得分是否相同？预测结果仍然是品种0吗？请自由尝试其他n_neighbors值，体验它们对结果的影响。

KNeighborsClassifier的内部工作原理

k近邻有时被称为一种懒惰（lazy）学习算法，因为大部分工作是在调用predict时完成的，而不是在调用fit时完成。事实上，从技术上讲，在训练时，除了生成一份供调用predict时使用的训练数据副本，它是不会做其他任何事情的。那么，在KNeighborsClassifier的fit方法中，实际会发生什么事情？

大多数情况下，fit会在内存中构建一个二叉树。这样一来，predict以后就不必对相邻样本进行蛮力查找，从而速度变得更快。如果KNeighborsClassifier判断二叉树没有帮助，在进行预测时就会采用蛮力法。这通常发生在训练数据很稀疏的时候——也就是说，大部分是0，只有零星几个非零值。

Scikit-Learn最棒的地方在于它是开源的。要知道更多关于某个特定的类或方法是如何工作的，可以直接去GitHub上找源代码。例如，在GitHub上，KNeighborsClassifier和RadiusNeighborsClassifier的源代码位于https://oreil.ly/mC4Rt。

这里采用的过程——加载数据、分割数据、创建分类器（classifier）或回归器（regressor）、调用fit将其和训练数据拟合、调用score用测试数据评估模型的准确率、最后调用predict进行预测——是你将在Scikit中反复使用的过程。在现实世界中，数据在用于训练和测试之前，经常需要先要走一遍清洗（cleaning）程序。例如，可能需要删除有缺失值的行，或者对数据进行删减以消除多余行。以后会看到很多这样的例子，但在目前这个例子中，所有数据都是完整的，而且开箱即用，结构良好，因此不需要进一步准备。

1.4 小结

机器学习为工程师和软件开发人员提供了一种解决问题的替代方法。机器学习不是使用传统的计算机算法将输入转化为输出，而是依靠学习算法从训练数据中建立数学模型。然后，它用这些模型将未来的输入转化为输出。

大多数机器学习模型都属于两类中的一类。其中，无监督学习模型被广泛用于通过强调相似性和差异来分析数据集。它们不需要带标签的数据。监督学习模型则从带标签的数据中学习，以便进行预测——例如，预测一笔信用卡交易是否合法。监督学习可以用来解决回归问题或分类问题。其中，回归模型预测数值结果，而分类模型预测所属类别。

k-means 聚类是一种流行的无监督学习算法，而 k-近邻是一种简单而有效的监督学习算法。许多（但不是全部）监督学习算法都可以用于回归*和*分类。例如，Scikit-Learn 的 `KNeighborsRegressor` 类将 k-近邻应用于回归问题，而 `KNeighborsClassifier` 类将同一算法应用于分类问题。

教育工作者经常使用 k-近邻向学生介绍监督学习，因为它很容易理解，而且在各种问题领域都表现得相当不错。在掌握了 k-近邻之后，为了进一步熟悉机器学习，下一步就是了解其他监督学习算法。这就是第 2 章的重点，它在回归建模的背景下介绍了几种流行的学习算法。

第 2 章 回归模型

第 1 章讲到，监督学习模型分为两类：回归模型和分类模型。另外还讲到，回归模型可以预测数值结果，例如房屋售价或网站访问量。回归建模是机器学习的一个重要方面，但有时却不被重视。零售商用它来预测需求（https://oreil.ly/pqs2a）。银行用它来筛选贷款申请，将信用评分、债务收入比（Debt-To-Income，DTI）和贷款价值比（Loan-To-Value）等变量考虑在内。保险公司则用它来设定保险费。但凡需要数字预测，回归模型就是合适的工具。

构建一个回归模型时，需要做出的第一个也是最重要的决定是使用什么学习算法。第 1 章介绍了一个简单的三分类模型，模型使用 k 近邻学习算法，根据萼片和花瓣的尺寸来识别鸢尾花的品种。k-近邻也可以用于回归，但它是在进行数值预测时可供选择的众多方法之一。其他学习算法往往能生成更准确的模型。

本章介绍了常见的回归算法（其中许多也可用于分类），并指导你构建一个回归模型，使用纽约市 Taxi and Limousine Commission（出租车和豪华轿车委员会）公布的数据来预测打车费用。另外，还描述了对回归模型的准确率进行评估的各种方法，并介绍了一种重要的测量准确率的技术，称为交叉验证。

2.1 线性回归

除了 K-近邻，线性回归可能是最简单的学习算法。它对相对线性的数据效果最好；也就是说，数据点大致沿着一条线落下。高中数学课讲过，二维平面上的直线方程是：

$y = mx + b$

其中，m 是直线的斜率，b 是直线与 y 轴相交的位置。图 1-11 的收入与工作年限数据集就很适合线性回归。图 2-1 展示了拟合了数据点的一条回归线。为了预测有 10 年经验的程序员的年薪，在直线上找到 $x = 10$ 的点即可。这条直线的方程是 $y = 3984x + 60040$。将 $x = 10$ 代入方程，即可预测年薪为 99 880 美元。

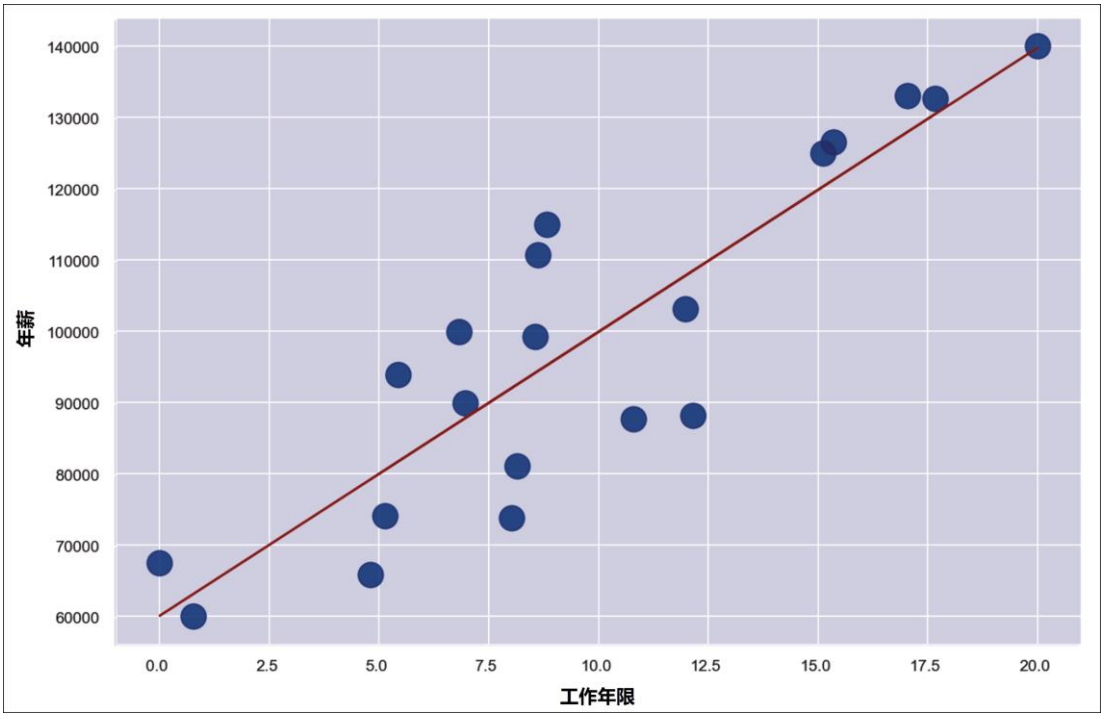


图 2-1 线性回归

训练线性回归模型的目标是最准确地预测 *m* 和 *b* 值。这一般是通过一个迭代过程来完成的，该过程从假定的 *m* 和 *b* 的值开始，反复进行，直到收敛到合适的值。

为了将一条直线拟合到一系列点，最常见的技术是普通最小二乘法（Ordinary Least Squares，OLS）回归（https://oreil.ly/n5fx9）。其工作原理是求每个点和回归线之间的 *y* 方向距离的平方，求它们的平方和，然后除以点的数量来计算均方误差（Mean Squared Error，MSE）。记住，对每个距离进行平方处理可以防止负的距离抵消正的距离。然后，它调整 *m* 和 *b* 以减少下一次的 MSE，并如此反复，直到 MSE 足够低。这里不打算讨论它具体如何确定在什么方向调整 *m* 和 *b*。这其实不难，但要涉及到一点微积分知识，特别是要使用 MSE 函数的偏导数（https://oreil.ly/dnOhg）来确定在下一次迭代中是增加还是减少 *m* 和 *b*。目前，你只需知道，只需十几次或更少的迭代，OLS 即可将一条直线拟合到一系列点。

Scikit-Learn 有许多类来帮助你构建线性回归模型，其中包括 `LinearRegression` 类（https://oreil.ly/e8r8p），它具现了 OLS。还有 `PolynomialFeatures` 类（https://oreil.ly/s3rJN），它将一条多项式曲线而不是一条直线拟合到训练数据。为了训练一个线性回归模型，代码可以简化成下面这个样子：

```
model = LinearRegression()
model.fit(x, y)
```

Scikit 还提供了其他线性回归类，例如 `Ridge`（https://oreil.ly/7I3ON）和 `Lasso`（https://oreil.ly/Fnj2v）。如果训练数据包含离群点（outliers），它们就非常有用了。第 1 章讲过，离群点是指与其他数据不一致的数据点。离群点会使模型产生偏差，或使其准确率下降。`Ridge` 和 `Lasso` 增加了正则化（regularization）（https://oreil.ly/x4Dt2）。它通过在训练期间调整系数，从而减轻离群点对结果的影响。处理离群点的另一种方法是完全删除它们，这是我们在本章末尾的打车费例子中要做的。

`Lasso` 回归还有一个额外的好处。如果训练数据存在多重共线性（multicollinearity）（https://oreil.ly/qNDA0），即两个或更多的输入变量线性相关，一个变量可以从另一个变量以合理的准确率进行预测，那么 `Lasso` 能有效地忽略多余的数据。

多重共线性的一个典型例子是，在数据集中，如果一列指定了房子中的房间数量，另一列指定了面积，那么就会出现多重共线性。更多的房间通常意味着更多的面积，因此这两个变量在某种程度上是相关的。

线性回归并不限于两个维度（x 和 y 值）。事实上，它适用于任何数量的维度。有一个自变量（x）的线性回归称为简单线性回归（simple linear regression），而有两个或更多自变量——例如，x_1、x_2、x_3 等——的线性回归则称为多元线性回归（multiple linear regression）。如果一个数据集是二维的，绘制数据以确定其形状非常简单。三维数据的绘制也不难。但是，绘制四维或五维数据集就颇有挑战性了。而具有数百或数千维的数据集根本不可能可视化。

那么，如何确定一个高维数据集是否适合线性回归？一个办法是使用主成分分析（Principal Component Analysis，PCA）（https://oreil.ly/jzc37）和 t 分布随机近邻嵌入（t-distributed Stochastic Neighbor Embedding，t-SNE）（https://oreil.ly/6mbe6）等技术，将 n 个维度减少至 2 或 3 个，这样就可以绘制它们。这些技术将在第 6 章介绍。这两种技术都能减少数据集的维度，同时不会造成大量的信息损失。以PCA为例，在保留原始数据集90%的信息的同时，将维数减少 90%的情况并不罕见。听起来有点儿像魔术，但实则不然。这完全是数学的功劳！

如果维数相对较少，可视化高维数据集的一个更简单的技术是配对图（pair plots，也称为散点图矩阵）（https://oreil.ly/kHiBd），它在传统的二维图中绘制成对的维度。在图 2-2 的配对图中，我们描绘了第 1 章介绍的鸢尾花数据集（https://oreil.ly/TjQb3）的萼片长度（sepal length）与花瓣长度（petal length）、萼片宽度（sepal width）与花瓣宽度（petal width）以及其他参数对。

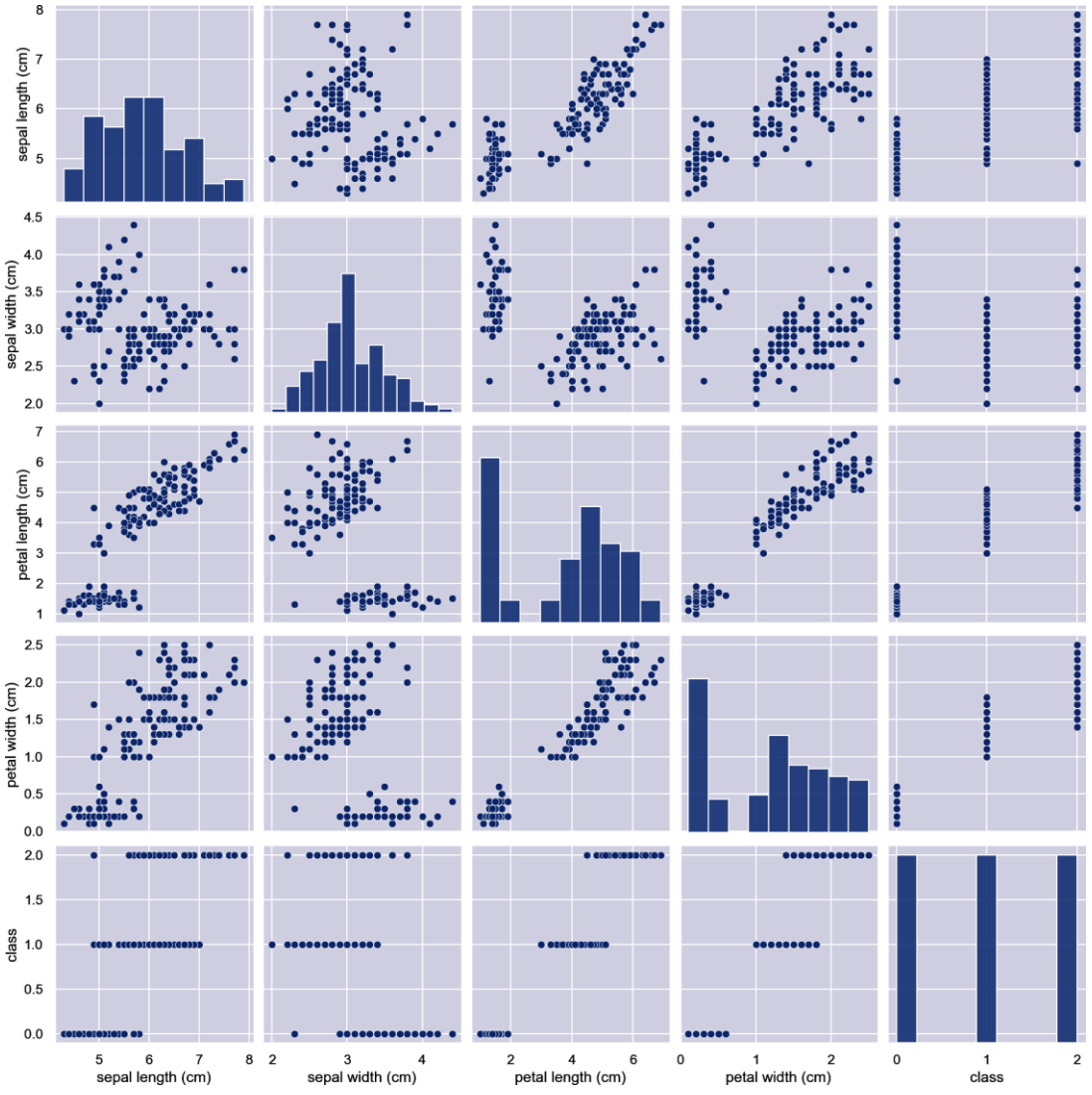


图 2-2 揭示变量对之间关系的配对图

使用 Seaborn 的 `pairplot` 函数（https://oreil.ly/2T9OS），可以很容易地创建配对图。图 2-2 的图用一行代码即可生成：

```
sns.pairplot(df)
```

配对图不仅有助于可视化数据集中的关系，而且在这个例子中，右下角的直方图也显示出数据集是平衡的，即所有三个类别的样本数量相等。出于第 3 章会讲到的原因，我们总是喜欢用平衡的数据集来训练分类模型。

线性回归是一种*参数化*学习算法，这意味着它的目的是检查数据集，并为方程式中的参数

（例如，*m* 和 *b*）找到最佳值。相反，k-近邻是一种非参数化的学习算法，因为它不将数据和一个方程式拟合。为什么一个学习算法是参数化还是非参数化很重要？这是因为用于训练参数化模型的数据集往往需要归一化（normalized）。用最简单的话来说，对数据进行归一化意味着确保所有列中的所有值都有一致的范围。第 5 章会详细介绍归一化，但现在要意识到，用未归一化的数据训练参数化的模型——例如，数据集在一列中包含 0~1 的值，而在另一列中包含 0~1000000 的值——会使这些模型的准确率降低，或使它们完全无法收敛于一个解。支持向量机（SVM）和神经网络尤其会如此，但其他参数化模型同样会如此。即便是 k-近邻模型，也是在归一化的数据下效果最好——因为虽然学习算法不是参数化的，但它在内部使用了基于距离的计算。

2.2 决策树

即使你没有上过计算机科学课程，也可能也知道二叉树是什么（https://oreil.ly/pE6EI）。在机器学习中，决策树（decision tree）是一种通过回答一系列问题来预测结果的树状结构。大多数决策树都是二叉树，在这种情况下，所有问题只需简单地回答是或否。

图 2-3 展示了 Scikit 从第一章介绍的收入与工作年限数据集建立的决策树。这个树很简单，因为数据集只包含一个特征列（工作年限），而且我把树的深度限制为 3。但是，这个技术可以扩展到无限大和无限复杂的树。在本例中，预测一个有 10 年经验的程序员的年薪只需要三个是/否的决定，如红色箭头所示。答案是 10 万美元，这和应用于同一数据集的 K-近邻和线性回归的预测结果相当接近。

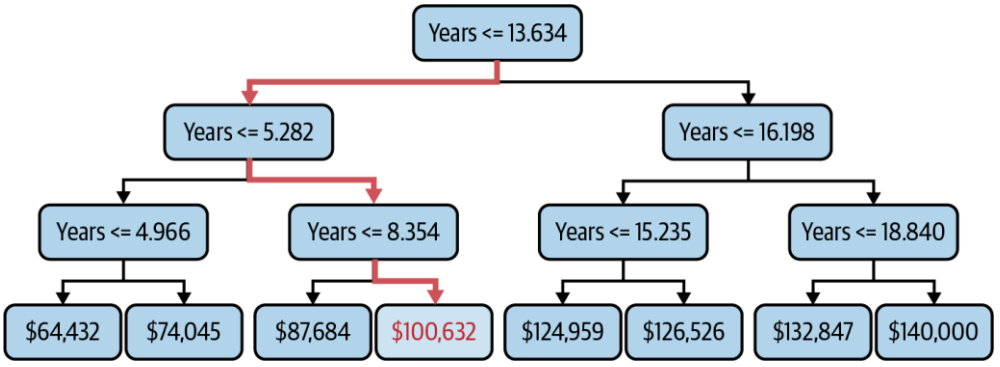


图 2-3 决策树

决策树可用于回归和分类。对于回归器来说，叶子节点（缺少子节点的节点）代表回归值。对于分类器来说，它们则代表分类（类别）。决策树回归器的输出是不连续的。输出始终是分配给叶子节点的值之一，而且叶子节点的数量有限。相反，线性回归模型的输出是连续的。它可以沿着训练数据的拟合线承担任何数值。在上例中，如果要求树预测一个有 10 年经验的人和一个有 13 年经验的人的年薪，那么将得到同样的答案。然而，如果将工作年限提高到 14 年，预测的年薪就会跳到 12.5 万美元（图 2-4）。如果让这棵树长得更深，

那么答案会变得更精确。但如果允许它生长得太深，就会导致大问题，原因我们马上就会讲到。

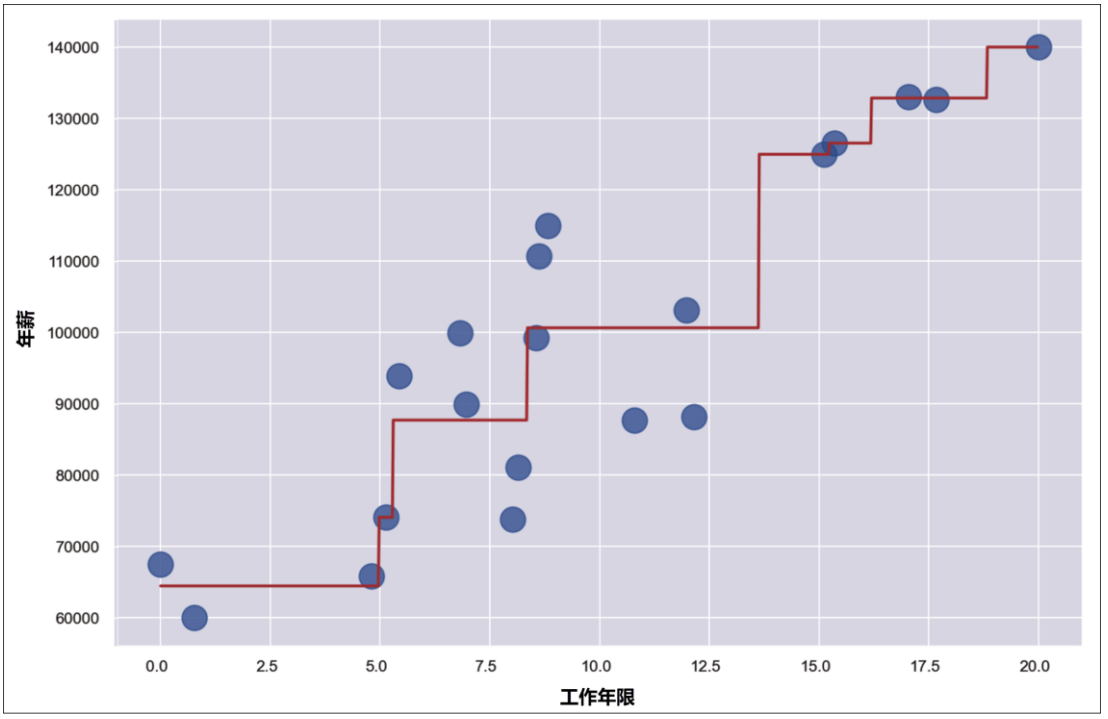


图 2-4 由决策树创建的数学模型

一旦决策树模型被训练出来——也就是说，一旦构建好这棵树——就可以迅速做出预测。但是，在每个节点上应该做什么决定？例如，为什么图 2-3 根节点所代表的年限等于 13.634？为什么不能是 10.000、8.742 或者其他一些数字？从这一点出发，如果数据集有多个特征列，那么应该在哪一个决策节点上拆分哪一列？

为了构建决策树，需要以递归方式对训练数据进行拆分（splitting）。拆分算法在向树添加一个节点时，所做的基本决策是：1）这个节点拆分的是哪一列；2）拆分所依据的值是什么。每次进行迭代时，目标是选择一个列和拆分值——对于分类问题来说，它最能减少剩余数据的“杂质”（impurity）；对于回归问题来说，它最能减少剩余数据的“方差”（variance）。分类器的一种常见的杂质度量是 Gini（基尼杂质）（https://oreil.ly/bCuJx），它大致量化了一个拆分值被错误分类（misclassify）的样本百分比。对于回归器，则通常改为使用平方误差和（sum of the squared error）或绝对误差和（sum of absolute error）。其中，“误差”（error）是指拆分值与拆分线两侧的值之间的差异。树的构建过程从根节点开始，并递归地向下构建，直到只剩下叶子节点，或者外部约束（例如对最大深度的限制）阻止树进一步生长。

利用 Scikit 的 `DecisionTreeRegressor` 类（https://oreil.ly/jLyVQ）和 `DecisionTreeClassifier` 类（https://oreil.ly/Ylrq4），我们可以非常方便地构建决策树。每个类都实现了著名的 CART（Classification and Regression Trees，分类与回归树）二叉树构建算法（https://oreil.ly/wQGjx），而且每个类都允许从几个杂质或方差度量标准中选择一个。每个类都支持像 `max_depth`、`min_samples_split` 和 `min_samples_leaf` 这样的参数，以便限制决策树的生长。如果全部接受默认值，构建一棵决策树可以像下面这样简单：

```
model = DecisionTreeRegressor()
model.fit(x, y)
```

决策树是非参数化的。训练决策树模型涉及的是一棵二叉树的构建，而不涉及将一个方程式拟合到数据集。这意味着用于构建决策树的数据不需要归一化（normalized）。

决策树有一个很大的优点：它们对非线性数据的处理和对线性数据的处理一样好。事实上，它们在很大程度上并不关心数据的形式。但是，它也有一个缺点。而且这是一个很大的问题，也是很少将单独的决策树用于机器学习的原因之一。这个问题就是过拟合（overfitting）（https://oreil.ly/ycffk）。

决策树很容易过拟合。如果让决策树成长得足够大，它基本上可以记住训练数据。可能看起来很准确，但如果和训练数据拟合得太紧，它的泛化或者说归纳（generalize）能力就会变得很差。换言之，在对之前没有见过的数据进行预测时，就不会那么准确。图 2-5 的决策树在不限制深度的前提下对收入和工作年限数据集进行拟合。注意红线穿过了所有点，形成了一个锯齿状的路径，这就是过拟合的一个明显迹象。过拟合是数据科学家的噩梦。一个不准确的模型就已经很糟糕了，但还有更糟糕的，即一个看起来准确，但实则不然的模型。

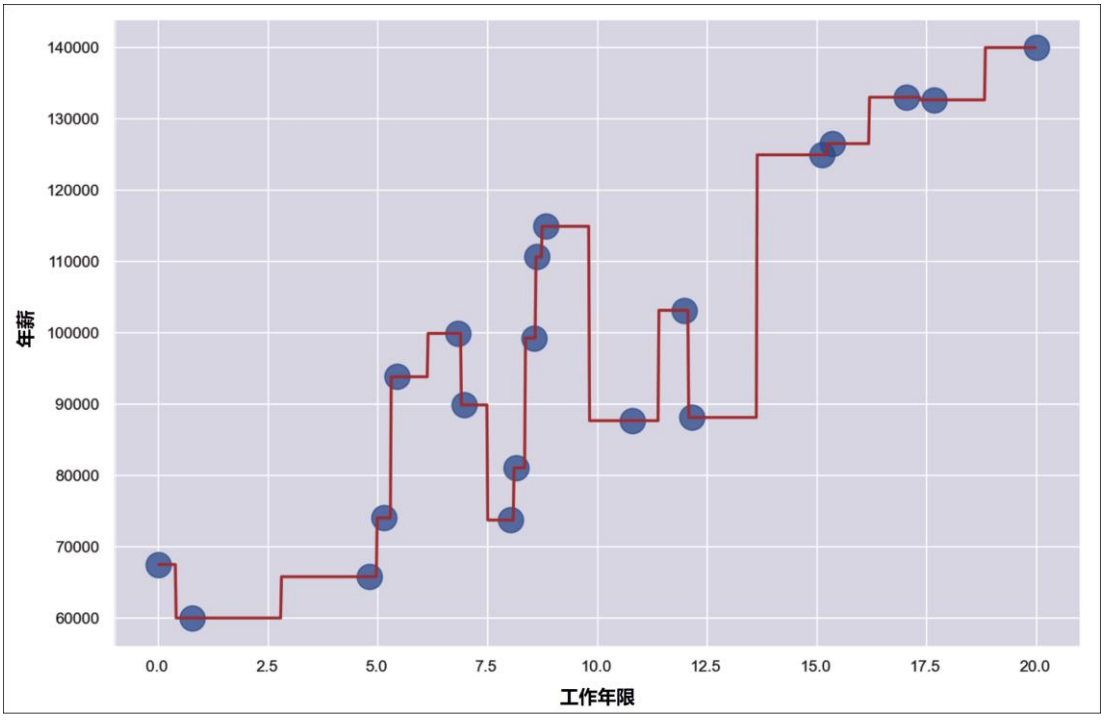


图 2-5 决策树对训练数据过拟合

使用决策树时，防止过拟合的一个方法是限制其增长，使其*不能*记住训练数据。另一个方法是使用称为随机森林的决策树分组。

2.3 随机森林

随机森林（random forest）是决策树的一个集合（一个集合通常有几百棵），每棵树在相同的数据上以不同的方式训练，如图 2-6 所示。通常，每棵树都在数据集中随机选择的行上训练，分支则基于每次拆分时随机选择的列。由于每棵树都在不同的数据子集上进行训练，所以模型不会与训练数据过于紧密地拟合。这些树是独立构建的，当模型进行预测时，它为所有决策树都运行输入，并对结果进行平均。由于树是独立构建的，所以训练过程可在支持它的硬件上并行化。

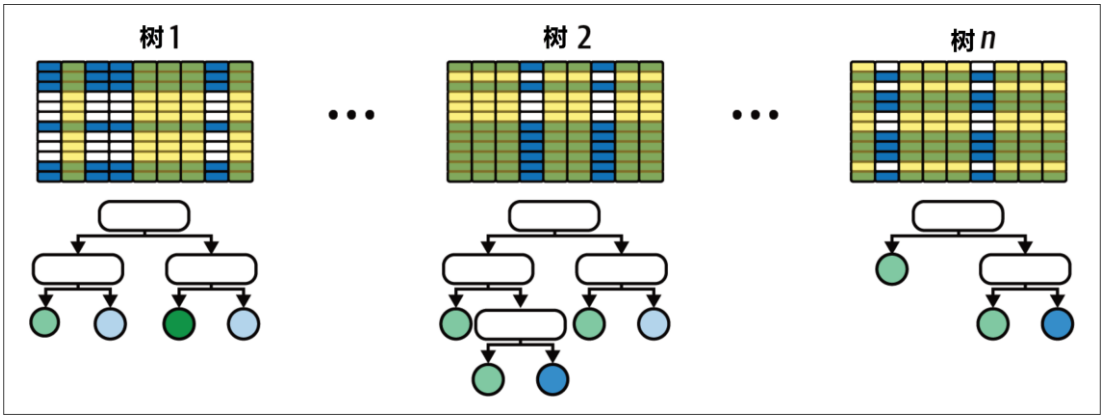


图 2-6 随机森林

这是一个简单的概念，而且在实践中运作良好。随机森林可以用于回归和分类，Scikit 提供了 `RandomForestRegressor`（https://oreil.ly/QwaN1）和 `RandomForestClassifier`（https://oreil.ly/xIXEp）等类来提供帮助。它们支持许多可调整的参数，其中包括 `n_estimators`，它指定了随机森林中树的数量（默认 100）；`max_depth`，它限制了每棵树的深度；以及 `max_samples`，它指定了训练数据中用于建立单棵树的行的比例。图 2-7 展示了在 `max_depth` = 3 和 `max_samples = 0.5`（没有一棵树能看到数据集中 50%以上的行）的情况下，`RandomForestRegressor` 对收入和工作年限数据集的拟合情况。

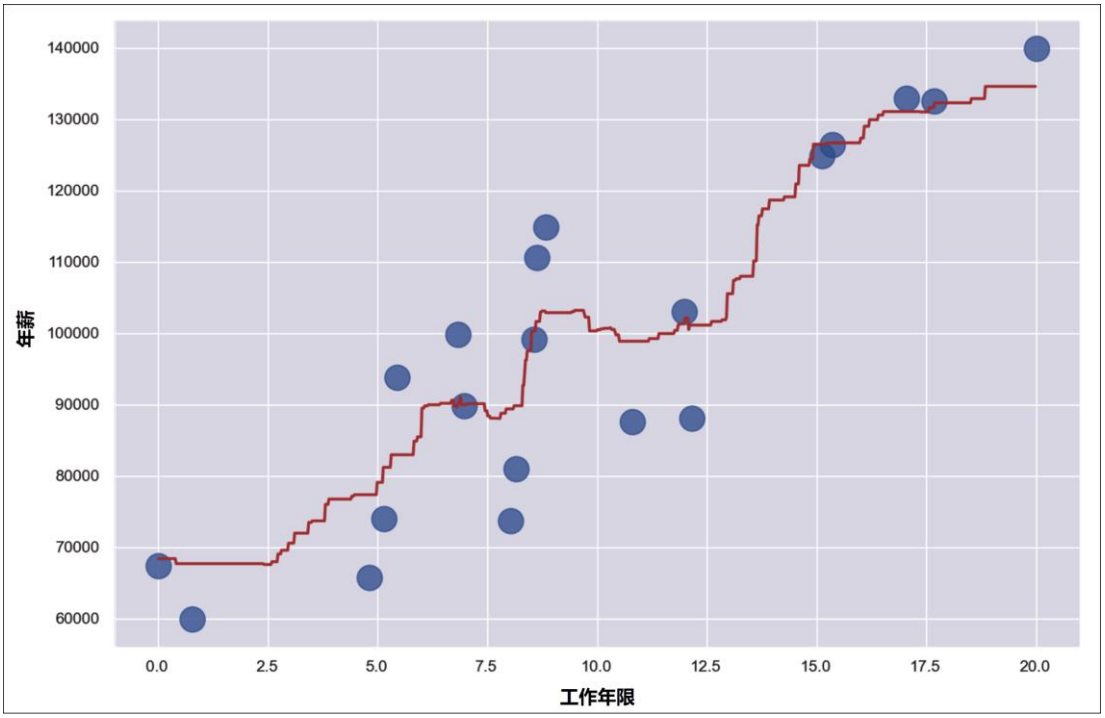


图 2-7 由随机森林创建的数学模型

由于决策树是非参数化的，所以随机森林也是非参数化的。尽管图 2-7 显示了随机森林如何拟合一个线性数据集，但随机森林也完全能对非线性数据集进行建模。

2.4 梯度提升机

随机森林证明了这样一个假设：可以把许多弱学习器（weak learners）——那些本身就不是强预测器的模型——结合起来形成准确的模型。随机森林中没有任何一棵树可以非常准确地预测一个结果。但将所有树放到一起，并对结果进行平均，它们的表现往往会超过其他模型。数据科学家将此称为集合建模（ensemble modeling）或集合学习（ensemble learning），详情请参见 https://oreil.ly/V9fqa。

另一种利用集合建模的方法是梯度提升（gradient boosting）（https://oreil.ly/dzOiH）。使用它的模型称为梯度提升机（Gradient-Boosting Machine，GBM）。大多数 GBM 都使用决策树，我们有时将这些 GBM 称为梯度提升决策树（Gradient-Boosted Decision Tree，GBDT）。和随机森林一样，GBDT 由决策树的集合构成。但是，GBDT 不是从数据的随机子集构建独立的决策树，而是一个接一个地构建依赖（dependent）决策树，也就是每棵树都用上一棵树的输出来训练。第一棵决策树对数据集进行建模。第二棵决策树对第一棵决策树的输出中的误差进行建模，第三棵对第二棵的输出中的误差进行建模，以此类推。为了进行预

测，GBDT 通过每棵决策树来运行输入，并将所有输出相加以得出结果。每加一次，结果就会变得稍微准确一些，这正是加法建模（additive modeling）（https://oreil.ly/vyJyq）这一术语的来历。这就像把高尔夫球打到球道上，再连续打出更短的球，直至最终入洞。

GBDT 模型中的每棵决策树都是一个弱学习器。事实上，GBDT 通常使用决策树桩（decision tree stumps），即深度为 1 的决策树（一个根节点和两个子节点），如图 2-8 所示。在训练过程中，首先要取训练数据中所有目标值的平均值，从而为预测创建一个基线。然后，从目标值中减去平均值，从而生成一组新的目标值或残差（residuals），供第一棵树预测。训练完第一棵树后，通过它来运行输入，从而生成一组预测值。然后，将这些预测值加到前一组预测值上，通过从原始（实际）目标值中减去总和来生成一组新的残差，并训练第二棵树来预测这些残差。对 *n* 棵树重复这个过程（*n* 通常为 100 或更多），最终生成一个集合模型（ensemble model）。为了确保每棵决策树都是一个弱学习器，GBDT 模型将每棵决策树的输出乘以一个学习率（learning rate），以减轻它们对结果的影响。学习率通常是一个很小的数字，例如 0.1，可以在使用实现了 GBM 的类时作为参数指定。

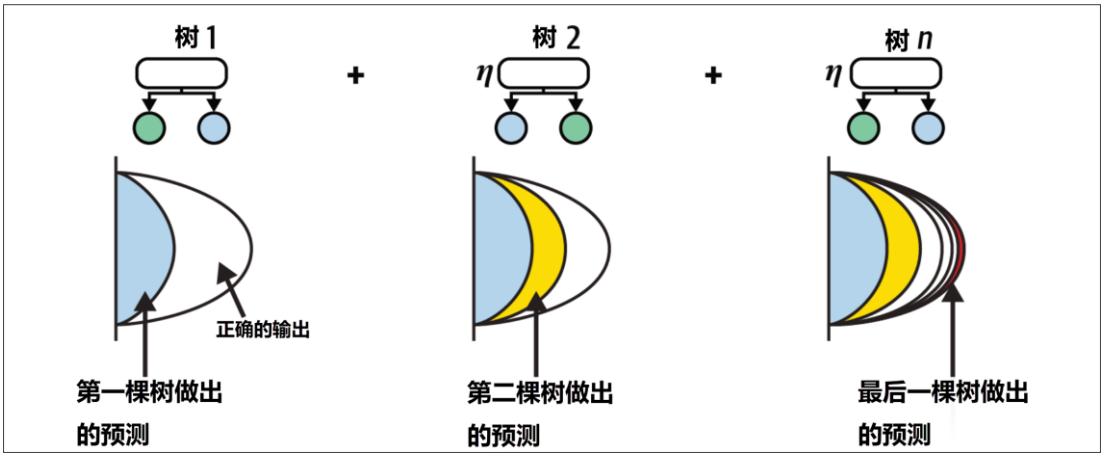


图 2-8 梯度提升机

Scikit 包括名为 `GradientBoostingRegressor`（https://oreil.ly/ZhSop）和 `GradientBoostingClassifier`（https://oreil.ly/9RlW5）的类来帮助你构建 GBDT。但是，如果真的想了解 GBDT 的工作原理，可以用 Scikit 的 `DecisionTreeRegressor` 类自己构建一个。例 2-1 的代码实现了一个有 100 个决策树桩的 GBDT，并预测了一个有 10 年经验的程序员的年薪。

例 2-1 梯度提升决策树的实现

```
learning_rate = 0.1 # 学习率
n_trees = 100 # 决策树的数量
trees = [] # 构成模型的树

# 计算所有目标值的均值
y_pred = np.array([y.mean()] * len(y))
baseline = y_pred

# 创建n_trees棵树，用来自上一棵树的输出
# 中的误差来训练每一棵树
for i in range(n_trees):
    error = y - y_pred
    tree = DecisionTreeRegressor(max_depth=1, random_state=0)
    tree.fit(x, error)
    predictions = tree.predict(x)
    y_pred = y_pred + (learning_rate * predictions)
    trees.append(tree)

# 为x=10预测一个y
y_pred = np.array([baseline[0]] * len(x))

for tree in trees:
    y_pred = y_pred + (learning_rate * tree.predict([[10.0]]))

y_pred[0]
```

在图 2-9 左侧，是将单一决策树树桩应用于收入和工作年限数据集的输出。该模型是一个弱学习器，只能预测两种不同的年薪水平。右侧是例 2-1 的模型的输出。弱学习器的加法效应产生了一个强学习器，预测有 10 年经验的程序员每年应该有 99 082 美元的收入，这和其他模型的预测结果一致。

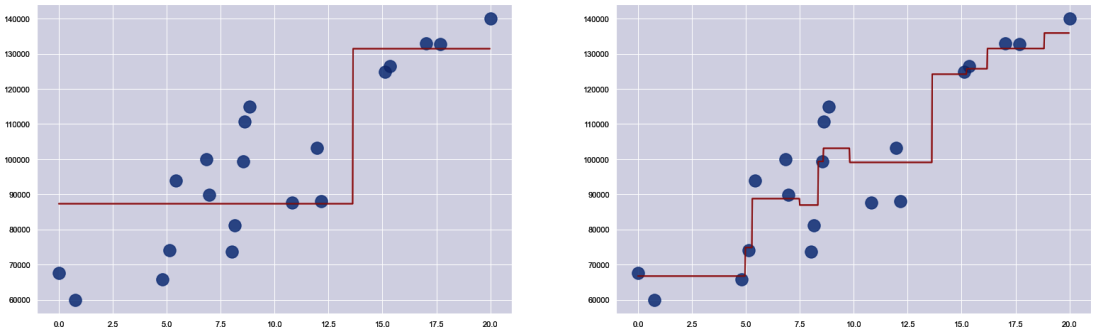


图 2-9 对比单一决策树和梯度提升决策树

GBDT 可用于回归和分类，而且是非参数化的。除了神经网络和支持向量机，在对复杂数据集进行建模时，数据科学家发现 GBDT 往往也最强大的。

有别于线性回归模型和随机森林，GBDT 很容易发生过拟合。在使用 `GradientBoostingRegressor` 和 `GradientBoostingClassifier` 时，缓解过拟合的一个方法是使用 `subsample` 参数来防止单个树看到整个数据集，这类似于 `max_samples` 之于随机森林的作用。另一个方法是使用 `learning_rate` 参数来降低学习率，其默认值为 0.1。

2.5 支持向量机

对支持向量机（Support Vector Machine，SVM）的全面讨论将在第 5 章进行，但和 GBM 一起，它们代表了统计机器学习的最前沿。它们通常可以对高度非线性的数据集进行模型拟合，其他学习算法则不能。它们是如此重要，以至于值得和其他所有算法分开讨论。它们采用一种称为核技巧（kernel tricks）的数学手段来模拟为数据增加维度的效果。其背后的思路在于，在 *m* 个维度中不可拆分的数据在 *n* 个维度中或许就能拆分。下面是一个简单的例子。

图 2-10 左侧的二维数据集中的类别不能用一条线拆分。但是，如果增加一个第三维，使靠近中心的点有较高的 *z* 值，而远离中心的点有较低的 *z* 值，如右图所示，那么可以在红点和紫点之间滑动一个平面，从而实现 100%的类别拆分。这就是 SVM 的工作原理。如果把它推广到任意的数据集上，虽然在数学上很复杂，但确实是一种极其强大的技术，而且已由 Scikit 进行了大幅简化。

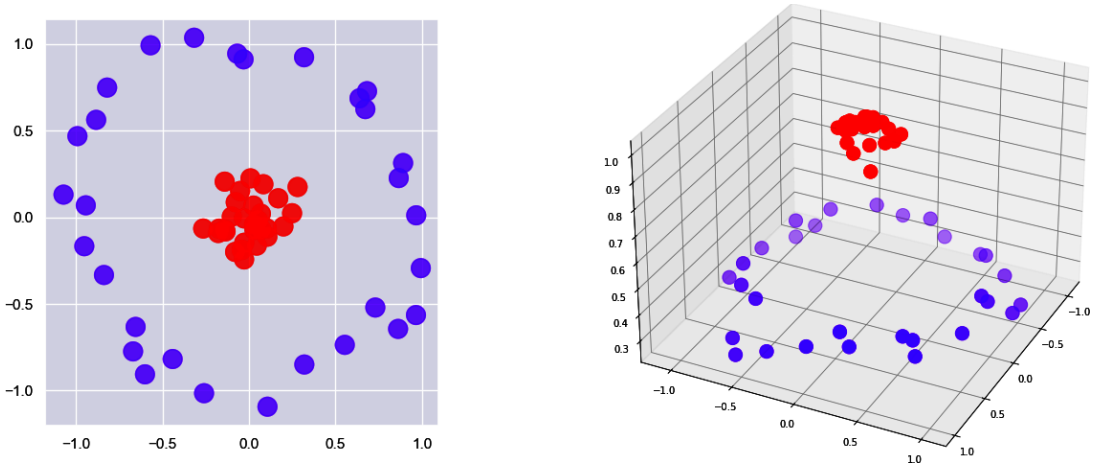


图 2-10 支持向量机

支持向量机主要用于分类，但它们也可以用于回归。Scikit 提供了针对这两种情况的类，其中包括用于分类问题的 `SVC` 类（https://oreil.ly/MWsSX）和用于回归问题的 `SVR` 类（https://oreil.ly/YPvZa）。我们将在第 5 章全面讨论这些类。不过，现在再去参加机器学习的讨论，只需抛出“支持向量机”这个词，你立刻就会成为聚会的主角。

2.6 回归模型的精度测量

第 1 章讲过，需要用一组数据来训练一个模型，用另一组数据来测试它。另外，可以将测试数据传递给模型的 `score` 方法，从而对模型的准确率进行评分。通过测试，可以量化模

型在预测方面的准确率。用和训练时不同的数据集对模型进行测试非常重要，因为否则的话，它或许能合理地学习训练数据，但这并不意味着它能很好地泛化（归纳）——即做出准确的*预测*。不测试一个模型，就不知道它到底有多准确。

工程师经常使用 Scikit 的 `train_test_split` 函数（https://oreil.ly/AkvO3）将一个数据集分割成一个训练数据集和一个测试数据集。但是，以这种方式分割一个小数据集时，不一定能信任模型的 `score` 方法所返回的分数。那么，这个分数到底意味着什么？对于回归模型和分类模型来说，这个问题的答案是不同的。另外，回归模型只有少数几种评分方法，而分类模型的评分方法有好多种。现在，让我们花点时间来了解 `score` 为回归模型返回的数字，以及可以做什么来对这个数字更有信心。

为了证明为什么在处理小数据集时，需要对 `score` 返回的数值持某种程度的怀疑，请做一个简单的实验。使用以下代码加载 Scikit 的加州房价数据集（https://oreil.ly/mt7ch），打乱（shuffle）所有行，并提取前 1000 行。

```
from sklearn.utils import shuffle
from sklearn.datasets import fetch_california_housing

df = fetch_california_housing(as_frame=True).frame
df = shuffle(df, random_state=0)
df = df.head(1000)
df.head()
```

该数据集包含的列名有 `MedInc`（收入中位数）和 `MedHouseVal`（房屋价值中位数）等，如下图所示。这些细节现在并不重要。重要的是，需要构建一个模型，使用所有其他列的值来预测 `MedHouseVal` 值。

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	MedHouseVal
14740	4.1518	22.0	5.663073	1.075472	1551.0	4.180593	32.58	-117.05	1.369
10101	5.7796	32.0	6.107226	0.927739	1296.0	3.020979	33.92	-117.97	2.413
20566	4.3487	29.0	5.930712	1.026217	1554.0	2.910112	38.65	-121.84	2.007
2670	2.4511	37.0	4.992958	1.316901	390.0	2.746479	33.20	-115.60	0.725
15709	5.0049	25.0	4.319261	1.039578	649.0	1.712401	37.79	-122.43	4.600

使用以下代码对数据进行 80/20 拆分，即用 80%的数据训练一个线性回归模型来预测房屋价格，然后用剩余 20%数据对模型进行测试，对其准确率进行评分。

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

x = df.drop(['MedHouseVal'], axis=1)
y = df['MedHouseVal']
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,
                                                    random_state=0)

model = LinearRegression()
model.fit(x_train, y_train)
model.score(x_test, y_test)
```

在本例中，score 返回 0.5863，这表面上是说该模型使用测试数据中的特征进行预测的准确率约为 59%。好吧，目前情况还不错。

现在将传递给 train_test_split 的 random_state 值从 0 改为 1，再次运行代码。

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,
                                                    random_state=1)

model = LinearRegression()
model.fit(x_train, y_train)
model.score(x_test, y_test)
```

这一次，score 返回 0.6255。那么，到底该信任谁呢？该模型的预测准确率是 59%还是 63%？为什么 score 会返回两个不同的值？

当 train_test_split 分割数据集的时候，它为训练数据集和测试数据集随机选择行。random_state 参数为做选择的随机数生成器提供种子值。通过指定两个不同的种子值，我们用两个不同的数据集训练模型，也用两个不同的数据集测试。当然，两个数据集之间肯定有一定的重叠。但事实是，为 train_test_split 的随机数生成器提供的种子值会影响结果。数据集越小，这种影响可能越大。

解决方案是交叉验证（cross-validation）。为了交叉验证一个模型，需要将数据集拆分为若干个折（folds），如图 2-11 所示。5 折很常见，但完全可以使用自己喜欢的任意数量的折。然后，对模型进行 5 次训练（每个折一次），每次都用不同的 80%数据集进行训练，用不同的 20%数据集进行测试。然后，对分数进行平均，生成一个交叉验证分数。这个分数比 score 返回的分数更可靠，因为它对数据的拆分方式不太敏感。这个过程称为 k 折交叉验证（k-fold cross-validation）。

图 2-11 5 折交叉验证

交叉验证的缺点是耗时较长。用 5 折交叉验证，需要对模型训练 5 次。好消息是，交叉验证一般只适用于小数据集。而如果数据集很小，模型可能很快就能训练出来。

可以自己写代码来做交叉验证，但没必这样做，因为 Scikit 会帮你做。对模型进行交叉验证模型只需一行代码：

```
cross_val_score(model, x, y, cv=5).mean()
```

在本例中，交叉验证的分数应该在 0.61 左右，介于由模型的 score 方法所生成的两个值之间。和其他两个分数相比，它是对模型准确率的一个更精确的度量。

模型在交叉验证之前不需要训练，因为 cross_val_score 为自动为你训练。然而，cross_val_score 训练的是模型的一个*副本*，而不是模型本身。所以，一旦用交叉验证来度量了模型的准确率后，仍然需要在预测前调用一次 fit。

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

注意，这里是将整个数据集传递给 fit，而不是传递拆分出一个子集供训练。这是交叉验证的另一个优势：可以用*所有*数据来训练模型。如果数据集一开始就很小的话，这个好处就显得非常重要。不再有硬性的要求留出其中一部分供测试。然而，即便使用了交叉验证，如果有专门用于测试的数据，用专门保留的数据对模型进行评分仍然是有用的。记住，除非知道了一个模型对未训练过的数据的反应，否则不会真正知道它有多准确。

作为一条经验法则，我们只对小数据集进行交叉验证。对于大数据集，则进行训练/测试拆分。数据集越大，它对数据是如何拆分的就越不敏感。

这就引出了一个问题：对一个模型进行评分或交叉验证时，返回的值到底代表什么？对于回归模型来说，它是决定系数（coefficient of determination）（https://oreil.ly/W1UHd），也称为 R 方分数（R-squared score），或简称 R^2。决定系数通常是一个 0~1 的值（之所以说“通常”，是因为在某些情况下，它可能变成负值），它量化了在输出中可由输入中的各种变量解释的方差。可以简单地这样理解，如果 R^2 分数为 0.8，那么意味着模型在预测时平均应该有 80%的准确率。R2 分数越高，模型就越准确。还有其他方法来度量回归模型的准确率，其中包括均方误差（Mean Squared Error，MSE）和平均绝对误差（Mean Absolute Error，MAE）。这些数字只有在输出值的范围内才有意义，而 R^2 给出的是一个简单的、与范围无关的数字。可以在 Scikit 文档（https://oreil.ly/G3LUI）中阅读更多关于回归指标和获取它们的方法。

对于分类模型来说，模型的 score 方法所返回的值是完全不同的。我们将在第 3 章讨论量化分类模型准确率的各种方法。

2.7 使用回归来预测车费

想象一下，你在一家出租车公司工作，客户们最大的抱怨之一是，他们在下车前不知道最后要付多少钱。这是因为距离并不是决定车费的唯一变量。所以，你决定开发一个移动应用来解决这个问题，客户可以在登上出租车时使用该应用来估算车费。为了在这个应用中引入 AI，你打算利用公司多年来收集的大量车费数据来构建一个机器学习模型。

让我们训练一个回归模型，根据当前是一天中的什么时间、是一周中的哪一天以及上下车地点来预测车费。首先，下载包含数据集的 CSV 文件（https://oreil.ly/qgx9X），把它复制到用于存放 Jupyter 笔记本的目录下的 Data 子目录中。然后，使用以下代码将数据集加载到笔记本中。该数据集包含大约 55000 行，是 Kaggle 举办的“New York City Taxi Fare Prediction”（纽约出租车费预测）比赛（https://oreil.ly/Dy9Ye）所用的一个更大的数据集的子集。这些数据在使用前需要进行相当多的准备工作——这在机器学习中并不罕见。对于数据科学家来说，收集和准备数据经常会占据他们 90%或者更多的时间。

```
import pandas as pd

df = pd.read_csv('Data/taxi-fares.csv', parse_dates=['pickup_datetime'])
df.head()
```

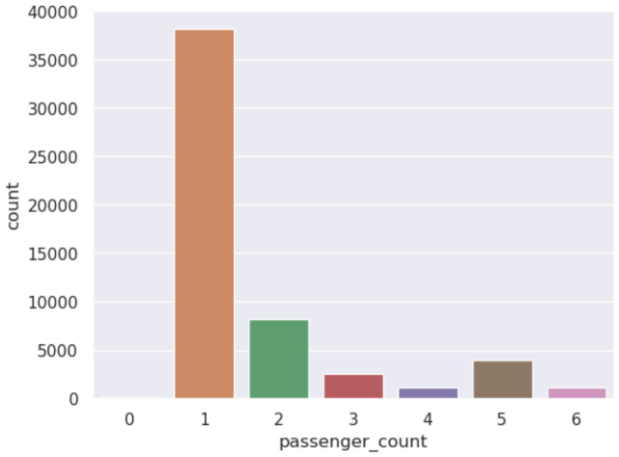
注意，这里使用 `read_csv` 函数的 `parse_dates` 参数将 `pickup_datetime` 列中的字符串解析为 Python `datetime` 对象 (https://oreil.ly/msqMQ)。代码的输出结果如下所示：

	key	fare_amount	pickup_datetime	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
0	2014-06-15 17:11:00.000000107	7.0	2014-06-15 17:11:00+00:00	-73.995420	40.759662	-73.987607	40.751247	1
1	2011-03-14 22:43:00.00000095	4.9	2011-03-14 22:43:00+00:00	-73.993552	40.731110	-73.998497	40.737200	5
2	2011-02-14 15:14:00.00000067	6.1	2011-02-14 15:14:00+00:00	-73.972380	40.749527	-73.990638	40.745328	1
3	2009-10-29 11:29:00.00000040	6.9	2009-10-29 11:29:00+00:00	-73.973703	40.763542	-73.984253	40.758603	5
4	2011-07-02 10:38:00.00000028	10.5	2011-07-02 10:38:00+00:00	-73.921262	40.743615	-73.967383	40.765162	1

每一行都代表一次打车，包含的信息有：车费、上车和下车地点（用经纬度表示）以及乘客人数。我们想预测的是车费。使用以下代码画一个直方图，显示有多少行的乘客人数为 1，有多少行的乘客人数为 2，以此类推。

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

sns.countplot(x=df['passenger_count'])
```

下面是输出：



由于只对预测单个乘客的车费感兴趣，所以使用以下代码删除所有含有多名乘客的行，并从数据集中删除 key 列，因为该列是不需要的——换言之，我们想要尝试的预测不基于该特征：

```
df = df[df['passenger_count'] == 1]
df = df.drop(['key', 'passenger_count'], axis=1)
df.head()
```

这样数据集就只剩下了 38 233 行，可用以下语句来验证：

```
df.shape
```

现在，使用 Pandas 的 corr 方法（https://oreil.ly/BNX2X）来找出输入变量（如经纬度）对 fare_amount（车费）列中的数值的影响程度。

```
corr_matrix = df.corr()
corr_matrix['fare_amount'].sort_values(ascending=False)
```

输出结果如下所示：

```
fare_amount          1.000000
dropoff_longitude    0.020438
pickup_longitude     0.015742
pickup_latitude     -0.015915
dropoff_latitude    -0.021711
Name: fare_amount, dtype: float64
```

这些数字看起来并不是很令人鼓舞。经纬度与车费关系不大，至少目前看起来是这样的。但是，凭借我们的直觉，它们应该和车费有很大关系，因为它们指定了起点和终点，而较长的车程自然会产生较高的车费。

现在，有意思的部分来了：创建对结果有更大影响的全新的数据列——其值是由其他列的值计算出来的。我们要增加一些列，指定打车的时候是一周中的哪一天（0=周一，1=周日，以此类推），乘客在几点钟上车（0~23），以及打车距离（直线距离，而非实际行驶距离；单位：英里）。为了计算距离，这段代码假设大多数打车时间都很短，因此可以安全地忽

略地球曲率。

```
from math import sqrt

for i, row in df.iterrows():
    dt = row['pickup_datetime']
    df.at[i, 'day_of_week'] = dt.weekday()
    df.at[i, 'pickup_time'] = dt.hour
    x = (row['dropoff_longitude'] - row['pickup_longitude']) * 54.6
    y = (row['dropoff_latitude'] - row['pickup_latitude']) * 69.0
    distance = sqrt(x**2 + y**2)
    df.at[i, 'distance'] = distance

df.head()
```

不是所有列都需要，所以用以下语句删除无关的列：

```
df.drop(columns=['pickup_datetime', 'pickup_longitude', 'pickup_latitude',
                 'dropoff_longitude', 'dropoff_latitude'], inplace=True)
df.head()
```

再用 corr 方法检查一下相关性：

```
corr_matrix = df.corr()
corr_matrix['fare_amount'].sort_values(ascending=False)
```

打车距离和车费之间仍然没有很强的相关性。以下语句或许能解释为什么：

```
df.describe()
```

输出结果如下所示：

	fare_amount	day_of_week	pickup_time	distance
count	38233.000000	38233.000000	38233.000000	38233.000000
mean	11.214115	2.951534	13.387989	12.018397
std	9.703149	1.932809	6.446519	217.357022
min	-22.100000	0.000000	0.000000	0.000000
25%	6.000000	1.000000	9.000000	0.762116
50%	8.500000	3.000000	14.000000	1.331326
75%	12.500000	5.000000	19.000000	2.402226
max	256.000000	6.000000	23.000000	4923.837280

该数据集包含离群值，而离群值经常使机器学习模型的结果出现偏差。通过消除负的车费，并对车费和距离进行合理限制来筛选数据集。然后，再次运行 corr。

```
df = df[(df['distance'] > 1.0) & (df['distance'] < 10.0)]
df = df[(df['fare_amount'] > 0.0) & (df['fare_amount'] < 50.0)]

corr_matrix = df.corr()
corr_matrix['fare_amount'].sort_values(ascending=False)
```

输出结果如下所示：

```
fare_amount    1.000000
distance       0.851913
day_of_week   -0.003570
pickup_time   -0.023085
Name: fare_amount, dtype: float64
```

这看起来好多了！大部分（85%）车费的变化是由打车距离解释的。一周中的哪一天和一天中的几点钟与车费之间的相关性仍然很弱。但是，这并不奇怪，因为打车距离是决定出租车打车费用的主要因素。让我们保留这些列，因为高峰时间从地点 A 打车到地点 B 可能需要更长的时间，或者周五下午 5 点的路况可能与周六下午 5 点的路况不一样，这些都是合理的。

现在，是时候训练一个回归模型了。让我们尝试三种不同的学习算法，确定哪一种产生最准确的拟合，并使用交叉验证来衡量准确率。先从线性回归模型开始：

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score

x = df.drop(['fare_amount'], axis=1)
y = df['fare_amount']
model = LinearRegression()
cross_val_score(model, x, y, cv=5).mean()
```

接着为同一个数据集尝试 `RandomForestRegressor`，看看它的准确率如何。之前说过，随机森林模型在数据上训练多个决策树，并对所有树的结果进行平均来做出预测。

```
from sklearn.ensemble import RandomForestRegressor

model = RandomForestRegressor(random_state=0)
cross_val_score(model, x, y, cv=5).mean()
```

最后尝试一下 `GradientBoostingRegressor`。梯度提升机使用多棵决策树，每棵树的训练都是为了补偿前一棵树输出中的误差。

```
from sklearn.ensemble import GradientBoostingRegressor

model = GradientBoostingRegressor(random_state=0)
cross_val_score(model, x, y, cv=5).mean()
```

由于 `GradientBoostingRegressor` 产生了最高的交叉验证决定系数，所以我们用整个数据集来训练它，如下所示：

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

最后，用训练好的模型做一对预测。首先，估计一下周五下午 5 点打车出行 2 英里需要多少车费。由于已经将包含列名的 `DataFrame` 传给了 `fit` 方法，所以如果为 `predict` 传递列表或 NumPy 数组，那么最新版本的 Scikit 会显示一个警告。因此，我们改为输入一个 `DataFrame`：

```
model.predict(pd.DataFrame({ 'day_of_week': [4], 'pickup_time': [17],
                             'distance': [2.0] }))
```

然后预测一天后（周六）下午 5 点 2 英里车程的费用：

```
model.predict(pd.DataFrame({ 'day_of_week': [5], 'pickup_time': [17],
                            'distance': [2.0] }))
```

该模型对星期六下午同一车程预测的费用是更高还是更低？鉴于数据来自纽约市的出租车，答案是否合理？正常情况下，周五下午的路况要比周六下午差一些。

2.8 小结

回归模型是有监督学习模型，用于预测数值结果，例如打车费用。用于回归的主要学习算法包括以下几种：

- 线性回归：将训练数据拟合为直线方程来建立模型。
- 决策树：使用二叉树，通过回答一系列是或否的问题来预测结果。
- 随机森林：使用多个独立的决策树对数据进行建模，对过拟合有一定的抗性。
- 梯度提升机：使用多棵*依赖*决策树，每棵树都对上一棵树输出中的误差进行建模。
- 支持向量机：采取完全不同的方法对数据进行建模，假定原始问题空间中不可线性分离的数据在高维空间中或许能线性分离。

Scikit 通过 `LinearRegression`、`RandomForestRegressor` 和 `GradientBoostingRegressor` 等类提供了这些以及其他学习算法的便捷实现。

量化回归模型准确率的一个常见指标是 R^2 分数，也称为决定系数。它通常是一个 0~1 的值。数字越大，准确率越高。从技术上讲，它度量的是在输出中可由输入中的值来解释的方差。对于小的数据集来说，相较于只将数据拆分一次（分别用于训练和测试），k 折交叉验证能使你对 R^2 分数更有信心，。k 折要对模型进行 k 次训练，每次都以不同的方式拆分数据集。

现实世界的数据集往往是混乱的，需要进一步准备才能用于机器学习。就像出租车费的例子展示的那样，训练数据中的离群值会影响模型的准确率，或者使模型根本不起作用。一个解决方案是在训练模型前识别离群值并将其移除。另一个办法是采用支持正则化的学习算法，如山脊回归或拉索回归。

回归模型在机器学习中很常见，但分类模型更常见。第 3 章将直面分类模型，介绍另一种领先的学习算法：逻辑回归，而且要以本章所学的内容为基础。

第 3 章 分类模型

上一章介绍的机器学习模型使用了各种形式的回归，根据出行距离、周几和几点钟来预测打车费用。回归模型预测数值结果，业界广泛用它来预测销售、价格、需求和其他驱动商业决策的数字。同等重要的是分类模型，它预测的是分类结果，例如信用卡交易是否是欺诈性的，或一个手写的字符代表哪个字母。

分类模型主要有两类：二分类模型（binary classification models），其中只有两种可能的结果；多分类模型（multiclass classification models），其中有两种以上可能的结果。在这两种情况下，模型都会为输入分配一个单一的类别，或者说类别标签（class label）。不太常见的是多标签分类模型（multilabel classification models），它允许将一个输入分为几类。例如，预测一个文档既是机器学习的论文，又是基因组学的论文。有的模型还能预测一个输入不属于任何一个可能的类别。

你所知道的关于回归模型的大部分内容也适用于分类模型。例如，许多支持回归模型的学习算法同样适用于分类模型。回归和分类之间的一个实质性区别是如何度量一个模型的准确率（accuracy）。对于分类模型来说，没有 R^2 分数这样的东西。取而代之的是大量的度量指标，例如精确率（precision）、召回率（recall）、特异度（specificity）、灵敏度（sensitivity）和 F1 得分等。为了熟练掌握分类模型，关键之一就是熟悉各种准确率指标，更重要的是，根据模型的预期应用，知道应该使用其中的哪一个（或哪几个）。

第 1 章的鸢（yuān）尾花教程已经展示了一个多分类的例子。现在是时候深入研究机器学习分类器了，首先从最久经考验的一个学习算法开始：逻辑回归（logistic regression）。该算法*只*适用于分类模型。

3.1 逻辑回归

有许多学习算法都能用于分类问题。第 2 章讲解了决策树、随机森林和梯度提升机（GBM）如何将回归模型与训练数据拟合。这些算法也可用于分类问题。Scikit 包含了 `DecisionTreeClassifier`（https://oreil.ly/8N6MG）、`RandomForestClassifier`（https://oreil.ly/3mNhv）和 `GradientBoostingClassifier`（https://oreil.ly/ilXW4）等类来为此提供帮助。第 1 章使用 Scikit 的 `KNeighborsClassifier` 类（https://oreil.ly/zdDCH）构建了一个三分类模型，并用 k-近邻作为学习算法。

这些都是重要的学习算法，它们在当代许多机器学习模型中都有使用。但是，最流行的分类算法之一是逻辑回归（https://oreil.ly/o5rBl），它分析数据的分布，并为其拟合一个方程，俗艳定义一个给定的样本属于两个可能的类别中的每一个的概率。例如，它可能判断一个样本中的值有 10%的机率对应于类别 0，90%的机率对应于类别 1。在这种情况下，逻辑回归会预测该样本对应于类别 1。虽然叫这个名字，但逻辑回归实际是一种分类算法，而

非回归算法。它的目的不是创建回归模型，而是对概率进行量化以便对输入样本进行分类。

以图 3-1 的数据点为例，它们属于两个类别：0（蓝色）和 1（红色）。我们假设 x 轴表示某人为一门考试而准备的小时数，而 y 轴表示考试通过（1）还是未通过（0）。蓝点落在 $x = 0$ 到 $x = 10$ 的范围内，红点则落在 $x = 5$ 到 $x = 15$ 的范围内。在这种情况下，无法选择一个 x 值直接将两个类别分开，因为两个类别都有在 $x = 5$ 到 $x = 10$ 之间的值。(试着画一条垂直线，一边只有红点，另一边只有蓝点，看看能不能做到。）但是，完全可以画一条曲线，给定一个 x，显示该 x 的点属于类别 1 的概率。随着 x 的增大，该点代表类别 1（通过）而不是类别 0（未通过）的概率也会增大。从曲线可以看出，如果 $x = 2$，那么该点属于类别 1 的概率不到 5%。但是，如果 $x = 10$，有大约 76%的概率属于类别 1。此时如果让你选择将这个点归为红点还是蓝点，你会得出结论说它是红点，因为它是红点的概率比蓝点大得多。

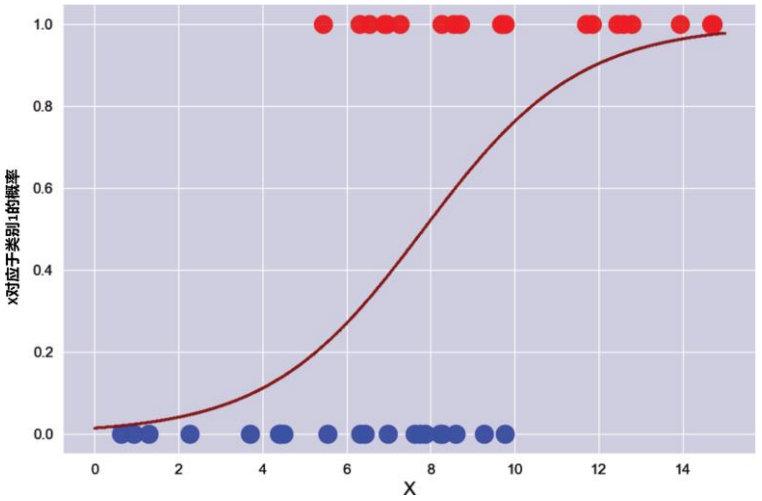


图 3-1 逻辑回归

图 3-1 中的曲线是一条 sigmoid（S 型）曲线。它描绘了所谓的 logistic 函数（也称为 logit 函数）（https://oreil.ly/tZTvE），在统计学中已经使用了几十年。对于逻辑回归，该函数是像下面这样定义的。其中，x 是输入值，m 和 b 是在训练期间学习的参数。

$$y = \frac{1}{1 + e^{-(mx + b)}}$$

这个方程式揭示了“逻辑回归”这个名字的来历。尽管它是一种分类算法，但 e 的指数恰好就是线性回归的方程式。

逻辑回归学习算法将 logistic 函数与一个数据分布拟合，并将得到的 y 值作为概率对数据点进行分类。它适用于任何数量的特征（不仅仅是 x，还有 x_1、x_2、x_3，等等）。另外，它还

是一种参数化学习算法，因为它使用训练数据来找出 m 和 b 的最佳值。具体如何找出最佳值是一种实现细节，由 Scikit-Learn 这样的库帮你处理。Scikit 默认使用一种称为限制内存 Broyden-Fletcher-Goldfarb-Shanno（L-BFGS）的数值优化算法（https://oreil.ly/hhJw0），但也支持其他优化方法。顺带一提，这也是 Scikit 在机器学习界如此受欢迎的原因之一。对于线性回归模型来说，从训练数据中计算出 m 和 b 并不难，但对于逻辑回归模型来说就比较难了，更不用说像“支持向量机”这样的更复杂的参数化模型。

Scikit 的 `LogisticRegression` 类（https://oreil.ly/wpwGs）封装好了逻辑回归算法。使用这个类，训练一个逻辑回归模型就可以像下面这样简单：

```
model = LogisticRegression()
model.fit(x, y)
```

一旦模型训练完成，就可以调用它的 `predict` 方法来预测输入属于哪个类别，或者调用它的 `predict_proba` 方法来获得计算出来的属于每一类的概率。如果为图 3-1 中的数据集拟合了一个 `LogisticRegression` 模型，那么以下语句可以预测 $x = 10$ 对应的是类别 1 还是 0：

```
predicted_class = model.predict([[10.0]])[0]
print(predicted_class) # 输出1
```

以下语句则显示了为每个类别计算的概率：

```
predicted_probabilities = model.predict_proba([[10.0]])[0]
print(f'Class 0: {predicted_probabilities[0]}') # 0.23508543966167028
print(f'Class 1: {predicted_probabilities[1]}') # 0.7649145603383297
```

Scikit 还包括 `LogisticRegressionCV` 类（https://oreil.ly/EPjhI），用于训练具有内置交叉验证功能的逻辑回归模型（第 2 章已经介绍过交叉验证）。以付出额外训练时间为代价，以下语句使用 5 折来训练一个逻辑回归模型：

```
model = LogisticRegressionCV(cv=5)
model.fit(x, y)
```

从技术上说，逻辑回归是一种二分类算法，但它也可以用于多分类。本章末尾对此进行详细说明。现在，请将逻辑回归看作是一种机器学习算法，它使用著名的 logistic 函数来量化一个输入对应于两个类别的概率。这样，就可以对逻辑回归有一个准确的概念性理解。

3.2 分类模型的准确率度量

可以用和回归模型一样的办法来量化分类模型的准确率，即调用模型的 `score` 方法。对于

分类器，score 返回真阳性（True Positive，TP）和真阴性（True Negative，TN）[1]之和除以样本总数。例如，假定测试数据包括 10 个阳性样本（类别 1 的 样本）和 10 个阴性样本（类别 0 的样本），而其中有 8 个阳性样本和 7 个阴性样本被模型正确识别，那么得分就是 (8 + 7) / 20，或者 0.75。这有时被称为模型的准确率分数（accuracy score）。

还有其他许多方法来为分类模型打分，具体哪一种种最合适，往往要取决于打算如何使用该模型。数据科学家有时并不计算准确率分数。相反，他们度量一个分类模型的精确率和召回率（https://oreil.ly/YJwCM）。

- 精确率（Precision）：真阳性数除以真阳性与假*阳性*数之和。
- 召回率（Recall）：真阳性数除以真阳性与假*阴性*数之和。

从本质上说，精确率会对假阳性（模型错误预测了一个 1 的情况）进行惩罚，召回率则通过降低模型错误预测了一个 0 时的分数来惩罚假阴性。

图 3-2 说明了这一区别。假设要训练一个模型来区分北极熊图像和海象图像。为了测试它，你提交了三张北极熊图像和三张海象图像。另外，假设该模型正确分类了两张北极熊图像，但错误地将两张海象图像分类为北极熊图像，如红框所示。在这种情况下，模型识别北极熊的精确率为 50%，因为在模型分类为北极熊的四张图中，实际只有两张是北极熊。但是，召回率是 67%，因为模型正确识别了三张北极熊图片中的两张。我们可以这样概括精确率和召回率：前者量化你相信一个阳性预测（positive prediction）有多准确，后者量化模型准确识别阳性样本（positive samples）的能力。这两者可以通过一个简单的公式合并成一个分数，称为 F1 分数（也称为 F 分数）（https://oreil.ly/qGnR0）。

图 3-2 使用精确率（precision）和召回率（recall）来度量分类器的准确率（accuracy）

[1] 译注：是的，你猜得没错——所有这些术语，包括真阳性（真正例：预测为正，实际也为正）、真阴性（真负例：预测为负、实际也为负）、假阳性（假正例：预测为正，实际为负）和假阴性（假负例：预测与负、实际为正）等，都借鉴自医疗行业。

Scikit 提供了一些辅助函数来帮助我们获取分类指标，例如 `precision_score`（https://oreil.ly/2x7yM）、`recall_score`（https://oreil.ly/xp7tO）和 `f1_score`（https://oreil.ly/fUu9A）。具体应该选择精确率还是召回率，要取决于假阳性（False Positives，FP）的成本和假阴性（False Negatives，FN）的成本哪个更高。当假阳性的成本很高时，就使用精确率——例如，如果“阳性”意味着将一封电子邮件识别为垃圾邮件。你宁愿垃圾邮件过滤器将几封垃圾邮件放入收件箱，也不愿意将一封合法的（可能是重要的）电子邮件放到垃圾邮件文件夹。相反，当假阴性的成本很高时，就使用召回率。一个很好的例子是使用机器学习在 X 射线和磁共振成像（MRI）扫描中发现肿瘤。你宁愿因为一个假阳性结果而把病人送去复查，也不愿在病人真的有肿瘤的时候告诉他没有肿瘤。

在野外发现北极熊

北极熊与海象的例子摘自我为微软写的一个教程，该教程开头是这样介绍的：

“你带领着一群气候科学家，他们对北极地区不断减少的北极熊数量感到担忧。为了解决这个问题，团队在整个地区的战略位置安放了数百个动态侦测摄像头。现在的挑战是设计一个自动化系统，实时处理来自这些摄像头的数据，并在其中一个摄像头拍摄到北极熊时在地图上显示警报，而不是人工检查每张照片以判断其中是否有北极熊。需要一个解决方案，利用人工智能（AI）来高度准确地判断一张照片是否包含北极熊。而且需要快速搞定这件事情，因为气候变化不等人。”

该教程结合了几个 Azure 服务，以形成一个端到端的解决方案，而且它使用微软的 Power BI 来进行可视化。感兴趣的读者可以在线查看（https://oreil.ly/Q8Bwb）。

准确率、精确率、召回率和 F1 分数适用于二分类*以及*多分类模型。还有一个指标仅适用于二分类，称为接收者操作特征（Receiver Operating Characteristic，ROC）曲线（https://oreil.ly/BDvv9），它绘制了不同概率阈值下的真阳性率（True-Positive Rate，TPR）与假阳性率（False-Positive Rate，FPR）。图 3-3 展示了一条示例 ROC 曲线。一条从左下角延伸到右上角的直线表明模型在 50%的时间内是正确的，对于二分类器来说，这和瞎猜没有区别。曲线越向左上角弯曲，模型就越准确。数据科学家经常使用曲线下面积（Area Under the Curve，AUC 或 ROC AUC）作为衡量准确率的总体指标。Scikit 提供了一个名为 `RocCurveDisplay`（https://oreil.ly/mJePr）的类来绘制 ROC 曲线，以及一个名为 `roc_auc_score`（https://oreil.ly/ryuxL）的函数来获取 ROC AUC 分数。这个函数返回的分数是 0.0 到 1.0 的值。分数越高，模型越准确。

为了评估分类模型的准确率，还有另一种方式是绘制混淆矩阵（confusion matrix）（https://oreil.ly/YxADc），如图 3-4 所示。它适用于二分类和多分类，为每个类别显示了模型在测试中的表现。在本例中，该模型被要求区分戴口罩（masked）和不戴口罩（unmasked）的面部图像。在出示戴口罩的面部图像时，它在 85 次中有 78 次是正确的。而在出示不戴口罩的人的面部图像时，58 次中有 41 次正确。Scikit 提供了一个计算混淆矩阵的 `confusion_matrix` 函数（https://oreil.ly/U4JBb）。还提供了一个

`ConfusionMatrixDisplay` 类（https://oreil.ly/CLnP2），其中有名为 `from_estimator`（https://oreil.ly/2XdMZ）和 `from_predictions`（https://oreil.ly/6sXhx）的方法用于绘制混淆矩阵。

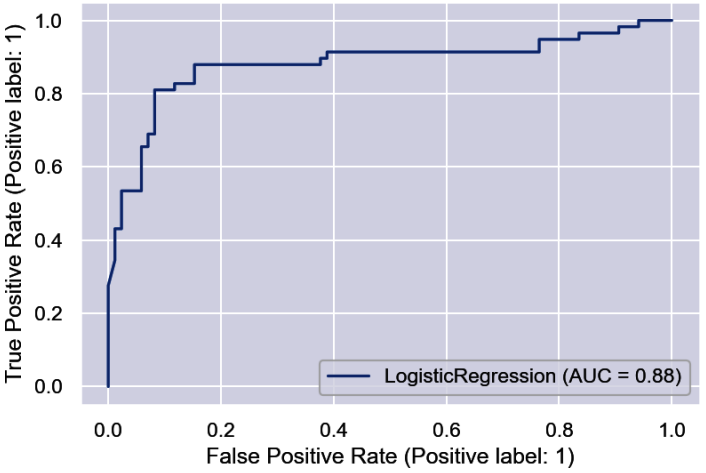


图 3-3 接收者操作特征（ROC）曲线

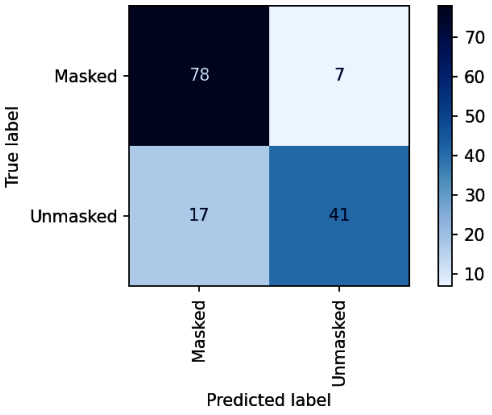


图 3-4 混淆矩阵

网上有无数代码示例使用 Scikit 的 `plot_confusion_matrix` 函数来显示混淆矩阵。目前，自 Scikit 1.0 引入的 `ConfusionMatrixDisplay` 才是生成混淆矩阵的正确方式。`plot_confusion_matrix` 预计将在 1.2 版本中从库中删除。

在讨论分类模型的准确率时，你可能遇到的其他术语还有灵敏度和特异度

（https://oreil.ly/EhfoW）等。其中，灵敏度（sensitivity）与召回率完全一致，所以 Scikit 没有单独包括计算它的方法。特异度（specificity）是针对阴性（负）类别而不是阳性（正）类别的召回率，计算方法是用真阴性数除以真阴性和假阳性数之和。Scikit 也没有提供专门的函数来计算特异性。但是，可以通过调用 `recall_score` 来轻松实现，用 `pos_label` 参数指定 0（而非 1）是阳性标签（positive label）即可。

```
recall_score(y_test, y_predicted, pos_label=0)
```

灵敏度和特异度在药物测试和癌症筛查中经常使用。假设你在国外旅行，要求在回国前的核酸检测结果为阴性。如果没有感染新冠，那么检测结果错误地说你感染了的几率有多大？（我最近在海外旅行时曾多次问自己这个问题。）答案是核酸检测的特异度——衡量检测在识别阴性样本时有多准确。另一方面，灵敏度揭示了如果检测结果说你感染了新冠，那么它有多大可能是正确的。这是一个微妙的区别，但是如果你担心一个错误的检测可能会造成自己无法回家（特异度），或者如果想在探望长辈之前确定自己没有感染新冠（灵敏度），那么这个区别就很重要了。

3.3 分类数据

机器学习从数字中寻找模式（patterns）。它*只*对数字起作用。然而，许多数据集都有包含了字符串值的列。这些字符串包括`"male"`和`"female"`，或者`"red"`、`"green"`和`"blue"`等。数据科学家将这些称为分类值（categorical values），将包含它们的列称为分类列（categorical columns）。机器学习不能直接处理分类值。要在模型中使用它们，必须将它们转换为数字。

有两种流行的技术可以将分类值转换为数值。一种是标签编码（label encoding），第 1 章的 K-means 聚类例子已经简要地展示了这个技术。标签编码将分类值替换为整数。如果一列有三个唯一的值，标签编码将它们替换为 0、1 和 2。为了体会它，请在 Jupyter 笔记本中运行以下代码：

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder

data = [[10, 'red'], [20, 'blue'], [12, 'red'], [16, 'green'], [22, 'blue']]
df = pd.DataFrame(data, columns=['Length', 'Color'])

encoder = LabelEncoder()
df['Color'] = encoder.fit_transform(df['Color'])
df.head()
```

上述代码创建了一个 `DataFrame`，包含一个名为 `Length` 的数字列和一个名为 `Color` 的分类列。分类列中包含三个不同的分类值。下面是编码前的数据集的样子：

	Length	Color
0	10	red
1	20	blue
2	12	red
3	16	green
4	22	blue

下面是使用 Scikit 的 `LabelEncoder` 类对 `Color` 列中的值进行标签编码后的样子：

	Length	Color
0	10	2
1	20	0
2	12	2
3	16	1
4	22	0

编码后的数据集可以用来训练机器学习模型。未编码的数据集则不能。可以从编码器的 `classes_`属性中得到编码类别的一个有序列表。

为了将分类值转换为数值，另一个常见的方法是独热编码（one-hot encoding，或称一位有效编码），它为分类列中的每个唯一值在数据集中添加一列，并在这些编码列中填充 1 和 0。独热编码可以通过 Scikit 的 `OneHotEncoder` 类（https://oreil.ly/fkDgF）或者在一个 Pandas `DataFrame` 上调用 `get_dummies`（https://oreil.ly/hqMPs）进行。下例演示了如何用后者编码数据集。

```
data = [[10, 'red'], [20, 'blue'], [12, 'red'], [16, 'green'], [22, 'blue']]
df = pd.DataFrame(data, columns=['Length', 'Color'])

df = pd.get_dummies(df, columns=['Color'])
df.head()
```

下面是结果：

	Length	Color_blue	Color_green	Color_red
0	10	0	0	1
1	20	1	0	0
2	12	0	0	1
3	16	0	1	0
4	22	1	0	0

标签编码和独热编码都可用于回归问题和分类问题。你肯定想问，到底应该使用哪一个？一般来说，数据科学家更喜欢独热编码而不是标签编码。前者为每个独特的值赋予了相同的权重，而后者意味着一些值可能比其他值更重要——例如，"red"（2）比"blue"（0）更重要。另一方面，标签编码的内存效率更高。标签编码前后的列数不会变化，而独热编码要为每个独特的值增加一列。对于一个非常大的数据集，如果它的一个分类列中有数千个唯一值，那么标签编码所需的内存会少得多。

机器学习模型的准确率很少受到你选择的编码方法的影响。如果存有疑虑，那么无脑选择独热编码，这很少会出错。如果想进行确定，那么可以同时对数据进行两种方式的编码，并在训练了机器学习模型后比较结果。

3.4 二分类

二分类器（binary classifiers）是用带标签的数据进行训练的监督学习模型。0 代表阴性类别（negative class），1 代表阳性类别（positive class）。它们做出的预测也是 0 和 1。它们还透露了属于每一类别的概率，你可以将其纳入自己的结论。例如，一家信用卡公司可能会决定，只有在模型至少有 99%的把握预测一笔交易具有欺诈性的情况下，才拒绝该交易。在这种情况下，模型计算出的概率比它做出的原始预测更重要。

为了帮助大家了解到目前为止所介绍的关于二分类的一切，让我们用 Scikit 来构建两个模型：首先是一个简单的模型，展示了核心原则，然后是第二个模型，解决了一个真正的商业问题。

3.4.1 对泰坦尼克号乘客进行分类

机器学习著名的公共数据集之一是泰坦尼克号数据集（https://oreil.ly/JtxlV），它包含关于数百名泰坦尼克号乘客的信息，包括哪些人幸存，哪些人没有。让我们用逻辑回归从数据集建立一个二分类模型，看看我们是否能在给定一名乘客的性别、年龄和票价舱位（头等舱、二等舱还是三等舱）的前提下，预测他/她的幸存几率。

第一步是下载数据集（https://oreil.ly/vMnPI），并将其复制到 Jupyter 笔记本所在目录下的 Data 子目录。然后在笔记本中运行以下代码来加载数据集，体会一下它的内容：

```
import pandas as pd

df = pd.read_csv('Data/titanic.csv')
df.head()
```

下面是输出：

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

该数据集包含 891 行和 12 列。其中一些列，如 `PassengerId` 和 `Name`，与机器学习模型无关。其他列则非常相关。我们主要关注的是：

- `Survived`：表明是幸存(1)还是没有幸存(0)。
- `Pclass`：表明乘客乘坐的是头等舱 (1)，二等舱(2)，还是三等舱(3)。
- `Sex`：表明乘客的性别
- `Age`：表明乘客的年龄

`Survived` 列是标签列——即我们要尝试预测的列。其他列跟这个预测相关。头等舱乘客更有可能在沉没中幸存下来（https://oreil.ly/i0N6s），因为他们的舱室更靠近船的顶层甲板，更靠近救生艇。另外，妇女和儿童更有可能在救生艇上获得空间。

现在，使用以下语句看看数据集是否缺失了任何值：

```
df.info()
```

下面是输出：

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

`Cabin`（船舱号）这一列缺少很多值，但我们不关心，因为不会用到这一列。`Age` 列是我们要用到的，但它也缺少一些值。可以用其他所有年龄的平均值来替换这些缺失的值，数据科学家将这种方法称为对缺失值进行插补（imputing）。但是，我们采用的是更简单的方法，

即删除含有缺失值的行。使用以下语句删除不需要的列，删除有缺失值的行，并对 Sex 和 Pclass 列中的值进行独热编码：

```
df = df[['Survived', 'Age', 'Sex', 'Pclass']]
df = pd.get_dummies(df, columns=['Sex', 'Pclass'])
df.dropna(inplace=True)
df.head()
```

下面是结果数据集：

	Survived	Age	Sex_female	Sex_male	Pclass_1	Pclass_2	Pclass_3
0	0	22.0	0	1	0	0	1
1	1	38.0	1	0	1	0	0
2	1	26.0	1	0	0	0	1
3	1	35.0	1	0	1	0	0
4	0	35.0	0	1	0	0	1

下个任务是针对训练和测试来拆分数据集：

```
from sklearn.model_selection import train_test_split

x = df.drop('Survived', axis=1)
y = df['Survived']

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,
                                                    stratify=y, random_state=0)
```

注意传递给 train_test_split 的 stratify=y 参数。它很重要，因为去除有缺失值的行后，在剩下的 714 个样本中，290 个代表幸存下来的乘客，424 个代表没有幸存下来的乘客。我们希望在训练数据集和测试数据集中包含相似比例的这两类乘客，而 stratify=y 就实现了这一点。不这样指定，模型最终会比实际情况更准确或更不准确。

现在创建一个逻辑回归模型，用拆分好的数据进行训练，并用测试数据进行评分。

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(random_state=0)
model.fit(x_train, y_train)
model.score(x_test, y_test)
```

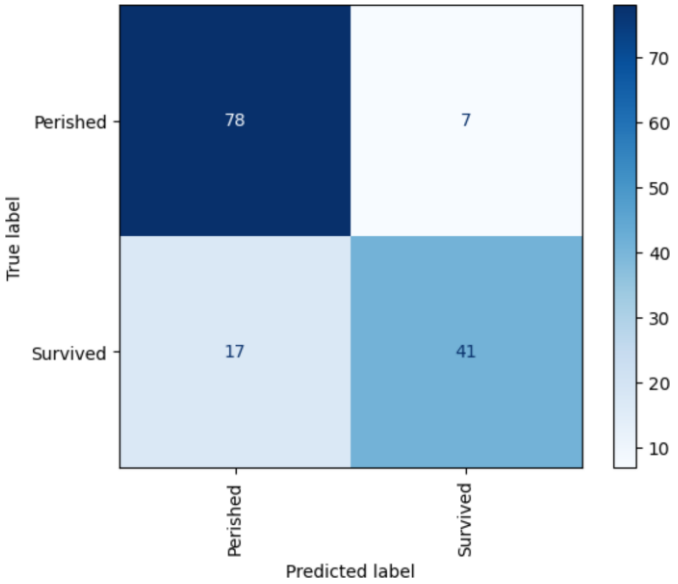
用交叉验证法再次对模型进行评分，以便对分数更有信心。记住，这是通过将真阳性和真阴性数相加，并除以样本总数计算出来的准确率分数。

```
from sklearn.model_selection import cross_val_score

cross_val_score(model, x, y, cv=5).mean()
```

使用以下语句来显示混淆矩阵，精确显示模型在测试期间的表现：

```
%matplotlib inline
from sklearn.metrics import ConfusionMatrixDisplay as cmd

cmd.from_estimator(model, x_test, y_test,
                   display_labels=['Perished', 'Survived'],
                   cmap='Blues', xticks_rotation='vertical')
```

下面是结果：



注意，当预测乘客*不会*幸存时，该模型比预测他们会幸存时更准确。这是因为在用于训练模型的数据集中，乘客死亡（Perished）的样本多于乘客幸存（Survived）的样本。我们当然喜欢用一个完全平衡的数据集来训练二分类模型，这种数据集包含相同数量的阳性和阴性样本。但是，如果在评估模型的准确率时考虑到了样本不平衡的问题，那么用不平衡的数据集来训练也是完全能够接受的。

现在，使用 Scikit 的 `precision_score` 和 `recall_score` 函数来计算模型的精确率、召回率、灵敏度和特异度：

```
from sklearn.metrics import precision_score, recall_score

y_pred = model.predict(x_test)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
sensitivity = recall
specificity = recall_score(y_test, y_pred, pos_label=0)

print(f'精确率: {precision}')
print(f'召回率: {recall}')
print(f'灵敏度: {sensitivity}')
print(f'特异度: {specificity}')
```

下面是结果：

```
精确率: 0.8541666666666666
召回率: 0.7068965517241379
灵敏度: 0.7068965517241379
特异度: 0.9176470588235294
```

我们知道，该模型更善于识别没有幸存下来的乘客，而不是那些幸存下来的乘客。高的特异度分数是否与这个观察结果相一致？如何解释相对较低的召回率和灵敏度分数？

现在，让我们用训练好的模型来做一些预测。首先，预测头等舱的一名 30 岁女性是否可能幸存。由于该模型是用包含多个列名称的一个 `DataFrame` 来训练的，所以我们将使用相同的列名称来制定一个输入：

```
female = pd.DataFrame({ 'Age': [30], 'Sex_female': [1], 'Sex_male': [0],
                        'Pclass_1': [1], 'Pclass_2': [0], 'Pclass_3': [0] })

model.predict(female)[0]
```

模型预测她能幸存下来，但幸存几率是多少呢？

```
probability = model.predict_proba(female)[0][1]
print(f'Probability of survival: {probability:.1%}')
```

所以，乘坐头等舱的一名 30 岁女性有 90%以上的机率幸存。但是，乘坐三等舱的一名 60 岁男性呢？

```
male = pd.DataFrame({ 'Age': [60], 'Sex_female': [0], 'Sex_male': [1],
                      'Pclass_1': [0], 'Pclass_2': [0], 'Pclass_3': [1] })

probability = model.predict_proba(male)[0][1]
print(f'Probability of survival: {probability:.1%}')
```

请自由尝试其他输入，看看模型怎么说。例如，二等舱的一名 12 岁男孩在泰坦尼克号沉没后幸存下来的机率有多大？

3.4.2 检测信用卡欺诈

今天，机器学习最引人注目的用途之一是发现欺诈性的金融交易。信用卡公司在销售点（POS）应用机器学习来决定是否接受或拒绝一次刷卡行为。这些公司不愿意公布细节或者他们用来训练模型的数据，但这是完全可以理解的。但是，至少有一个这样的数据集已经开放给公众使用。为了保护隐私，其中的数据是用一种称为主成分分析（Principal Component Analysis，PCA）的技术进行匿名的，具体将在第 6 章解释。

该数据集如图 3-5 所示。数据来自欧洲信用卡持有人在 2013 年 9 月的真实交易。大多数列都有隐晦的名称，例如 `V1` 和 `V2`，并包含同样不透明的值。其中，有三列——`Time`、`Amount` 和 `Class`——具有真实的名称和未经修改的值，分别揭示了刷卡交易发生的时间、交易金额以及交易是合法（Class=0）还是具有欺诈性（Class=1）。

	Time	V1	V2	V3	...	V27	V28	Amount	Class
0	0.0	-1.359807	-0.072781	2.536347	...	0.133558	-0.021053	149.62	0
1	0.0	1.191857	0.266151	0.166480	...	-0.008983	0.014724	2.69	0
2	1.0	-1.358354	-1.340163	1.773209	...	-0.055353	-0.059752	378.66	0
3	1.0	-0.966272	-0.185226	1.792993	...	0.062723	0.061458	123.50	0
4	2.0	-1.158233	0.877737	1.548718	...	0.219422	0.215153	69.99	0
5	2.0	-0.425966	0.960523	1.141109	...	0.253844	0.081080	3.67	0
6	4.0	1.229658	0.141004	0.045371	...	0.034507	0.005168	4.99	0
7	7.0	-0.644269	1.417964	1.074380	...	-1.206921	-1.085339	40.80	0
8	7.0	-0.894286	0.286157	-0.113192	...	0.011747	0.142404	93.20	0
9	9.0	-0.338262	1.119593	1.044367	...	0.246219	0.083076	3.68	0

图 3-5 欺诈检测数据集

每一行都代表一笔交易。在数据集中的 284 807 笔交易中，只有 492 笔是欺诈性的。由于该数据集高度不平衡，所以你自然预期对于基于该数据集进行训练的机器学习模型来说，在合法交易的分类上比欺诈交易准确得多。这其实不一定是个问题，因为信用卡公司宁可对欺诈性交易进行错误的分类，让其中 100 笔事实上非法的交易顺利通过，也不愿对一笔合法的交易进行错误的分类，从而激怒客户。

首先，下载一个包含数据集的 ZIP 文件（https://oreil.ly/EYbNK）。将 ZIP 文件中的 creditcard.csv 复制到笔记本所在目录的 Data 子目录中。然后，在 Jupyter 笔记本中运行以下代码来加载数据集并显示前几行：

```
import pandas as pd

df = pd.read_csv('Data/creditcard.csv')
df.head()
```

然后，了解数据集中包含多少行，这些行中是否有缺失的值：

```
df.info()
```

结果表明，该数据集包含 284 807 行，没有任何缺失值。接着，拆分数据以分别进行训练和测试，并在调用 `train_test_split` 方法时指定 `stratify` 参数来确保合法交易和欺诈交易的比例在训练数据集和测试数据集中是一致的。

```
from sklearn.model_selection import train_test_split

x = df.drop(['Time', 'Class'], axis=1)
y = df['Class']

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,
                                                    stratify=y, random_state=0)
```

然后，训练一个逻辑递归模型来划分不同类别：

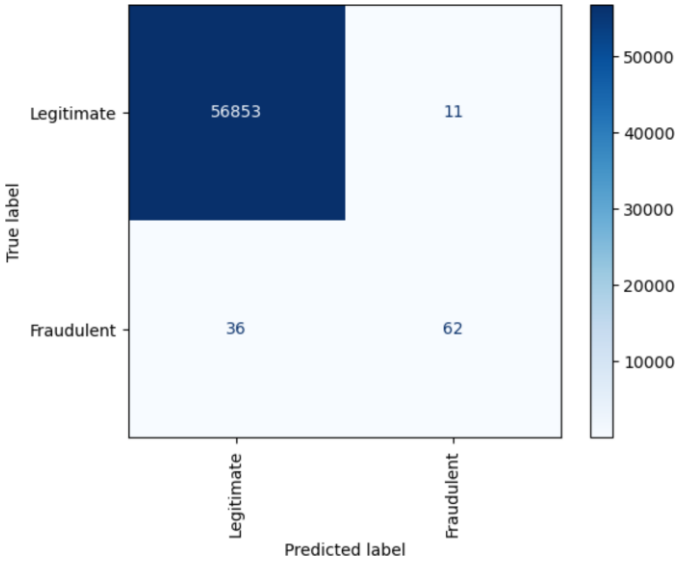
```
from sklearn.linear_model import LogisticRegression

lr_model = LogisticRegression(random_state=0, max_iter=5000)
lr_model.fit(x_train, y_train)
```

注意传递给 `LogisticRegression` 方法的 `max_iter=5000` 参数。`max_iter` 指定了在针对一个数据集拟合 logistic 函数时，允许收敛于一个解的最大迭代次数。默认值是 100，这在本例中是不够的。将这个限制提升至 5000，可以为内部求解器提供它所需要的空间来找到一个解。

用真阳性和真阴性数之和除以测试样本的数量，所计算出来的典型的准确率分数并不是很有帮助，因为数据集是如此的不平衡。欺诈性交易在所有样本中占比不到 0.2%，这意味着模型可以简单地猜测每笔交易都是合法的，而且 99.8%的时间都是正确的。让我们使用混淆矩阵来可视化模型在测试中的表现。

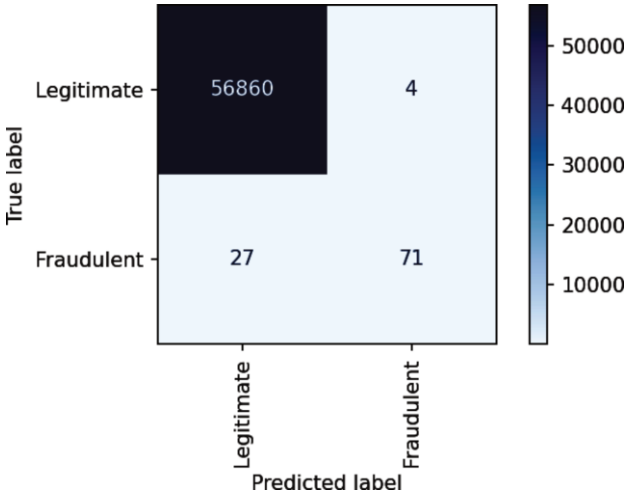
```
%matplotlib inline
from sklearn.metrics import ConfusionMatrixDisplay as cmd

labels = ['Legitimate', 'Fraudulent']
cmd.from_estimator(lr_model, x_test, y_test, display_labels=labels,
                   cmap='Blues', xticks_rotation='vertical')
```

下面是结果：



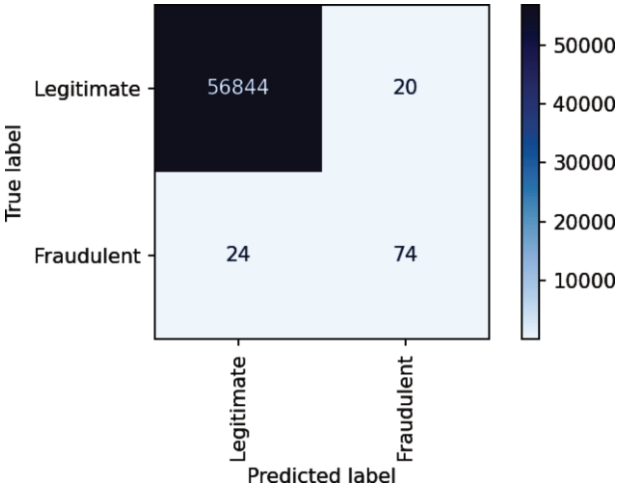
一个逻辑回归模型正确识别了 56 853 笔交易为合法交易，而将合法交易错误归类为欺诈性交易的次数只有 11 次。我们想尽量减少后一个数字，因为我们不想因为拒绝合法交易而惹恼客户。让我们看看随机森林分类器是否能做得更好：

```
from sklearn.ensemble import RandomForestClassifier

rf_model = RandomForestClassifier(random_state=0)
rf_model.fit(x_train, y_train)

cmd.from_estimator(rf_model, x_test, y_test, display_labels=labels,
                   cmap='Blues', xticks_rotation='vertical')
```



随机森林仅将 4 笔合法交易误认为是欺诈性的。这是对逻辑回归的一种改进。再来看看梯度提升分类器是否能做得更好。

```
from sklearn.ensemble import GradientBoostingClassifier

gbm_model = GradientBoostingClassifier(random_state=0)
gbm_model.fit(x_train, y_train)

cmd.from_estimator(gbm_model, x_test, y_test, display_labels=labels,
                   cmap='Blues', xticks_rotation='vertical')
```



梯度提升机（GBM）比随机森林错误地分类了更多的合法交易，所以我们坚持使用随机森林。在 56 864 笔合法交易中，随机森林正确分类了 56 860 笔。这意味着合法交易有99.99%以上的时间被正确分类。与此同时，该模型抓出了大约 75%的欺诈性交易。

使用以下语句来衡量随机森林分类器的精确率、召回率、灵敏度和特异度：

```
from sklearn.metrics import precision_score, recall_score

y_pred = rf_model.predict(x_test)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
sensitivity = recall
specificity = recall_score(y_test, y_pred, pos_label=0)

print(f'精确率: {precision}')
print(f'召回率: {recall}')
print(f'灵敏度: {sensitivity}')
print(f'特异度: {specificity}')
```

下面是输出：

```
精确率: 0.9466666666666667
召回率: 0.7244897959183674
灵敏度: 0.7244897959183674
特异度: 0.9999296567248172
```

鉴于信用卡公司一方面希望客户快乐，另一方面希望客户消费，所以你认为他们对这些指标中的哪个最感兴趣？如果你的回答是特异性就对了。特异性衡量的是测试在不将阴性样本错误地归为阳性的可靠程度——在本例中，即为不将合法交易归类为欺诈（类似于在之前的例子中，不将核酸检测的阴性样本归为阳性）。

遗憾的是，我们无法用这个模型进行预测，因为不知道 `V1` 到 `V28` 列中各个数字的含义，也无法从一笔新的交易中生成列值，因为不知道当初对原始数据集是如何转换的。甚至不知道原始数据集是什么样子的。最有可能的是，每一行都包含了关于持卡人的信息——例

如，年收入、信用分、年龄、居住国和去年刷卡金额。另外，还有关于购买的产品和刷卡地点的信息。机器学习在特征工程（feature-engineering）方面——弄清楚哪些数据与试图建立的模型有关——与数据准备和选择学习算法一样具有挑战性，甚至更有挑战性。

在现实生活中，信用卡公司用来检测欺诈行为的模型比这更复杂，而且它们通常包含几个模型，因为没有一个能保证百分百准确。例如，一家公司可能会建立三个模型，并让它们对一笔给定的交易是否合法进行投票。无论如何，你现在已经证明了这样一个原则：只要有正确的特征，就能建立一个分类模型，在检测信用卡欺诈方面具有相当的准确率。另外，你亲眼见证了 Scikit 允许你很容易地试验不同的学习算法，以确定哪种算法能生成最有用的模型。

3.5 多分类

现在是时候讨论多分类（multiclass classification）了，它允许有 N 个可能的结果，而不是像二分类那样只有两个。多分类的一个很好的例子是光学字符识别（OCR）：检查一个手写的数字并预测它对应的数字（0~9）。另一个例子是检查一张面部照片，并通过一个训练好的模型来识别照片中的人，该模型能识别数百个人。

到目前为止学到的关于二分类的几乎所有知识也适用于多分类。在 Scikit 中，任何适用于二分类的分类器也适用于多分类模型。这一点的重要性怎么强调都不为过。一些学习算法，例如逻辑回归，只在二分类场景下工作。许多机器学习库要求写显式的代码来扩展逻辑回归以执行多分类，或者使用一种形式全然不同的逻辑回归。但是，Scikit 不是这样设计的。相反，它确保 `LogisticRegression`（逻辑回归）等分类器在两种情况下都能工作。必要时，它会在幕后做额外的工作来确保这一点。

对于逻辑回归，Scikit 使用两种策略中的一种来扩展算法，使其能在多分类情况下工作。（可用 `LogisticRegression` 类的 `multi_class` 参数来指定使用的策略，或者接受默认的 `"auto"`，让 Scikit 自己选择。）其中一个策略是多分类逻辑回归（multinomial logistic regression），它用一个 softmax 函数替代了 logistic 函数(https://oreil.ly/BXtMC)，能生成多个概率，每个类别一个（one-vs-one）。另一个策略是 one-vs-rest，也称为 one-vs-all，它训练 *n* 个二分类模型，其中 *n* 是模型可以预测的类别的数量。*n* 个模型中的每一个都将一个类别与其他所有类别配对。要求模型做出预测时，它通过全部 *n* 个模型运行输入，最终使用的是能产生最高概率的那个模型的输出。图 3-6 展示了这个策略。

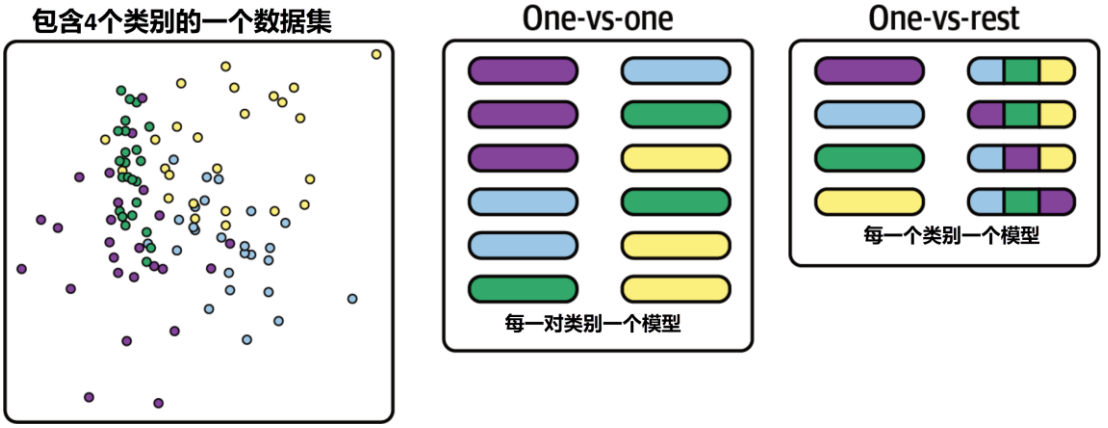


图 3-6 扩展二分类算法以支持多分类的策略

one-vs-rest方法对逻辑回归很有效，但是对于一些仅二分类的分类算法，Scikit会用one-vs-one方法来代替。例如，使用Scikit的SVC类（一种支持向量机分类器，详情参见第5章）进行多分类时，Scikit 会为每一对类别建立一个模型。如果模型包括 4 个可能的类，那么Scikit会在幕后构建不少于7个模型。

构建多分类模型时，其实并不需要知道所有这些细节。但是，它确实解释了为什么一些多分类模型需要更多的内存，而且训练速度比其他的要慢。一些分类算法，例如随机森林和GBM（梯度提升机），本身就支持多分类。对于那些不支持的算法，Scikit 是你的后盾。它填补了这一空白，并尽可能地透明化。

这里需要重申一下：所有 Scikit 分类器都能进行二分类和多分类。这简化了编码，让你专注于模型的构建和训练，而不必理解算法的底层机制。

3.6 构建数字识别模型

想亲身体验一下多分类吗？一个检查扫描的手写数字，并预测它们和 0~9 中哪一个对应的模型怎么样？美国邮政总局（USPS）多年前建立了一个类似的模型来识别手写的邮政编码，作为邮件分类自动化努力的一部分。我们将使用 Scikit 内置的一个样本数据集：加州大学欧文分校的手写数字 OCR 数据集（https://oreil.ly/i55Ob），其中包含近 1800 个手写数字。每个数字由一个 8×8 的数字数组表示。其中，每个数字的范围是从 0 到 16，数字越大表示像素越暗。我们将使用逻辑回归从这些数据中预测。图 3-7 展示了数据集中的前 50 个数字。

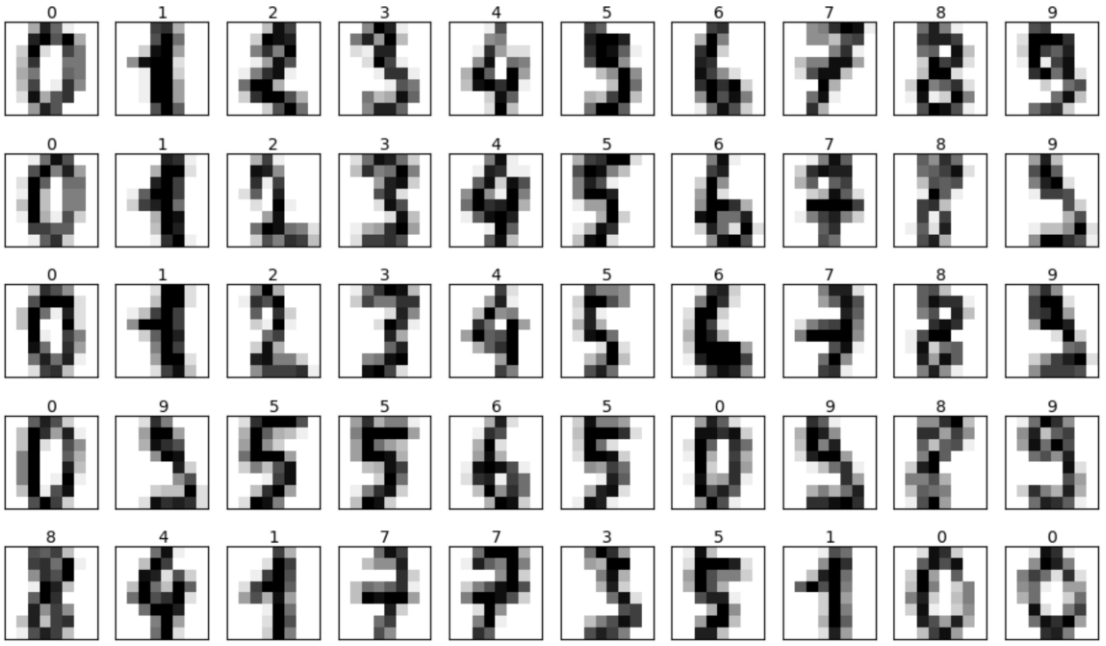


图 3-7 手写数字 OCR 数据集中的前 50 个数字

首先创建一个 Jupyter 笔记本，在第一个单元格中执行以下语句：

```
from sklearn import datasets

digits = datasets.load_digits()
print('digits.images: ' + str(digits.images.shape))
print('digits.target: ' + str(digits.target.shape))
```

下面是第一个数字的数值形式：

```
digits.images[0]
```

以下语句显示人眼看到的结果：

```
%matplotlib inline
import matplotlib.pyplot as plt

plt.tick_params(axis='both', which='both', bottom=False, top=False, left=False,
                right=False, labelbottom=False, labelleft=False)
plt.imshow(digits.images[0], cmap=plt.cm.gray_r)
```

这显然是一个 0，但可以从它的标签中确认这一点：

```
digits.target[0]
```

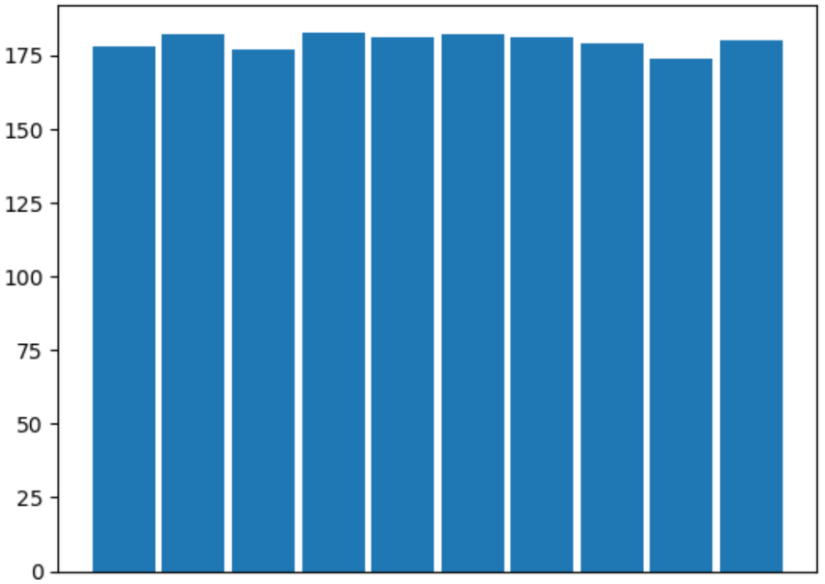
绘制前 50 张图，并显示相应的标签：

```
fig, axes = plt.subplots(5, 10, figsize=(12, 7),
                         subplot_kw={'xticks': [], 'yticks': []})

for i, ax in enumerate(axes.flat):
    ax.imshow(digits.images[i], cmap=plt.cm.gray_r)
    ax.text(0.45, 1.05, str(digits.target[i]), transform=ax.transAxes)
```

分类模型在平衡的数据集上效果最好。使用以下语句来绘制样本的分布：

```
plt.xticks([])
plt.hist(digits.target, rwidth=0.9)
```

下面是输出：



数据集几乎是完全平衡的，所以让我们拆分它来训练一个逻辑回归模型：

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(
    digits.data, digits.target, test_size=0.2, random_state=0)

model = LogisticRegression(max_iter=5000)
model.fit(x_train, y_train)
```

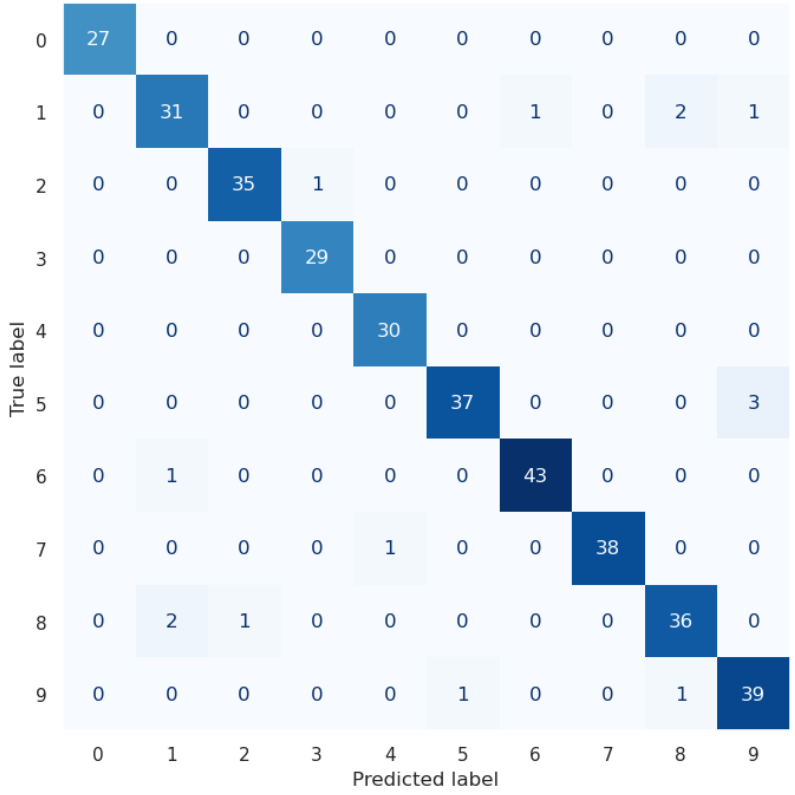
使用 score 来量化模型的准确率：

```
model.score(x_test, y_test)
```

使用一个混淆矩阵，看看模型在测试数据集上的表现如何：

```
%matplotlib inline
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay as cmd
import seaborn as sns
sns.set()

fig, ax = plt.subplots(figsize=(8, 8))
ax.grid(False)
cmd.from_estimator(model, x_test, y_test, cmap='Blues', colorbar=False, ax=ax)
```

由此产生的输出描绘了一个令人鼓舞的画面：沿对角线的是大数字和深色，对角线外的是小数字和浅色。一个完美的模型在对角线外全是零。但是，当然，完美的模型并不存在。



从数据集中选取一个数字，并绘制它，看看它是什么样子的：

```
sns.reset_orig() # Undo sns.set()
plt.tick_params(axis='both', which='both', bottom=False, top=False, left=False,
                right=False, labelbottom=False, labelleft=False)
plt.imshow(digits.images[100], cmap=plt.cm.gray_r)
```

把它传给模型，模型预测它是什么数字？

```
model.predict([digits.data[100]])[0]
```

模型预测它是每个可能的数字的概率是多少？

```
model.predict_proba([digits.data[100]])
```

预测这个数字是 4 的概率是多少？

```
model.predict_proba([digits.data[100]])[0][4]
```

当用于二分类时，`predict_proba` 返回两个概率：一个用于阴性类别（negative class），一个用于阳性类别（positive class）。对于多分类，`predict_proba` 则返回属于每个可能的类别的概率。这样一来，就可以评估模型对 `predict` 返回的预测结果的信心。毫不奇怪，`predict` 返回的是被赋予最高概率的类别。

3.7 小结

分类模型在业界被广泛用于预测分类结果，例如一笔刷卡交易是该接受还是拒绝。二分类模型预测两个结果中的一个，多分类模型则预测两个以上的结果。分类模型最常用的学习算法之一是逻辑回归，它为训练数据拟合一个方程式。为了预测结果，它计算每种类别是正确的概率，并选择概率最高的结果。

有许多方法可以对分类模型进行评分。具体应该选择哪种方法，要取决于你打算如何使用该模型。例如，当假阳性的成本很高时，使用精确率来评估模型的准确率。精确率的计算方法是用真阳性数除以真阳性数与假*阳性*数之和。另一方面，如果假阴性的成本很高，就用召回率代替。召回率的计算方法是用真阳性数除以真阳性数与假*阴性*数之和。与精确率和召回率密切相关的是灵敏度和特异度。灵敏度与召回率完全一致，而特异度其实也是召回率，只是用阴性类别取代了阳性类别。混淆矩阵提供了一种方便的方法，可以直观地看到一个模型在测试数据上的表现，而不是显示一个干巴巴的准确率数字。

有的学习算法只适用于二分类，但 Scikit 在幕后施展了一些魔法，确保*任何*学习算法都能用于二分类或多分类。这是 Scikit 机器学习库的一个特点，其他库并非都是如此。有的库将其学习算法限制在特定的场景中，或者要求你写额外的代码，才能在多分类模型中使用一个二分类算法。

本章所有模型都是用数值数据进行训练的，尽管有的数据集包含分类值——即字符串而非数值，而它们必须用独热编码转换为数字。你可能想知道如何构建一个*只*针对文本的分类模型——例如，一个为餐厅评价打分或者将电子邮件分类为垃圾/非垃圾邮件的模型。这非常合理，将是第 4 章的主题。

第 4 章 文本分类

二分类的一个更新颖的用途是情感分析（sentiment analysis），它检查一个文本样本，例如一条产品评价、一条推特或者网站上的一条评价，并在 0.0 到 1.0 的范围内打分。其中，0.0 代表负面情感，1.0 代表正面情感。诸如“物美价廉”这样的评价可能会得到 0.9 分，而“太贵了，也不好用”可能会得到 0.1 分。这个分数反映了文本表达积极情感的概率。情感分析模型很难用算法建立，但用机器学习来打造则很容易。要想更多地了解情感分析当前的商用情况，请参考 Nicholas Bianchi 的文章“情感分析的 8 个现实用例”（https://oreil.ly/nWq4a）。

情感分析是在文本数据而非数值数据上进行分类任务的一个例子。由于机器学习只能用数字进行，所以在训练一个情感分析模型、识别垃圾邮件的模型或者其他任何对文本进行分类的模型之前，都必须先将文本转换成数字。一个常见的方法是构建词频表，称为词袋（bag of words）（https://oreil.ly/W4M6O）。Scikit-Learn 提供了一些帮助。它还支持文本的归一化。例如，使“awesome”和“Awesome”不会被算作两个不同的词。

本章首先描述了如何准备要在分类模型中使用的文本。在构建了一个情感分析模型之后，我们将讨论另一种流行的学习算法，即朴素贝叶斯（Naive Bayes），它对文本的处理效果特别好。我们将使用它来构建一个模型，以区分合法邮件和垃圾邮件。最后，将讨论一种度量两个文本样本相似度的数学技术，并用它来构建一个应用程序，根据你喜欢的其他电影来推荐电影。

4.1 准备用于分类的文本

在训练一个模型对文本进行分类之前，必须先将文本转换成数字，这个过程称为矢量化（vectorization）[①]。图 4-1 重现了第 1 章的图，它展示了一种常见的文本矢量化技术。每行代表一个文本样本，例如一条推特或者一条影评。每一列则代表训练文本中的一个词。行中的数字是词数（word counts），每行的最后一个数字是标签（label）：0 代表负面情感，1 代表正面情感。

[①] 译注：本书中文版将混合使用“矢量”和“向量”。两者没有本质的区别。

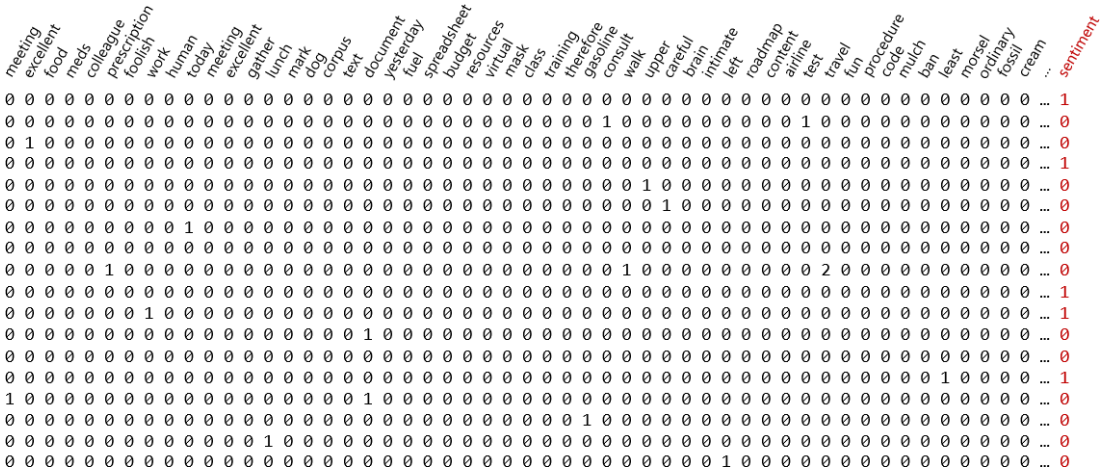


图 4-1 用于情感分析的数据集

文本在矢量化之前通常需要先清洗一遍。清洗（cleaning）的例子包括转换成小写（例如，“Excellent”相当于“excellent”），去除标点符号，并选择性地去除停用词（stop word）——像 the 和 and 这样常见的词可能对结果影响不大。一旦清洗完毕，句子就会被分成单个的词（即 tokenized，或者说对句子进行“分词”），这些词被用来生成像图 4-1 那样的数据集。

Scikit-Learn 提供了三个类来完成文本清洗和矢量化的繁重工作：

- `CountVectorizer` (https://oreil.ly/00918)：从训练文本的语料库中创建一个字典（或称 vocabulary，词汇表），并生成一个像图 4-1 那样的词数矩阵。
- `HashingVectorizer` (https://oreil.ly/vTJ4b)：使用词哈希（word hashes）而不是内存中的词汇来生成词数，因此更节省内存。
- `TfidfVectorizer` (https://oreil.ly/xzaYZ)：从提供给它的词中创建一个字典，并生成一个像图 4-1 那样的矩阵，但该矩阵不包含整数词数，而是包含 0.0~1.0 的词频-逆文档频率（Term Frequency-Inverse Document Frequency，TFIDF）（https://oreil.ly/NQrDJ）值，反映单个词的相对重要性。

所有这三个类都能将文本转换为小写字母，去除标点符号，去除停用词，将句子拆分为单独的词，等等。它们还支持 *n*-grams，即两个或更多连续的词（数字 *n* 由你指定）的组合应被视为一个词。它背后的思路在于，像信用（credit）和分（score）这样的词如果在一个句子中紧挨在一起出现，可能比它们相隔很远出现更有意义。如果没有 *n*-grams，词的相对接近性就会被忽略。使用 *n*-grams 的缺点在于，它增加了内存消耗和训练时间。然而，只要能合理地使用，它能使文本分类模型更加准确。

神经网络有其他更强大的方法将词序考虑进去，而不要求相关的词紧挨在一起出。传统的机器学习模型无法将“我喜欢蓝色，因为它是天空的颜色”这句话中的蓝色和天空联系起来，但神经网络可以。我将在第 13 章进一步说明这一点。

下例演示了 CountVectorizer 的作用及其用法：

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer

lines = [
    'Four score and 7 years ago our fathers brought forth,',
    '... a new NATION, conceived in liberty $$$,',
    'and dedicated to the PrOpOsItIoN that all men are created equal',
    'One nation\'s freedom equals #freedom for another $nation!'
]

# Vectorize the lines
vectorizer = CountVectorizer(stop_words='english')
word_matrix = vectorizer.fit_transform(lines)

# Show the resulting word matrix
feature_names = vectorizer.get_feature_names_out()
line_names = [f'Line {(i + 1):d}' for i, _ in enumerate(word_matrix)]

df = pd.DataFrame(data=word_matrix.toarray(), index=line_names,
                  columns=feature_names)

df.head()
```

下面是输出：

	ago	brought	conceived	created	dedicated	equal	equals	fathers	forth	freedom	liberty	men	nation	new	proposition	score	years
Line 1	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1
Line 2	0	0	1	0	0	0	0	0	0	0	1	0	1	1	0	0	0
Line 3	0	0	0	1	1	1	0	0	0	0	0	1	0	0	1	0	0
Line 4	0	0	0	0	0	0	1	0	0	2	0	0	2	0	0	0	0

本例的文本语料库是一个 Python 列表中的 4 个字符串。CountVectorizer 将这些字符串拆分为词，删除停用词和符号，并将剩余所有词转换为小写。这些词构成了数据集中的列，行中的数字则代表一个给定的词在每个字符串中出现的次数。stop_words='english'参数告诉 CountVectorizer 使用包含了 300 多个英语停用词的一个内置字典来删除停用词。如果愿意，也可以用一个 Python 列表提供自己的停用词列表（还可以选择保留停用词，它们经常都并不重要）。用另一种语言的文本进行训练时，可以从其他 Python 库获得多语言的停用词列表，例如 Natural Language Toolkit（NLTK）（https://oreil.ly/2WzKr）和 Stopwords(https://oreil.ly/Z4mRJ)。

从输出结果可知，尽管含义相似，但 equal 和 equals 被算成单独的词。数据科学家在为机器学习准备文本时，有时还会更进一步，对单词进行词干提取（stemming）

（https://oreil.ly/q5ZhR）或词形还原（lemmatizing）（https://oreil.ly/BbiUx）。对上述文本进行词干提取，出现的所有 equals 都会被转换为 equal。Scikit 缺乏对词干提取和词形还原的支持，但这个功能可从其他库（如 NLTK）中获得。

`CountVectorizer` 会删除标点符号，但不会删除数字。第 1 行中的 7 之所以被忽略了，因为它会忽略单字符。但是，如果把 7 改为 777，777 这个词就会出现在词汇表中。为了解决这个问题，一个方法是定义一个删除数字的函数，并通过 `preprocessor` 参数把它其传递给 `CountVectorizer`。

```
import re

def preprocess_text(text):
    return re.sub(r'\d+', '', text).lower()

vectorizer = CountVectorizer(stop_words='english', preprocessor=preprocess_text)
word_matrix = vectorizer.fit_transform(lines)
```

注意这里调用 `lower` 将文本转换为小写。如果提供了一个预处理函数，`CountVectorizer` 就不会自动将文本转换为小写，所以预处理函数必须自己转换。不过，它仍然会自动删除标点符号。

`CountVectorizer` 另一个有用的参数是 `min_df`，它可以忽略那些出现次数少于指定次数的词。参数值可以是指定了最小计数的一个整数（例如，忽略在训练文本中出现少于 5 次的词，即 `min_df = 5`）。也可以是一个 0.0~1.0 的浮点值，指定在所有样本中，必须包含一个词的样本的最小百分比（例如，忽略只有不到 10%样本才有的词，即 `min_df=0.1`）。这可以帮助我们过滤掉那些可能没有意义的词，而且它通过减少词汇表的大小来减少内存消耗和训练时间。`CountVectorizer` 还支持一个 `max_df` 参数，用于消除那些出现频率过高的词。

前面的例子使用的是 `CountVectorizer`，那么什么时候（以及为什么）要改为使用 `HashingVectorizer` 或 `TfidfVectorizer` 呢？`HashingVectorizer` 在处理大型数据集时非常有用。它不是在内存中存储词，而是对每个词进行哈希处理，并将哈希值作为一个词计数数组的索引。因此，它可以用更少的内存做更多的事情。另外，在对矢量化器（vectorizers）进行序列化时，它还有助于减少这些矢量化器的大小。序列化的目的是以后能够恢复它们——详情将在第 7 章讨论。`HashingVectorizer` 的缺点在于，它不能让你从反方向从矢量化的文本转换为原始文本。`CountVectorizer` 则可以，它为这个目的提供了一个 `inverse_transform` 方法（https://oreil.ly/jlJxb）。

`TfidfVectorizer` 经常用于进行关键词提取：检查一个或一组文档，并提取反映其内容特征的关键词。它为词分配反映其重要性的数值权重，并使用两个因素来确定权重：词在单个文档中出现的频率，以及词在整个文档集中出现的频率。如果一个词在单个文档中出现频率较高，但在较少的文档中出现，那么它会获得较高的权重。这里不再深入这一主题，但如果你想更多地了解，那么可以实验本书的 GitHub repo 包含的一个笔记本（https://oreil.ly/WNOiU），它使用 `TfidfVectorizer` 从第 1 章的英文版手稿中提取关键词。

4.2 情感分析

为了训练情感分析模型，我们需要一个带标签的数据集。公共领域有几个这样的数据集可用（https://oreil.ly/qYYGM）。其中一个是 IMDB 影评数据集（https://oreil.ly/INB8o），它包含 25000 条负面评价和 25000 条正面评价的样本，这些评价都发布在 Internet Movie Database 网站上（https://imdb.com）。每条评价都被细致地标记为 0 代表负面情感，1 代表正面情感。为了演示情感分析是如何工作的，让我们构建一个二分类模型，并用这个数据集来训练它。我们将使用逻辑回归作为学习算法。这个模型产生的情感分析分数只是输入表达了正面情感的概率，可以调用 `LogisticRegression` 的 `predict_proba` 方法来轻松获取这个概率。

首先下载数据集（https://oreil.ly/uex7A）并将其复制到你的 Jupyter 笔记本目录下的 Data 子目录。然后，在笔记本中运行以下代码，加载数据集并显示前 5 行：

```
import pandas as pd

df = pd.read_csv('Data/reviews.csv', encoding='ISO-8859-1')
df.head()
```

`encoding` 属性是必须的，因为 CSV 文件使用 ISO-8859-1 字符编码而不是 UTF-8。输出结果如下所示：

	Text	Sentiment
0	Once again Mr. Costner has dragged out a movie...	0
1	This is an example of why the majority of acti...	0
2	First of all I hate those moronic rappers, who...	0
3	Not even the Beatles could write songs everyon...	0
4	Brass pictures (movies is not a fitting word f...	0

了解数据集包含多少行，并确认没有缺失值：

```
df.info()
```

使用以下语句来查看每个类别有多少个实例（0 代表负面，1 代表正面）：

```
df.groupby('Sentiment').describe()
```

下面是输出：

	Text				
	count	unique	top	freq	
Sentiment					
0	25000	24697	When i got this movie free from my job, along ...	3	
1	25000	24884	Loved today's show!!! It was a variety and not...	5	

正面和负面样本数量持平，但是在每一种情况下，唯一样本数量都少于该类别的样本数量。这意味着该数据集有重复的行，而重复的行可能会使机器学习模型产生偏差。使用以下语句删除重复的行，并再次检查是否平衡：

```
df = df.drop_duplicates()
df.groupby('Sentiment').describe()
```

现在没有重复行了，正面和负面样本的数量也大致相等。

接着，使用 CountVectorizer 准备 Text 列中的文本并进行矢量化。将 min_df 设为 20，以忽略那些在训练文本中出现频率较低的词。这减少了发生“内存不足”错误的机率，并可能会使模型更准确。另外，我们还使用了 ngram_range 参数，允许 CountVectorizer 包括词对以及单个词。

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(ngram_range=(1, 2), stop_words='english',
                             min_df=20)

x = vectorizer.fit_transform(df['Text'])
y = df['Sentiment']
```

现在将数据集拆分为训练和测试两部分。我们将使用 50/50 分割，因为总共有近 50000 个样本。

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.5,
                                                    random_state=0)
```

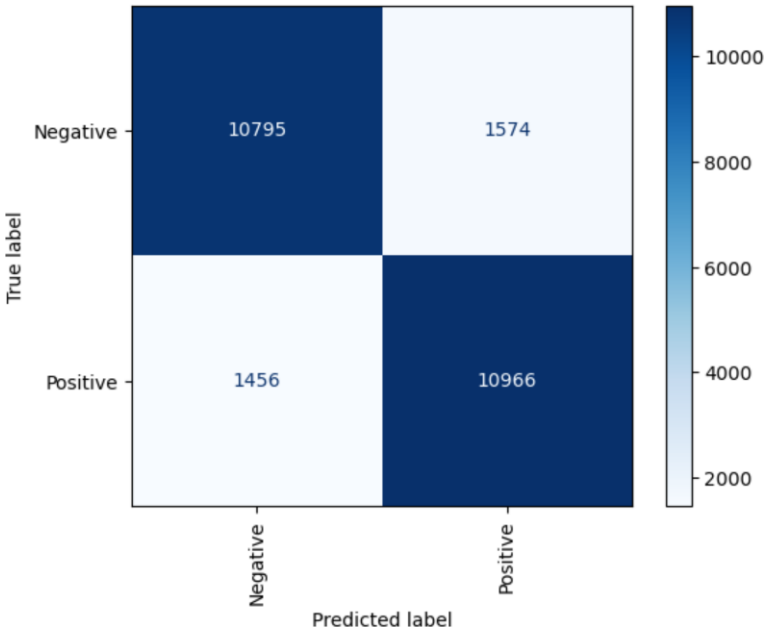
下一步是训练一个分类器。我们将使用 Scikit 的 LogisticRegression 类，它使用逻辑回归来为数据拟合一个模型。

```
from sklearn.linear_model import LogisticRegression

model = LogisticRegression(max_iter=1000, random_state=0)
model.fit(x_train, y_train)
```

用留作测试的 50%数据集来验证训练好的模型，并在一个混淆矩阵中显示结果。

```
%matplotlib inline
from sklearn.metrics import ConfusionMatrixDisplay as cmd

cmd.from_estimator(model, x_test, y_test,
                   display_labels=['Negative', 'Positive'],
                   cmap='Blues', xticks_rotation='vertical')
```

混淆矩阵显示，该模型正确识别了 10795 条负面评价，而错误地分类了其中 1574 条。它正确识别了 10966 条正面评价，错了 1456 次。



现在，有趣的地方到了：分析输入的一句话的情感。使用以下语句为“The long lines and poor customer service really turned me off”（大排长队，客户服务差劲）这句话生成一个情感分数。

```
text = 'The long lines and poor customer service really turned me off'
model.predict_proba(vectorizer.transform([text]))[0][1]
```

以下是输出结果：

```
0.09183447847778639
```

现在，对“The food was great and the service was excellent!”（食物很好，服务也不错！）做同样的事情。

```
text = 'The food was great and the service was excellent!'
model.predict_proba(vectorizer.transform([text]))[0][1]
```

如果你期望这句话能得到一个高分，那么结果不会令你失望：

```
0.8536277207125618
```

请自行尝试输入各种句子，看看自己是否同意该模型所预测的情感分数。它并不完美，但足够好。如果通过它运行数百条评价或意见，应该会得到文本所表达的情感的可靠指示。

有的时候，`CountVectorizer` 内置的停用词列表会*降低*模型的准确率，因为该列表过于宽泛。作为一个实验，可以从 `CountVectorizer` 中删除 `stop_words='english'`，然后再次运行代码。检查混淆矩阵。准确率是提高了还是下降了？也可以随意更改其他参数，例如 `min_df` 和 `ngram_range`。在现实世界中，数据科学家经常会尝试许多不同的参数组合，以确定哪一个能产生最佳结果。

4.3 朴素贝叶斯

逻辑回归是分类模型的一种常用算法，在对文本进行分类时往往非常有效。但是，在涉及文本分类的场景中，数据科学家经常转向另一种学习算法，即朴素贝叶斯（Naive Bayes）（https://oreil.ly/MqGwH）。这是一种基于贝叶斯定理（https://oreil.ly/dZxN3）的分类算法，它提供了一种计算条件概率的方法。在数学上，贝叶斯定理是这样表述的：

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)}$$

它的意思是说，在 B 为真的情况下，A 为真的概率等于在 A 为真的情况下 B 为真的概率乘以 A 为真的概率，结果除以 B 为真的概率。这句拗口的话虽然准确，但并不能说明为什么朴素贝叶斯对文本分类如此有用——或者应该如何应用。例如，如何把它应用于一个电子邮件集合，以判断哪些是垃圾邮件。

先从一个简单的例子开始。假设收到的所有电子邮件中有 10%是垃圾邮件。这个值就是 P(A)。分析发现，收到的垃圾邮件中有 5%含有“congratulations”（恭喜）一词，但所有邮件中只有 1%含有这个词。因此，P(B|A)为 0.05，P(B)为 0.01。如果一封电子邮件含有“congratulations”一词，那么它是垃圾邮件的概率为 P(A|B)，即(0.05×0.10) / 0.01 = 0.50。

当然，垃圾邮件过滤器必须考虑电子邮件中的所有词，而非只考虑一个。事实证明，如果做一些简单的（朴素的）假设——词在电子邮件中的顺序并不重要，而且每个词都有相同的权重——那么可以这样为垃圾邮件分类器写贝叶斯方程：

$$P(S \mid message) = P(S) \cdot P(word_1 \mid S) \cdot P(word_2 \mid S) ... P(word_n \mid S)$$

其中，*message* 是邮件，而 *word* 是其中的单词。通俗地说，上述方程表明，一封邮件是垃圾邮件的概率与以下各项的乘积成正比：

- 数据集中任何一封邮件都是垃圾邮件的概率，即 *P(S)*
- 邮件中的每个单词出现在垃圾邮件中的概率，即 *P(word|S)*

*P(S)*很容易计算，它就是垃圾邮件在数据集的所有邮件中的占比。如果用 1000 封邮件训练一个机器学习模型，其中 500 封是垃圾邮件，那么 *P(S)*=0.5。对于一个给定的单词，*P(word|S)*是该单词在垃圾邮件中出现的次数除以所有垃圾邮件中的单词数。这样，整个问题就简化为对单词进行计数。可以做一个类似的计算来计算一封邮件不是垃圾邮件的概率，然后用这两个概率中较高的那个来做预测。

下面是一个涉及 4 个电子邮件样本的例子。这些邮件是：

文本	垃圾邮件
Raise your credit score in minutes	1
Here are the minutes from yesterday's meeting	0
Meeting tomorrow to review yesterday's scores	0
Score tomorrow's meds at yesterday's prices	1

如果去掉停用词，将字符转换为小写，并对 tomorrow's 这样的词进行词干提取（变成 tomorrow），那么会得到以下结果：

文本	垃圾邮件
raise credit score minute	1
minute yesterday meeting	0
meeting tomorrow review yesterday score	0
score tomorrow med yesterday price	1

由于 4 封邮件中有两封是垃圾邮件，两封不是，所以任何一封邮件是垃圾邮件的概率（*P(S)*）为 0.5。任何一封邮件不是垃圾邮件的概率亦是如此（*P(N)*=0.5）。此外，垃圾邮件包含 9 个独特的词，而非垃圾邮件总共包含 8 个。

下一步是构建以下词频表。以“yesterday”（昨天）这个词为例。它在被标记为垃圾邮件的邮件中出现一次，所以 *P(yesterday|S)*为 1/9，即 0.111。它在非垃圾邮件中出现两次，所以 *P(yesterday|N)*为 2/8，即 0.250。

词	*P*(*word*|*S*)	*P*(*word*|*N*)
raise	1/9 = 0.111	0/8 = 0.000
credit	1/9 = 0.111	0/8 = 0.000
score	2/9 = 0.222	1/8 = 0.125
minute	1/9 = 0.111	1/8 = 0.125
yesterday	1/9 = 0.111	2/8 = 0.250
meeting	0/9 = 0.000	2/8 = 0.250
tomorrow	1/9 = 0.111	1/8 = 0.125
review	0/9 = 0.000	1/8 = 0.125
med	1/9 = 0.111	0/8 = 0.000
price	1/9 = 0.111	0/8 = 0.000

这在一定程度上已经可用了。但是，表格中的零值还是一个问题。例如，假定要判断“Scores must be reviewed by tomorrow”是否为垃圾邮件。我们去掉停用词，只剩下“score review tomorrow”。可以像下面这样计算它是垃圾邮件的概率：

$$P(S|score\ review\ tomorrow) = P(S) \cdot P(score|S) \cdot P(review|S) \cdot P(tomorrow|S)$$
$$P(S|score\ review\ tomorrow) = 0.5 \cdot 0.222 \cdot 0.0 \cdot 0.111 = 0$$
$$P(S|score\ review\ tomorrow) = 0$$

结果是 0，因为任何垃圾邮件都没有“review”一词，而 0 乘以任何东西都是 0。算法根本无法给“Scores must be reviewed by tomorrow”分配一个垃圾邮件概率。

为了解决这个问题，一个常见的方法是应用拉普拉斯平滑法（Laplace smoothing）（https://oreil.ly/iRt2y），也称为加法平滑法（additive smoothing）。通常，这涉及到在每个分子上加 1，在每个分母上加数据集中独特词的数量（本例为 10）。现在，*P(review|S)*求值为(0 + 1) / (9 + 10)，即 0.053。这不多，但总比没有好。下面是新的词频表，这次用拉普拉斯平滑法进行了修订。

词	*P(word|S)*	*P(word|N)*
raise	(1 + 1) / (9 + 10) = 0.105	(0 + 1) / (8 + 10) = 0.056
credit	(1 + 1) / (9 + 10) = 0.105	(0 + 1) / (8 + 10) = 0.056
score	(2 + 1) / (9 + 10) = 0.158	(1 + 1) / (8 + 10) = 0.111
minute	(1 + 1) / (9 + 10) = 0.105	(1 + 1) / (8 + 10) = 0.111
yesterday	(1 + 1) / (9 + 10) = 0.105	(2 + 1) / (8 + 10) = 0.167
meeting	(0 + 1) / (9 + 10) = 0.053	(2 + 1) / (8 + 10) = 0.167
tomorrow	(1 + 1) / (9 + 10) = 0.105	(1 + 1) / (8 + 10) = 0.111
review	(0 + 1) / (9 + 10) = 0.053	(1 + 1) / (8 + 10) = 0.111
med	(1 + 1) / (9 + 10) = 0.105	(0 + 1) / (8 + 10) = 0.056
price	(1 + 1) / (9 + 10) = 0.105	(0 + 1) / (8 + 10) = 0.056

现在，可以通过进行两个简单的计算来判断“Scores must be reviewed by tomorrow”是不是垃圾邮件。

$$P(S|score\ review\ tomorrow) = 0.5 \cdot 0.158 \cdot 0.053 \cdot 0.105 = 0.000440$$
$$P(N|score\ review\ tomorrow) = 0.5 \cdot 0.111 \cdot 0.111 \cdot 0.111 = 0.000684$$

根据结果，“Scores must be reviewed by tomorrow”很可能不是垃圾邮件。注意，这些概率是相对的，但可以将它们归一化（normalize），并得出结论，即根据用于训练模型的邮件，这封邮件有 40%的概率是垃圾邮件，60%的概率不是。

幸好，所有这些计算都不需要手动进行。Scikit-Learn 提供了几个类来帮助你，其中包括 `MultinomialNB` 类（https://oreil.ly/twFtY），它与由 `CountVectorizer` 生成的单词计数表配合得很好。

4.4 垃圾邮件过滤

现代垃圾邮件过滤器非常善于识别垃圾邮件，这并不是巧合。几乎所有过滤器都依赖于机器学习。这种模型很难用算法来实现，因为使用 credit 和 score 等关键词来判断一封邮件是否为垃圾邮件的算法很容易被愚弄。相比之下，机器学习检查是电子邮件的正文，并使用它学到的东西对新邮件进行分类。这样的模型往往能达到 99%以上的准确率。而且，随着时间的推移，它们会越来越聪明，因为用于训练它们的电子邮件越来越多。

在之前的例子中，我们使用逻辑回归来预测输入的文本表达的是正面还是负面的情感。它使用文本表达正面情感的*概率*作为情感分数。如果看到诸如“The long lines and poor customer service really turned me off”（大排长队，客户服务差劲）这样的表述，它的分数会接近 0.0。而对于诸如“The food was great and the service was excellent!”（食物很好，服务也不错!）这样的表述，它的分数会接近 1.0。现在，让我们构建一个二分类模型，将电子邮件分类为垃圾邮件或非垃圾邮件，并使用朴素贝叶斯将该模型与训练数据拟合。

在公共领域有几个垃圾邮件分类数据集。每个数据集都包含一个电子邮件的集合，其中的样本用 1 标记为垃圾邮件，用 0 标记为非垃圾邮件。我们将使用一个相对较小的数据集，其中包含 1000 个样本。首先，下载数据集（https://oreil.ly/hljvo）并把它复制到笔记本所在目录的 Data 子目录。然后，加载数据并显示前五行。

```
import pandas as pd

df = pd.read_csv('Data/ham-spam.csv')
df.head()
```

接着，检查数据集是否有重复的行。

```
df.groupby('IsSpam').describe()
```

该数据集含有一个重复的行。让我们删除它并检查是否平衡。

```
df = df.drop_duplicates()
df.groupby('IsSpam').describe()
```

数据集现在包含 499 个不是垃圾邮件的样本以及 500 个是垃圾邮件的样本。下一步是使用 `CountVectorizer` 对邮件进行矢量化。同样地，我们允许 `CountVectorizer` 考虑单词对以及单个词，并使用 Scikit 内置的英语停用词词典来去除停止词。

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(ngram_range=(1, 2), stop_words='english')
x = vectorizer.fit_transform(df['Text'])
y = df['IsSpam']
```

拆分数据集，将 80%用于训练，20%用于测试。

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,
                                                    random_state=0)
```

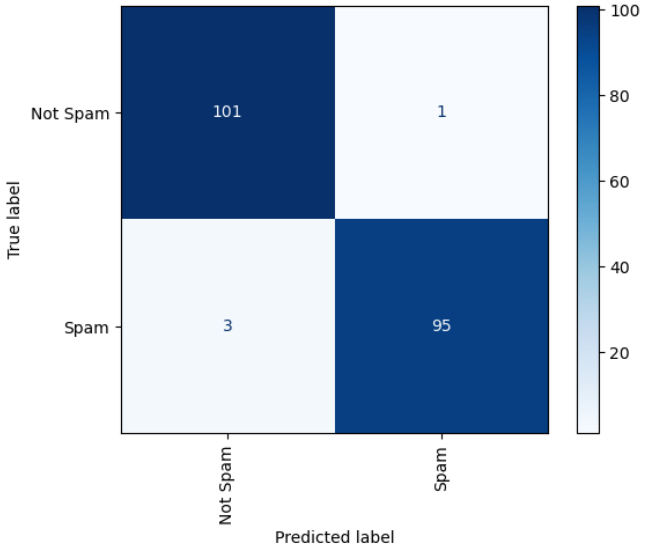
下一步是使用 Scikit 的 `MultinomialNB` 类（https://oreil.ly/0CtOh）来训练一个朴素贝叶斯分类器。

```
from sklearn.naive_bayes import MultinomialNB

model = MultinomialNB()
model.fit(x_train, y_train)
```

使用混淆矩阵，用预留的 20%数据集进行测试，验证训练好的模型。

```
%matplotlib inline
from sklearn.metrics import ConfusionMatrixDisplay as cmd

cmd.from_estimator(model, x_test, y_test,
                   display_labels=['Not Spam', 'Spam'],
                   cmap='Blues', xticks_rotation='vertical')
```

该模型将 102 封合法邮件中的 101 封正确识别为非垃圾邮件，将 98 封垃圾邮件中的 95 封正确识别为垃圾邮件。



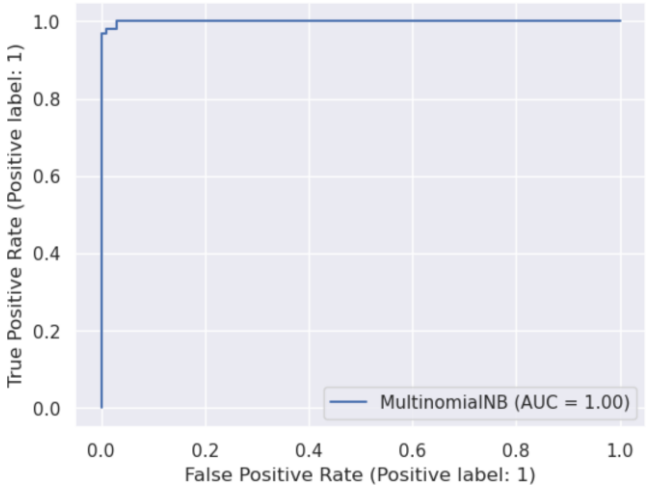
使用 `score` 方法来大致估计模型的准确率：

```
model.score(x_test, y_test)
```

然，使用 Scikit 的 `RocCurveDisplay` 类来可视化 ROC 曲线。

```
from sklearn.metrics import RocCurveDisplay as rcd
import seaborn as sns
sns.set()

rcd.from_estimator(model, x_test, y_test)
```

结果令人振奋。仅用 999 个样本进行训练，ROC 曲线下面积（AUC）表明该模型在将电子邮件分类为垃圾邮件或非垃圾邮件方面的准确率超过 99.9%。



现在，让我们看看该模型如何对它以前没有见过的几封邮件进行分类，首先是一封不是垃圾邮件的邮件。该模型的 predict 方法返回是邮件的类别：0 代表非垃圾邮件，1 代表垃圾邮件。

```
msg = 'Can you attend a code review on Tuesday to make sure the logic is solid?'
input = vectorizer.transform([msg])
model.predict(input)[0]
```

模型说这封邮件不是垃圾邮件，但它不是垃圾邮件的概率是多少？为此，我们可以调用 predict_proba，它返回包含两个值的一个数组。按照顺序，两个值分别是预测类别为 0 的概率，以及预测类别为 1 的概率。

```
model.predict_proba(input)[0][0]
```

模型似乎非常确定这封电子邮件是合法的：

```
0.9999497111473539
```

现在，用一封垃圾邮件来测试该模型。

```
msg = 'Why pay more for expensive meds when you can order them online ' \
      'and save $$$?'

input = vectorizer.transform([msg])
model.predict(input)[0]
```

该邮件不是垃圾邮件的概率是多少？

```
model.predict_proba(input)[0][0]
```

答案是：

```
0.00021423891260677753
```

该邮件是垃圾邮件的概率是多少？

```
model.predict_proba(input)[0][1]
```

答案是：

```
0.9997857610873945
```

注意，`predict` 和 `predict_proba` 接收的是一个输入列表。基于这个观察结果，你是否能执行一次方法调用，就对一整批邮件进行分类？如何获得每封邮件的结果？

4.5 推荐系统

近年来，机器学习衍变出的另一个分支是推荐系统（recommender systems）——向客户推荐产品或服务的系统。事实证明，它非常有用。据报道，Amazon 的推荐系统推动了其 35% 的销售额（https://oreil.ly/ue81Q）。一个好消息是，不需要成为 Amazon，也不需要拥有 Amazon 的资源，就能自己构建一个推荐系统。一旦掌握了几条基本原则，创建这种系统是相当容易的。

推荐系统有多种形式。其中，基于流行度的系统（popularity-based systems）根据当前流行的产品和服务向客户提供选择。例如，“这些是本周的畅销书”。协同系统（collaborative systems）则根据其他人的选择来做推荐。例如，“购买了这本书的顾客也同时购买了这些书”。这两种系统都不需要机器学习。

相反，还有一种“基于内容的系统”能极大地从机器学习中受益。这种系统的一个例子是：“如果你购买本书，你可能还会喜欢这些书”。这种系统需要对商品之间的相似性进行量化的一种方法。如果你喜欢电影“虎胆龙威”，那么可能喜欢、也可能不喜欢“巨蟒与圣杯”。相反，如果喜欢“玩具总动员”，那么也可能喜欢“虫虫危机”。但是，如何用算法来做出这种判断？

基于内容的推荐器需要两个要素：一种将构成服务或产品特质的属性进行矢量化（即转换为数字）的方法，以及一种对所生成的矢量之间的相似度进行计算的方法。第一个要素很容易。`CountVectorizer` 直接就能将文本转换为单词计数表。所以，为了构建一个推荐系统，你只需一种对单词计数行之间的相似性进行度量的方法。其中，最简单、最有效的一个方法是所谓的余弦相似（cosine similarity）技术。

4.5.1 余弦相似性

余弦相似性（cosine similarity）（https://oreil.ly/948eP）是一种数学手段，用于计算一对矢量（或被视为矢量的数字行）之间的相似性。其基本思路是，获取一个样本中的每个值（例如矢量文本行中的单词数），并将其作为一个矢量的端点坐标，另一个端点则位于坐标系的原点。对两个样本这样做，然后在 *m* 维空间中计算矢量之间的余弦。其中，*m* 是值在每个样本中的数量。由于 0 的余弦是 1，所以两个完全一样的矢量的相似度为 1。矢量越不相似，余弦就越接近于 0。

下面是一个二维空间中的例子。假设有三行数据，每行包含两个值。

1	2
2	3
3	1

我们想判断与第 2 行更相似的是第 1 行还是第 3 行。仅凭数字很难判断，而且在现实生活中，会有多得多的数字。如果简单地将每一行的数字相加，然后比较这些和，那么得出的结论会是第 2 行与第 3 行更相似。但是，如果像图 4-2 那样将每一行视为一个矢量呢？

- 第 1 行：(0, 0) → (1, 2)
- 第 2 行：(0, 0) → (2, 3)
- 第 3 行：(0, 0) → (3, 1)

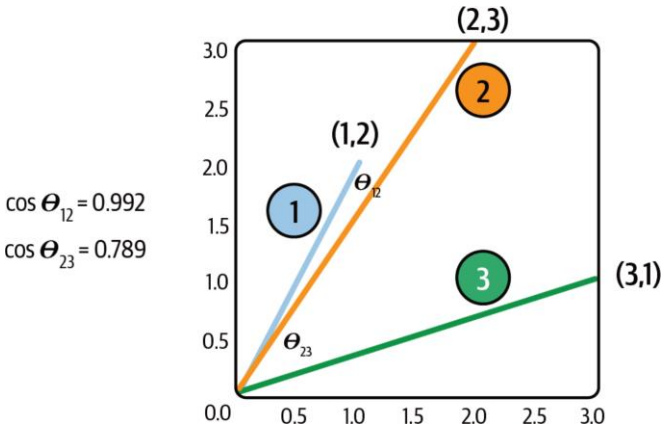


图 4-2 余弦相似性

然后，可以将每一行作为一个矢量来绘制，计算 1 和 2 以及 2 和 3 所形成的角度的余弦。这样会得出一个不同的结论：第 2 行更像第 1 行而不是第 3 行。这就是余弦相似性的简单原理。

余弦相似性并不局限于两个维度，它在更高维度的空间也能发挥作用。为了帮助计算余弦相似性（无论多少维），Scikit 提供了 cosine_similarity 函数（https://oreil.ly/vc1Uv）。以下代码计算了上例中三个样本的余弦相似度：

```
data = [[1, 2], [2, 3], [3, 1]]
cosine_similarity(data)
```

返回值是一个相似性矩阵（similarity matrix），其中包含了每个矢量对的余弦值。矩阵的宽度和高度等于样本的数量。

```
array([[1.        , 0.99227788, 0.70710678],
       [0.99227788, 1.        , 0.78935222],
       [0.70710678, 0.78935222, 1.        ]])
```

从中可以看出，第 1 行和第 2 行的相似度为 0.992，而第 2 行和第 3 行的相似度为 0.789。换言之，第 2 行与第 1 行的相似度比第 3 行高。第 2 行和第 3 行之间的相似度（0.789）也比第 1 行和第 3 行之间的相似度（0.707）高。

4.5.2 构建一个电影推荐系统

现在，让我们利用余弦相似度来构建一个基于内容的电影推荐系统。首先下载数据集（https://oreil.ly/ydE3v），它是 Kaggle.com 提供的几个电影数据集之一。这个数据集有大约 4800 部电影的信息，包括标题（title）、预算（budget）、类型（genres）、关键字（keywords）、演员（cast）等等。将 CSV 文件放在 Jupyter 笔记本所在目录的 Data 子目录下，然后加载数据集并浏览其内容。

```
import pandas as pd

df = pd.read_csv('Data/movies.csv')
df.head()
```

该数据集包含 24 列，其中只有几个是需要用来描述电影的。使用以下语句提取关键列，例如 title 和 genres，并用空字符串填充缺失的值：

```
df = df[['title', 'genres', 'keywords', 'cast', 'director']]
df = df.fillna('') # 用空字符串填充缺失值
df.head()
```

接着，添加一个名为 features（特征）的列，将其他列中的所有单词合并起来：

```
df['features'] = df['title'] + ' ' + df['genres'] + ' ' + \
            df['keywords'] + ' ' + df['cast'] + ' ' + \
            df['director']
```

使用 CountVectorizer 对 features 列中的文本进行矢量化：

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(stop_words='english', min_df=20)
word_matrix = vectorizer.fit_transform(df['features'])
word_matrix.shape
```

单词计数表包含 4803 行（每部电影一行）以及 918 列。下一个任务是计算每个行对（row pair）的余弦相似度：

```
from sklearn.metrics.pairwise import cosine_similarity

sim = cosine_similarity(word_matrix)
```

这个系统的终极目标是输入一个电影名称，并确定与该电影最相似的 *n* 部电影。为此，定义一个名为 `get_recommendations` 的函数，它接受一个电影标题、一个包含所有电影信息的 `DataFrame`、一个相似度矩阵以及要返回的电影标题的数量。

```
def get_recommendations(title, df, sim, count=10):
    # 获取DataFrame中指定电影标题（名称）的行索引
    index = df.index[df['title'].str.lower() == title.lower()]

    # 如果没有找到和指定标题对应的条目，就返回一个空列表
    if (len(index) == 0):
        return []

    # 获取相似度矩阵中的相应行
    similarities = list(enumerate(sim[index[0]]))

    # 将该行中的相似度分数按降序排列
    recommendations = sorted(similarities, key=lambda x: x[1], reverse=True)

    # 获取前n个推荐，忽略列表中的第一个条目，
    # 因为它对应标题本身（所以相似度为1.0）
    top_recs = recommendations[1:count + 1]

    # 从top_recs中的索引生成一个电影标题列表
    titles = []

    for i in range(len(top_recs)):
        title = df.iloc[top_recs[i][0]]['title']
        titles.append(title)

    return titles
```

该函数对余弦相似度进行降序排序，以确定与 `title` 参数所指定的那部电影相似度最高的 `count` 部电影。然后，返回这些电影的标题。

现在，使用 `get_recommendations` 来搜索数据库中的相似电影。首先，询问与 007 系列“Skyfall”（大破天幕杀机）最相似的 10 部电影：

```
get_recommendations('Skyfall', df, sim)
```

输出结果如下所示：

```
['Spectre',
 'Quantum of Solace',
 'Johnny English Reborn',
 'Clash of the Titans',
 'Die Another Day',
 'Diamonds Are Forever',
 'Wrath of the Titans',
 'I Spy',
 'Sanctum',
 'Blackthorn']
```

再次调用 `get_recommendations` 来列出和“Mulan”（花木兰）相似的电影：

```
get_recommendations('Mulan', df, sim)
```

请自行尝试其他电影。注意，只能输入数据集中已有的电影标题（名称）。使用以下语句来打印一份完整的电影清单：

```
pd.set_option('display.max_rows', None)
print(df['title'])
```

相信你也会同意，这个系统在推荐相似电影时相当靠谱。对于大约 20 行的代码来说，这已经很不错了！

4.6 小结

对文本进行分类的机器学习模型很常见，在行业和日常生活中都有多种用途。例如，有哪个理性的人不希望有一根魔法棒来清除所有垃圾邮件？

用来训练文本分类模型的文本必须在训练前进行准备并矢量化。准备工作包括转换成小写，去除标点符号。另外，可能还要去除停用词，去除数字，以及进行词干提取或词形还原。准备好的文本通过转换为一个词频表来进行矢量化。Scikit 的 `CountVectorizer` 类可以快速完成矢量化过程，并帮你完成一些准备工作。

一旦将文本转换为数值形式，就可以用逻辑回归和其他流行的分类算法对其进行分类。然而，对于文本分类任务，朴素贝叶斯学习算法经常要胜过其他算法。通过做出一些“朴素”（naive）的假设，例如单词在文本样本中出现的顺序不重要，朴素贝叶斯被简化为一个对单词进行计数的过程。Scikit 的 `MultinomialNB` 类提供了一个好用的朴素贝叶斯实现。

余弦相似性是一种计算两行数字之间相似性的数学方法。它的一个用途是构建推荐系统，根据客户购买过的其他产品或服务来推荐新的产品或服务。我们将 `CountVectorizer` 从文本描述生成的词频表与余弦相似性结合，从而创建一个智能推荐系统，使公司的利润最大化。

请随意使用本章的例子作为自己做实验的起点。例如，看看是否能在任何一个例子中调整传递给 `CountVectorizer` 的参数，并提高所生成的模型的准确率。数据科学家将寻找最佳

参数组合的过程称为超参数调优（hyperparameter tuning），这是下一章要讨论的主题。

第 5 章 支持向量机

支持向量机（Support Vector Machine，SVM）代表了机器学习的最前沿。它们最常用于解决分类问题，但也可用于回归。由于以独特的方式将数学模型和数据拟合，所以 SVM 经常在其他模型无能为力的情况下成功地找到类别之间的分离。从技术上讲，它们只能进行二分类，但 Scikit-Learn 使它们能使用第 3 章中介绍的技术进行多分类。

Scikit-Learn 通过针对分类模型的 `SVC`（Support Vector Classifier，支持向量分类器）（https://oreil.ly/IAiWs）和针对回归模型的 `SVR`（Support Vector Regressor，支持向量回归器）（https://oreil.ly/f8B8K）等类来简化支持向量机的构建。即使不了解 SVM 的工作原理，也能使用这些类。但是，如果真正了解它们的工作原理，那么会更大地挖掘它们的潜力。同样重要的是，应该了解如何为单独的数据集进行 SVM 调优，以及如何在训练模型前准备数据。在本章的最后，我们将构建一个能进行面部识别的 SVM。但首先让我们深入幕后，理解为什么说 SVM 通常是对真实世界数据集进行建模的首选机制。

5.1 支持向量机的工作原理

首先，为什么把它们称为支持向量机（SVM）？SVM 分类器的目的与其他任何分类器相同，即找到一个决策边界，干净地分离出各个类别。SVM 通过在二维空间找到一条线，在三维空间找到一个平面，或者在更高维的空间找到一个超平面（hyperplane）来实现这一目的，从而以最大的确定性区分不同的类别。在图 5-1 的例子中，我们可以画无数条直线来区分两个类，但最好的直线是产生最宽边缘的那条（如右图所示）。边缘宽度（margin width）是指沿着垂直于边界的一条直线，每个类别中最靠近边界的两个点之间的距离。这些点就是所谓的支持向量（support vectors），如红圈所示。

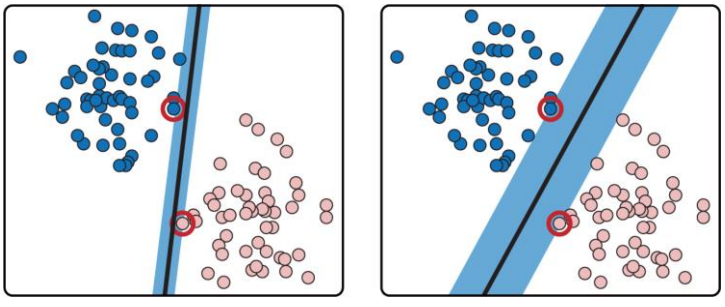


图 5-1 最大边缘分类法

当然，真实的数据很少能如此干净地分离。类别与类别之间的重叠不可避免地阻碍了完美的拟合。为了顺应这种情况，SVM 支持一个通常称为 `C` 的正则化参数，可以通过调整它来放松或收紧拟合。如图 5-2 所示，较低的 `C` 值会产生一个更宽的边缘，在决策边界的两侧

有更多的误差。较高的 C 值则会产生与训练数据更紧密的拟合；相应地，边缘会更窄，误差也更少。但是，如果 C 值过高，那么模型可能不能很好地泛化或者说归纳（generalize）。最佳值因数据集而异。数据科学家通常会尝试不同的 C 值，以确定哪一个对测试数据表现最好。

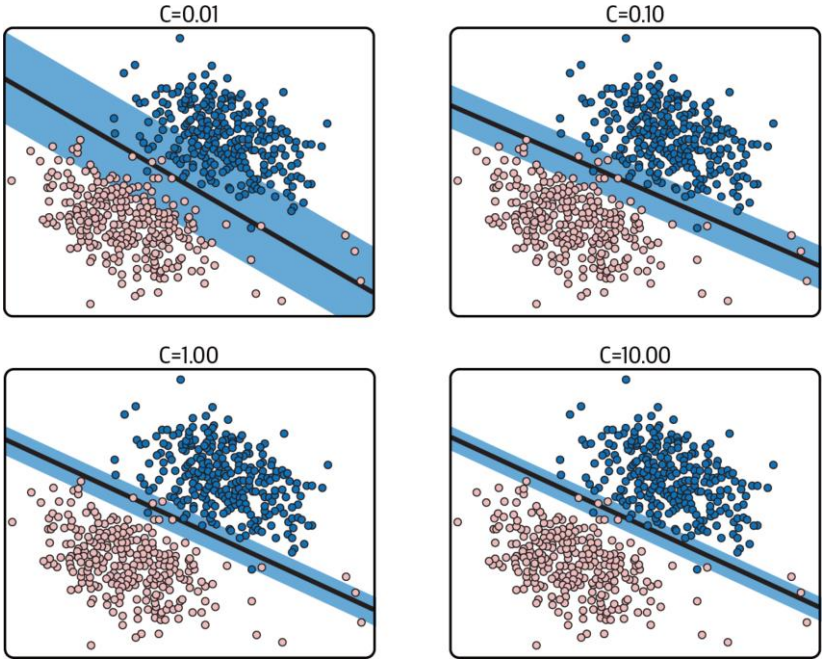


图 5-2 C 值对边缘宽度的影响

好了，虽然前面讲述的事实都是成立的，但还是没有解释为什么 SVM 如此出色。SVM 并不是唯一以数学手段寻找不同类别的分离边界的模型。SVM 的特殊之处在于所谓的“核”（kernels），其中一些核能为数据增加维度，以寻找低维度上不存在的边界。图 5-3 展示了一个例子。在本例中，不可能画一条线将红点和紫点完全分开。但是，如果添加了如右图所示的第三个维度——*z* 维度，其值基于一个点到中心的距离——那么就可以在紫点和红点之间滑动一个平面，从而实现 100%的分离。在本例中，在二维中不可线性分离的数据在三维中是可以线性分离的。在这里起作用的是 Cover 定理（https://oreil.ly/BAsz2）。该定理指出，如果使用非线性变换投影到高维空间，那么不可线性分离的数据也许能够线性分离。

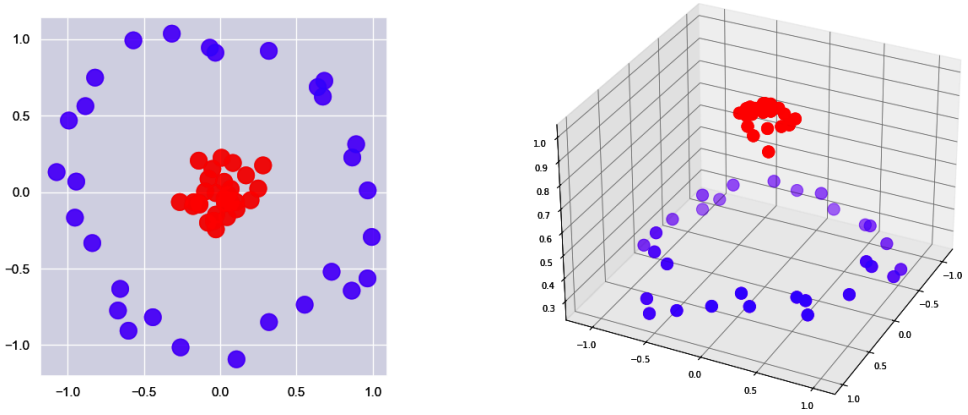


图 5-3 增加维度以实现线性分离

本例使用一个特定的核变换（kernel transformation），即在每个 *x* 和 *y* 上增加一个 *z*，从而将二维数据投影到三维空间。这种变换对于这个特定的数据集来说是有效的。但是，为了使 SVM 更通用，我们需要一个与特定数据集的形状无关的核。

5.1.1 核

Scikit-Learn 内置了几个通用的核，包括线性核、RBF 核[①]、多项式核和 sigmoid（S 型）核。线性核不增加维度，它在处理可直接线性分离的数据时效果不错，但在处理不可线性分离的数据时就很差劲。将其应用于图 5-3 所描述的问题，将生成如图 5-4 左侧所示的决策边界。将 RBF 核应用于相同的数据则生成了右边的决策边界。RBF 核将 *x* 和 *y* 值投影到一个高维空间，并找到一个超平面，将紫点和红点干净地分开。当投影回二维空间时，决策边界大致形成一个圆。在这个数据集上，使用适当调优的多项式核也能获得类似的结果，但一般来说，RBF 核可以在非线性数据中找到多项式核无法找到的决策边界。正是由于这个原因，所以 RBF 是 Scikit 的默认核类型。

一个合理的问题是，RBF 核是否有在每个 *x* 和 *y* 上加了一个 *z*？简单的回答是没有。它高效地将数据点投影到一个具有*无限*多维度的空间。注意，这里的重点是“高效”。核使用称为核技巧（kernel tricks）的数学手段来衡量增加新维度的效果，同时不实际计算它们的值。这就是 SVM 的数学显得很“高大上”的地方。核被精心设计为计算 *m* 维空间中的两个 *n* 维向量之间的点积（*m* 大于 *n*，甚至可以无限大）（https://oreil.ly/yGUYS），同时并不真的生成所有这些新维度。最终，点积成为 SVM 计算决策边界所需要的全部。这在数学上相当于既得到了鱼，也得到了熊掌。这正是使 SVM 显得如此强大的秘密。SVM 在大数

[①] RBF 是“径向基函数”（Radial Basis Function）的简称，详情请参见 https://oreil.ly/IRswE。

据集上的训练可能需要很长时间，但它的一个好处在于，在行数或样本数较少的小数据集上往往能比其他学习算法做得更好。

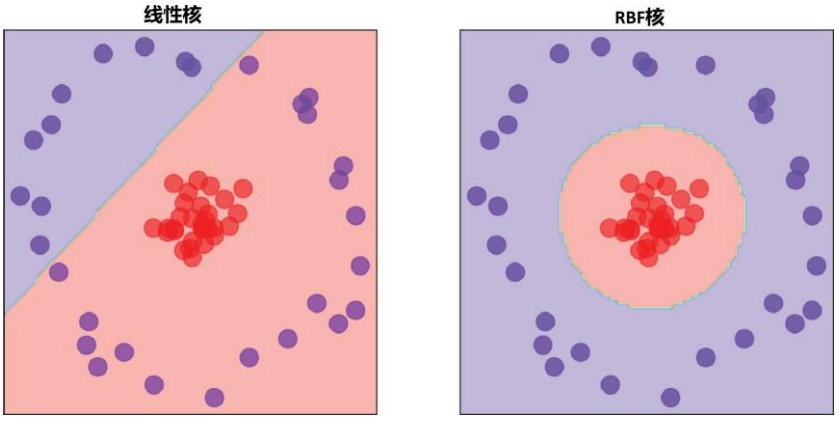


图 5-4　对比线性核与 RBF 核

5.1.2 核技巧

想看看如何利用核技巧来计算高维空间的点积，同时不为新维度实际计算值的例子吗？本节的内容可以略过。但是，如果你像我一样，觉得从具体的例子中学习效果更好，那么可能会发现本节的内容非常有帮助。

先从前面介绍的二维圆形数据集开始，但这次让我们用以下方程式把它投影到三维空间。

$$x' = x^2$$
$$y' = y^2$$
$$z = x \cdot y \cdot \sqrt{2}$$

换言之，我们将通过在二维空间中对 x 和 y 进行平方来计算三维空间中的 x 和 y（x′ 和 y′ ），并加上一个 z，即原始 x 和 y 与 2 的平方根的乘积。以这种方式投影数据，会在紫点和红点之间产生一个清晰的分离，如图 5-5 所示。

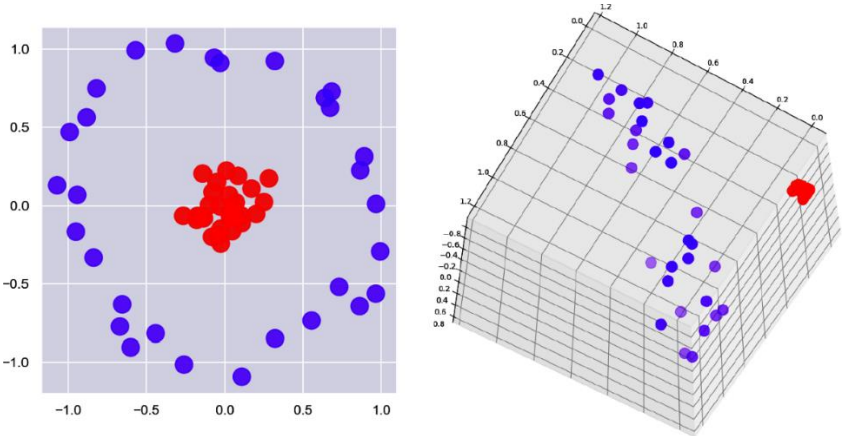


图 5-5 将二维点投影到三维，以分离两个类别

SVM 的效率来自它能计算高维空间中两个向量（或点，可将它们视为向量）的点积，同时不必将它们投影到该空间——也就是说，只使用原始空间的值。让我们创建两个点来做实验：

$a = (3, 7)$
$b = (2, 4)$

两个点的点积是这样计算的：

$$(3 \cdot 2) + (7 \cdot 4) = 34$$

当然，二维点积并不是很有帮助。一个 SVM 需要这些点在三维空间中的点积。让我们用前面的方程式将 *a* 和 *b* 投影到三维空间，然后计算结果的点积：

$$a = \left(3^2, 7^2, 3 \cdot 7 \cdot \sqrt{2}\right) = (9, 49, 29.6984)$$
$$b = \left(2^2, 4^2, 2 \cdot 4 \cdot \sqrt{2}\right) = (4, 16, 11.3137)$$
$$(9 \cdot 4) + (49 \cdot 16) + (29.6984 \cdot 11.3137) = 1{,}156$$

现在，我们有了一对二维点在三维空间的点积，但还必须在三维空间生成坐标才能得到它。这正是最有趣的地方。以下函数，或者称为核技巧，只用原始二维空间的值就能产生同样的结果：

$$K(a, b) = \langle a, b \rangle^2$$

$\langle a, b \rangle$是 *a* 和 *b* 的点积，所以$\langle a, b \rangle^2$ 是 *a* 和 *b* 的点积的平方。由于我们已经知道如何计算 *a* 和 *b* 的点积，所以：

$$K((3, 7), (2, 4)) = ((3 \cdot 2) + (7 \cdot 4))^2 = 34^2 = 1{,}156$$

这和对点进行显式投影所计算出来的结果一致，但又不需要真的进行投影。这就是核技巧的常识。从二维到三维时，它节省了时间和内存。试想一下，在投影到无限多的维度时（前面说过，这正是 RBF 核所做的事情），能够节省多少时间！

这里使用的核技巧并不是凭空捏造的。它恰好是一个 2 次（degree-2）多项式核所使用的。在 Scikit 中，可以用 2 次多项式核为一个数据集拟合 SVM 分类器，如下所示：

```
model = SVC(kernel='poly', degree=2)
model.fit(x, y)
```

把它应用于前面描述的圆形数据集并绘制决策边界（图 5-6 右侧），其结果几乎与 RBF 核生成的结果相同。有趣的是，一个 1 次多项式核（图 5-6 左侧）生成的决策边界与线性核相同，因为直线就是一个 1 次多项式。

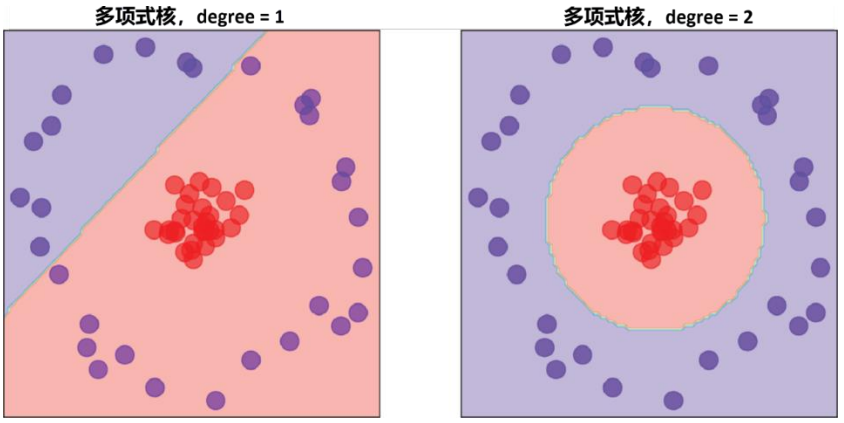


图 5-6 对比 1 次与 2 次多项式核

核技巧很特殊。每一个都是为了模拟向高维的一个特定投影而设计的。Scikit 提供了几个核，但还有一些核是 Scikit 没有内置的。可以用自己的核来扩展 Scikit，但它提供的那几个核在绝大多数情况下都够用了。

5.2 超参数调整

一开始的时候，很难知道哪个内置核能生成最准确的模型。也很难知道最恰当的 `C` 值是多少——这个值应该在训练数据的欠拟合和过拟合之间提供一个最佳的平衡，而且在用测试数据运行模型时能产生最佳结果。对于 RBF 和多项式核，还有第三个称为 `gamma` 的值会影响准确率。另外，对于多项式核，`degree` 参数会影响模型从训练数据中学习的能力。

`C` 参数控制模型与训练数据的拟合度。该值越高，拟合越紧密，过拟合的风险越大。图 5-7 展示了 RBF 核是如何在不同 `C` 值下对一组包含三个类别的训练数据进行拟合的。在 Scikit 中，默认 `C = 1`。但是，完全可以指定一个不同的值来调整拟合。可以在右下方的图中看到过拟合的危险。一个位于最右边的点被归类为蓝色，尽管它可能属于黄色或棕色类别。欠拟合也是一个问题。在左上方的图中，几乎所有非棕色的数据点都会被归为蓝色。

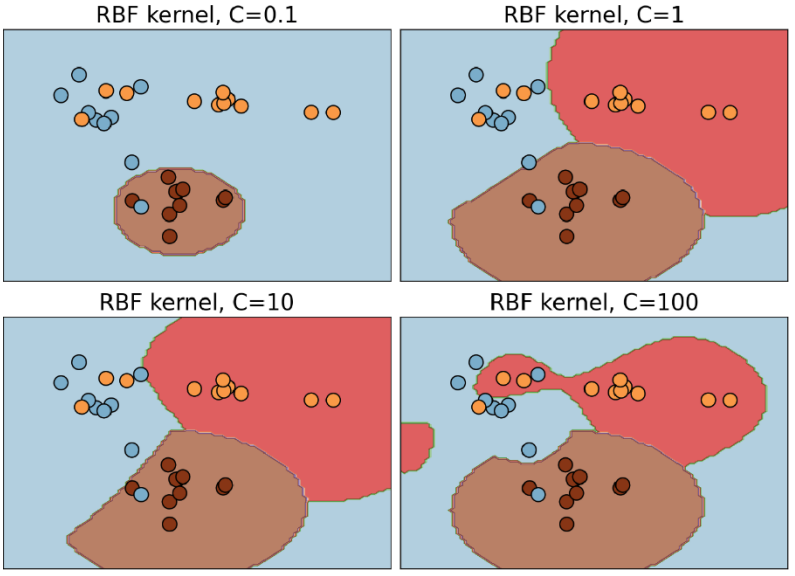


图 5-7 C 值对 RBF 核的影响

除了 C 参数，使用 RBF 核的 SVM 要等到 gamma 也有了一个合适的值后，才能开始调优。gamma 控制单个数据点在计算决策边界时的影响范围。较低的值会使用更多的点，并产生更平滑的决策边界；较高的值涉及更少的点，并和训练数据更紧密地拟合。图 5-8 对此进行了演示，在保持 C 不变的情况下，增大 gamma 会使决策边界更紧密地包围各个类别聚类（clusters of classes）。gamma 可为任意非零正值，但 0 到 1 之间的值最常见。如果不手动指定 gamma 的值，Scikit 会通过算法挑选一个默认值，而不是硬编码一个默认值。

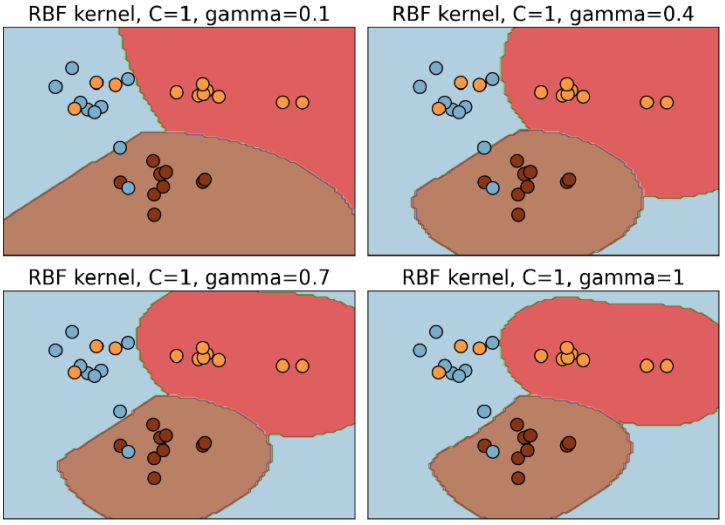


图 5-8 gamma 对 RBF 核的影响

在实践中，数据科学家用不同的核和不同的参数值进行试验，以找到生成最准确模型的组合，这个过程称为超参数调优（hyperparameter tuning）。超参数调优并不是只对 SVM 有用，但几乎总是可以通过找到核类型、C 和 gamma 的最佳组合（对于多项式核，还有 degree）来使 SVM 更准确。

为了辅助进行超参数调优，Scikit 提供了一个优化器系列（https://oreil.ly/IFpOA），其中包括 GridSearchCV（https://oreil.ly/n32OC），它尝试一组指定参数值的所有组合，并内置交叉验证，以确定哪种组合能生成最准确的模型。这些优化器使你不必自己写代码来使用所有独特的参数值组合进行蛮力查找。事实上，它们是自己多次训练模型，每次都用不同的值组合来进行蛮力查找。最后，可以从 best_estimator_属性获取最准确的模型，从 best_params_属性获取生成最准确模型的参数值，并从 best_score_属性中获取最佳分数。

下面是一个使用 Scikit 的 SVC 类来实现 SVM 分类器的例子。对于初学者来说，可以创建使用了默认参数值的一个 SVM 分类器，并通过以下两行代码将其和数据集拟合：

```
model = SVC()
model.fit(x, y)
```

上述代码使用 C = 1 的 RBF 核。可以像下面这样指定核的类型以及 C 和 gamma 的值：

```
model = SVC(kernel='poly', C=10, gamma=0.1)
model.fit(x, y)
```

假设要尝试 2 个不同的核以及 5 个 C 和 gamma 值，看看哪个组合能产生最好的结果，那么不需要写一个嵌套 for 循环。相反，可以这样做：

```
model = SVC()

grid = {
    'C': [0.01, 0.1, 1, 10, 100],
    'gamma': [0.01, 0.25, 0.5, 0.75, 1.0],
    'kernel': ['rbf', 'poly']
}

grid_search = GridSearchCV(estimator=model, param_grid=grid, cv=5, verbose=2)
grid_search.fit(x, y) # 用参数值的不同组合来训练模型
```

对 fit 的调用要过一段时间才会返回。它对模型进行了 250 次训练，因为 kernel、C 和 gamma 共有 50 种不同的组合，而 cv=5 指定用 5 折交叉验证（2.6 节）来评估结果。一旦训练完成，就可以像下面这样获取最佳模型：

```
best_model = grid_search.best_estimator_
```

像上面这样进行多次查证并不罕见——第一次使用常规参数值，以后每次都基于从 best_params_获得的值进行小范围变化。前期更多的训练时间是为最终准确的模型而必须付出的代价。重申一下，几乎总是可以通过找到参数的最佳组合来使 SVM 更加准确。而

不管怎样，蛮力法（brute force）都是识别最佳组合的最有效的方法。

SVC 类有一个特别的地方需要留意，即它默认不会计算概率。因此，如果想在一个 SVC 实例上调用 predict_proba，那么必须在创建实例时将 probability 设为 True，如下所示：

```
model = SVC(probability=True)
```

模型的训练速度会比较慢，但最后除了能获取预测结果，还能同时获取概率。另外，Scikit 文档警告说：“predict_proba 的结果可能与 predict 不一致”。欲知详情，请参见文档的 1.4.1.2 节（https://oreil.ly/Jg8X0）。

5.3 数据归一化

第 2 章指出，一些学习算法在归一化的数据上能更好地工作。非归一化的数据包含了取值范围有很大区别的数值列——例如，一列中的值在 0~1 之间，而另一列中的值在 0~1000000 之间。SVM 是一种参数化学习算法。它用归一化数据进行训练很重要，因为 SVM 使用距离来计算边界（margin）。如果一个维度占的边界宽度比另一个维度大*得多*，那么用于寻找最大边界的内部算法可能很难收敛于一个解。

用归一化数据训练机器学习模型的重要性并不限于 SVM。决策树和学习算法（例如依赖于决策树的随机森林和梯度提升决策树）都是非参数化的。所以，无论归一化还是非归一化的数据，它们的表现一致。但是，它们只是一个例外。其他大多数学习算法都要在某种程度上受益于数据的归一化。其中包括 k-近邻，尽管它是非参数化的，但它在内部使用基于距离的计算来区分不同的类别。

Scikit 提供了几个用于对数据进行归一化的类。其中最常用的是 MinMaxScaler（https://oreil.ly/nz2wR）和 StandardScaler（https://oreil.ly/OTTrm）。前者通过将每一列的值按比例缩减为 0.0~1.0 的值来归一化数据。这在数学上是很简单的一件事情。对于数据集中的每一列，MinMaxScaler 从该列的所有值中减去该列的最小值，然后用每个值除以最小值和最大值之差。这样一来，在结果列中，最小值为 0.0，而最大值为 1.0。

为了演示，我从 Scikit 内置的乳腺癌数据集（https://oreil.ly/IQAC9）中提取了两列取值范围有很大区别的子集。每一列都包含 100 个值。下面是前 10 行：

```
[[1.001e+03 3.001e-01]
 [1.326e+03 8.690e-02]
 [1.203e+03 1.974e-01]
 [3.861e+02 2.414e-01]
 [1.297e+03 1.980e-01]
 [4.771e+02 1.578e-01]
 [1.040e+03 1.127e-01]
 [5.779e+02 9.366e-02]
 [5.198e+02 1.859e-01]
 [4.759e+02 2.273e-01]]
```

第一列的取值范围为 201.9~1878.0；第二列则为 0.000692~0.3754。图 5-9 展示了用 x 轴和 y 轴等比例绘制数据的情况。由于第一列的值比第二列的值大得多，所以数据点看起来形成了一条直线。如果调整坐标轴的比例，使之与每一列的取值范围相匹配，那么会得到一个全然不同的结果，如图 5-10 所示。

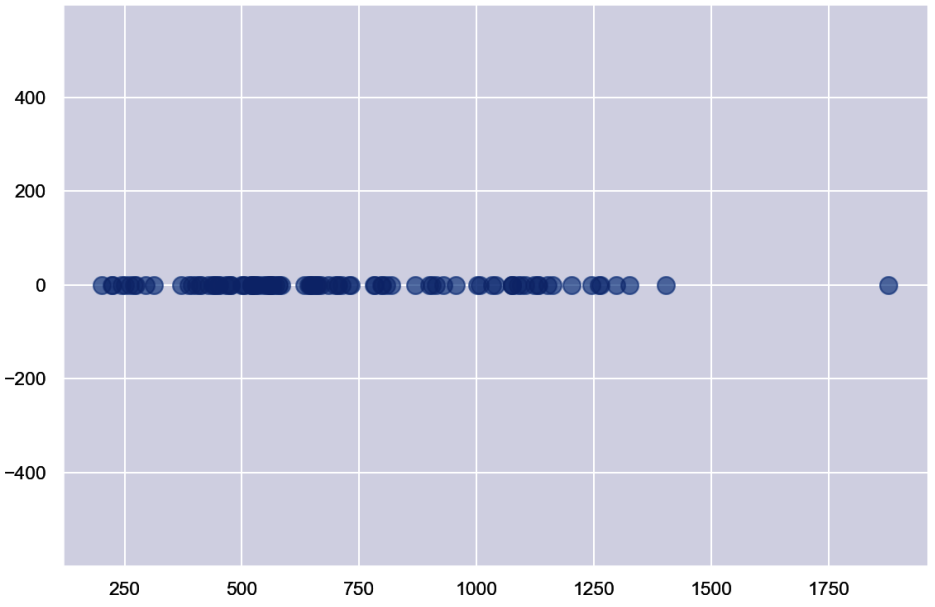


图 5-9 用等比例的轴绘制的非归一化的数据

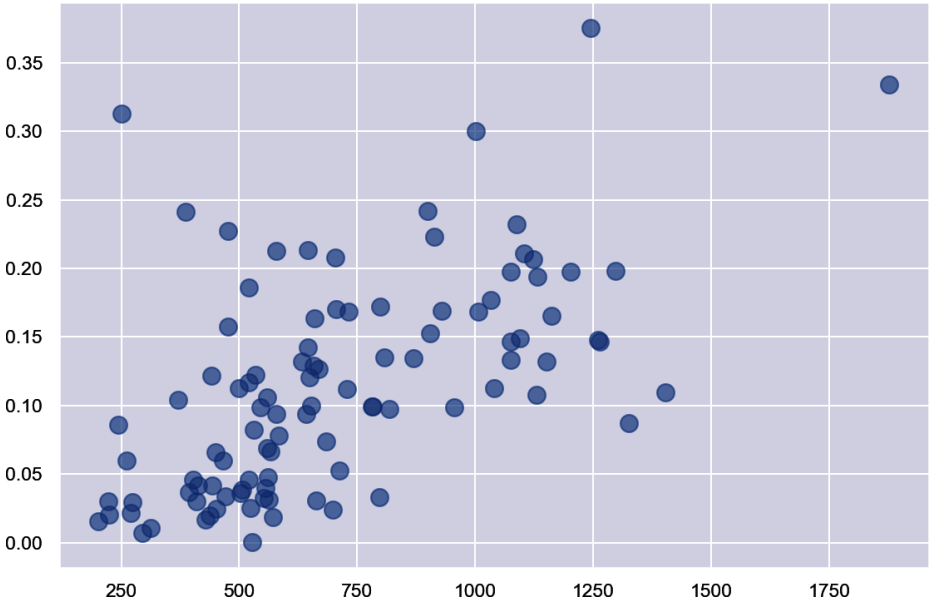


图 5-10 用成比例的轴绘制的非归一化的数据

这种非归一化数据会给参数化学习算法带来问题。解决这个问题的一个办法是对数据应用 MinMaxScaler，如下所示：

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
normalized_data = scaler.fit_transform(data)
```

下面是在进行了最小-最大归一化（MinMaxScaler）后的前 10 行：

```
[[0.47676153 0.79904352]
 [0.67066404 0.23006715]
 [0.5972794  0.52496344]
 [0.10989798 0.64238821]
 [0.65336197 0.52656469]
 [0.16419068 0.41928115]
 [0.50002983 0.29892076]
 [0.22433029 0.24810786]
 [0.18966649 0.49427287]
 [0.16347473 0.60475891]]
```

图 5-11 显示的是归一化后数据的等轴图。数据的形状并未改变。发生改变的是，两列现在都包含 0.0~1.0 的值。

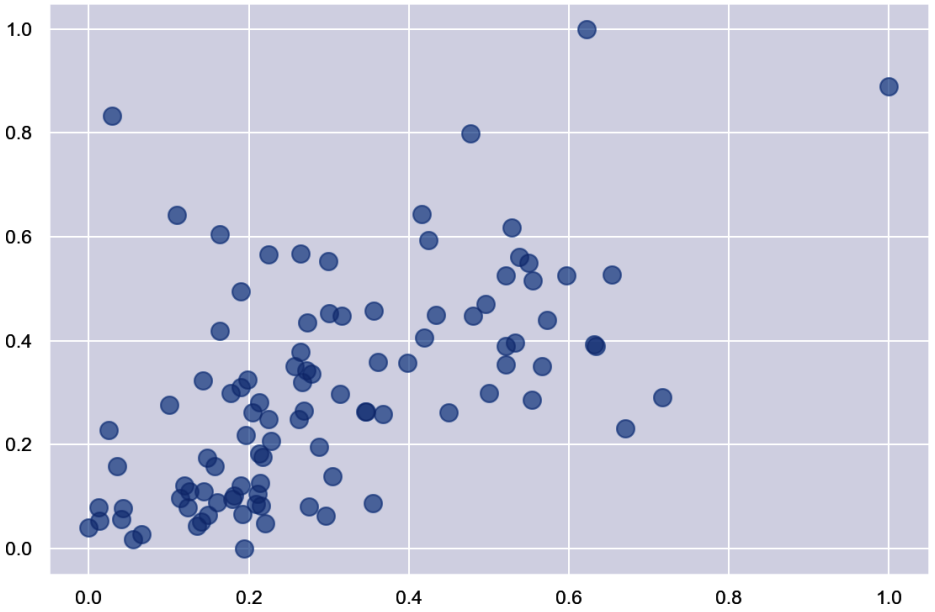


图 5-11 用 MinMaxScaler 归一化的数据

SVM 几乎总是能用归一化的数据进行更好的训练。但是，MinMaxScaler 执行的简单归一化有时还不够。SVM 倾向于使用一种叫做标准化（standardization）（https://oreil.ly/UrsYP）或者 Z-score（Z 分数）归一化的技术，它将数据归一化为单位方差（unit variance），从而获得更好的响应。单位方差是通过对数据集中的每一列执行以下操作来获得的：

- 计算该列中所有值的平均值和标准差
- 从该列的每个值中减去平均值
- 用该列中的每个值除以标准差

这正是 Scikit 的 StandardScaler 类对数据集执行的转换。要将单位方差应用于数据集，执行以下寥寥几行代码即可：

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
normalized_data = scaler.fit_transform(data)
```

原始数据集中的值可能在每一列之间存在很大差异，但是转换后的数据集将包含以 0 为中心的数字列，其范围与每一列的标准差成正比。将 StandardScaler 应用于数据集，生成的前 10 行如下所示：

```
[[ 0.93457642  2.36212718]
 [ 1.95483237 -0.35495682]
 [ 1.56870474  1.05328794]
 [-0.99574783  1.61403698]
 [ 1.86379415  1.06093451]
 [-0.71007617  0.5486138 ]
 [ 1.05700714 -0.02615397]
 [-0.39363986 -0.26880538]
 [-0.57603023  0.90672853]
 [-0.71384326  1.4343424 ]]
```

而它将生成如图 5-12 所示的分布。同样地，数据的形状没有发生改变，但*定义*该形状的值却发生了很大变化。

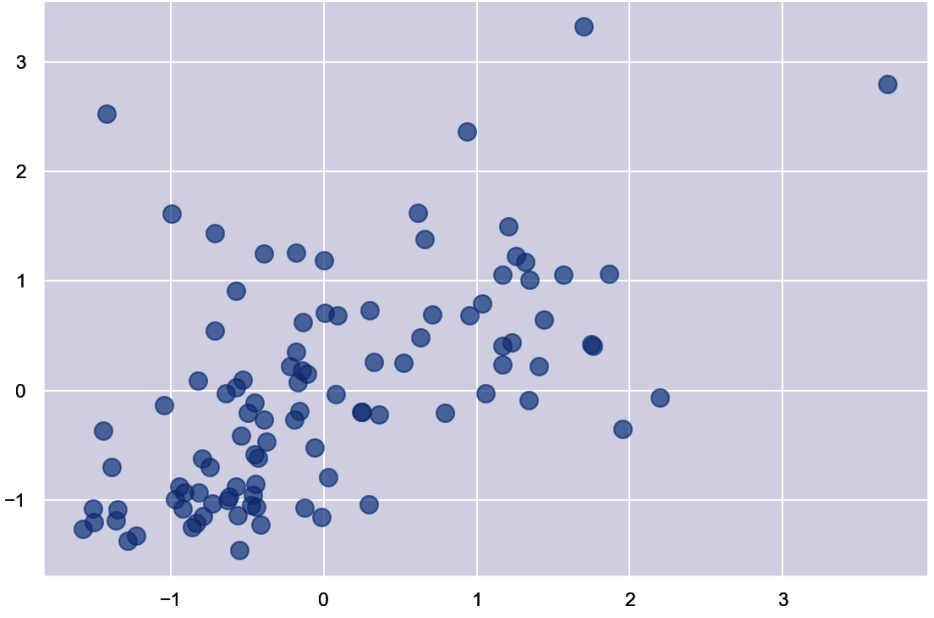


图 5-12 用 StandardScaler 归一化的数据

用标准化的数据进行训练时，SVM 的表现熊掌是最好的。即使所有列都有类似的范围，情况也是召引（顺便说一句，神经网络也是如此）。后者最典型的情况就是图像数据。图像数据集的每一列虽然都有类似的取值范围（每一列都是 0~255 的像素值），但仍然能从归一化中受益。当然也会有例外。但是，在不了解数据的分布——尤其是它是否有单位方差——的情况下，将一堆数据扔给 SVM 通常是个错误。

5.4 管道化

如果对用于训练一个机器学习模型的值进行归一化或标准化，那么必须对模型的 predict

方法的输入值进行同样的转换。也就是说，如果像下面这样训练一个模型：

```
model = SVC()
scaler = StandardScaler()
x = scaler.fit_transform(x)
model.fit(x, y)
```

那么必须像下面这样进行预测：

```
input = [0, 1, 2, 3, 4]
model.predict([scaler.transform([input])
```

否则，会得到无意义的预测结果。

为了简化编码，并防止你忘记以同样的方式转换训练数据和预测数据，Scikit 提供了 make_pipeline 函数（https://oreil.ly/HHN2p）。它允许将预测模型——Scikit 称之为估计器（estimators），或者像 SVC 这样的类的实例——与模型的输入数据转换进行合并。下例展示了如何使用 make_pipeline 来确保任何输入到模型的数据也是用 StandardScaler 来转换的：

```
# 训练模型
pipe = make_pipeline(StandardScaler(), SVC())
pipe.fit(x, y)

# 用模型做预测
input = [0, 1, 2, 3, 4]
pipe.predict([input])
```

有了这个管道后，用于训练模型的数据会应用 StandardScaler，而用于预测的数据也会以同样的方式进行转换。

那么，如果想用 GridSearchCV 为一个合并了数据转换和估计器的管道找出最佳参数集呢？这并不难，但有一个技巧需要知道。它涉及到在传递给 GridSearchCV 的 param_grid 字典中使用以双下划线开头的类名，如下例所示：

```
pipe = make_pipeline(StandardScaler(), SVC())

grid = {
    'svc__C': [0.01, 0.1, 1, 10, 100],
    'svc__gamma': [0.01, 0.25, 0.5, 0.75, 1.0],
    'svc__kernel': ['rbf', 'poly']
}

grid_search = GridSearchCV(estimator=pipe, param_grid=grid, cv=5, verbose=2)
grid_search.fit(x, y) # 用参数值的不同组合来训练模型
```

这个例子对模型进行 250 次训练，以便为管道中的 SVC 实例找到 kernel、C 和 gamma 的最佳组合。注意“svc__”这个命名模式，它与传递给 make_pipeline 函数的 SVC 实例相对应。

5.5 使用 SVM 进行面部识别

现代的面部识别一般通过神经网络来实现，但支持向量机同样靠谱。让我们建立一个面部识别模型来进行证明。我们准备使用 Labeled Faces in the Wild (LFW)数据集（https://oreil.ly/xG3LG），它包含了从网上收集的 13000 多张名人面部图像，并作为一个样本数据集内置于 Scikit。在该数据集所代表的 5000 多人中，有 1680 人有两张或更多的面部图像，而只有 5 人有 100 张或更多。我们将每个人的最小面孔数设置为 100，这意味将导入和那 5 个名人对应的 5 组面部图像。每张面部图像都有对应的人名标签。

首先创建一个新的 Jupyter 笔记本，并使用以下语句来加载数据集：

```
import numpy as np
import pandas as pd
from sklearn.datasets import fetch_lfw_people

faces = fetch_lfw_people(min_faces_per_person=100)
print(faces.target_names)
print(faces.images.shape)
```

总共加载了 1140 张面部图像。每张图的尺寸为 47×62 像素，每张图片共有 2914 个像素。这意味着该数据集包含 2914 个特征。使用以下代码来显示数据集前 24 张图和对应的人：

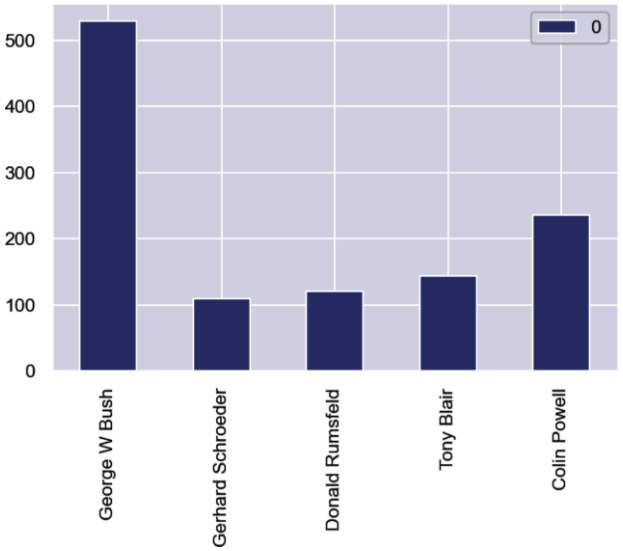
```
%matplotlib inline
import matplotlib.pyplot as plt

fig, ax = plt.subplots(3, 8, figsize=(18, 10))
for i, axi in enumerate(ax.flat):
    axi.imshow(faces.images[i], cmap='gist_gray')
    axi.set(xticks=[], yticks=[], xlabel=faces.target_names[faces.target[i]])
```

下面是输出：

通过生成直方图来显示为每个人导入了多少张面部图像，从而检查数据集的平衡性：

```
import seaborn as sns
sns.set()

from collections import Counter
counts = Counter(faces.target)
names = {}

for key in counts.keys():
    names[faces.target_names[key]] = counts[key]

df = pd.DataFrame.from_dict(names, orient='index')
df.plot(kind='bar')
```

输出结果显示，George W. Bush（小布什）的图像远远多于数据集中其他任何人的图像。



分类模型最好用平衡的数据集来训练。所以，使用以下代码，将数据集缩减为每人 100 张图像：

```
mask = np.zeros(faces.target.shape, dtype=bool)

for target in np.unique(faces.target):
    mask[np.where(faces.target == target)[0][:100]] = 1

x = faces.data[mask]
y = faces.target[mask]
x.shape
```

注意，*x* 包含 500 张面部图像，*y* 包含与之对应的标签。0 代表 Colin Powell（科林·鲍威尔），1 代表 Donald Rumsfeld（唐纳德·拉姆斯菲尔德），以此类推。现在让我们看看 SVM 是否能够理解这些数据。将训练三个不同的模型：一个使用线性核，一个使用多项式核，还有一个使用 RBF 核。在每种情况下，都将使用 `GridSearchCV` 来优化超参数。从一个线性模型和四个不同的 `C` 值开始：

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV

svc = SVC(kernel='linear')

grid = {
    'C': [0.1, 1, 10, 100]
}

grid_search = GridSearchCV(estimator=svc, param_grid=grid, cv=5, verbose=2)
grid_search.fit(x, y) # 用不同参数训练模型
grid_search.best_score_
```

这个模型达到了 84.4%的交叉验证准确率。我们也许能通过对图像数据进行标准化来提高准确率。再次运行相同的网格查找（grid search），但这次使用 `StandardScaler` 对所有像素值应用单位方差：

```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
svc = SVC(kernel='linear')
pipe = make_pipeline(scaler, svc)

grid = {
    'svc__C': [0.1, 1, 10, 100]
}

grid_search = GridSearchCV(estimator=pipe, param_grid=grid, cv=5, verbose=2)
grid_search.fit(x, y)
grid_search.best_score_
```

对数据进行标准化后，精度得到了提高。`C` 的什么值产生了这个精度？

```
grid_search.best_params_
```

多项式核有可能胜过线性核吗？有一个简单的方法可以找到答案。注意，这次在 `grid` 参数中引入了 `gamma` 和 `degree` 参数。这些参数和 `C` 一起，会显著影响多项式核对训练数据的拟合能力。

```
scaler = StandardScaler()
svc = SVC(kernel='poly')
pipe = make_pipeline(scaler, svc)

grid = {
    'svc__C': [0.1, 1, 10, 100],
    'svc__gamma': [0.01, 0.25, 0.5, 0.75, 1],
    'svc__degree': [1, 2, 3, 4, 5]
}

grid_search = GridSearchCV(estimator=pipe, param_grid=grid, cv=5, verbose=2)
grid_search.fit(x, y) # 用不同的参数值组合来训练模型
grid_search.best_score_
```

多项式核取得了与线性核相同的准确率。是什么参数值导致了这个结果？

```
grid_search.best_params_
```

best_params_属性表明，degree 的最佳值是 1，这意味着多项式核的表现和线性核一样。既然如此，它能达到同样的准确率也就不足为奇了。那么，RBF 核的表现是否更好？

```
scaler = StandardScaler()
svc = SVC(kernel='rbf')
pipe = make_pipeline(scaler, svc)

grid = {
    'svc__C': [0.1, 1, 10, 100],
    'svc__gamma': [0.01, 0.25, 0.5, 0.75, 1.0]
}

grid_search = GridSearchCV(estimator=pipe, param_grid=grid, cv=5, verbose=2)
grid_search.fit(x, y)
grid_search.best_score_
```

RBF 核的表现不如线性核和多项式核好。这里得到了一个教训。在和非线性数据拟合的时候，RBF 核的表现通常比其他核更好。但是，这并不绝对。这就是为什么在使用 SVM 的时候，最佳策略应该是为不同的核尝试不同的参数值。最佳组合将因数据集而异。对于 LFW 数据集，似乎线性核最好。这其实正合我们的意思，因为线性核是 Scikit 提供的所有核中最快的。

除了 SVC 类之外，Scikit-Learn 还包括名为 LinearSVC 和 NuSVC 的 SVM 分类器。后者支持与 SVC 类相同的一套核（kernels），但它使用一个称为 nu 的正则化参数代替了 C，以不同的方式控制拟合的严格程度。NuSVC 不像 SVC 那样可以扩展到大数据集，根据我的经验，它很少会用得到。LinearSVC 只实现了线性核，但它使用了一种不同的优化算法，训练速度更快。如果用 SVC 训练很慢，而且你确定线性核能生成最好的模型，那么可以考虑把 SVC 换成 LinearSVC。如果使用 GridSearchCV 训练一个模型数百次，那么即使对于规模不大的数据集来说，更快的训练时间也是有意义的。要更多地了解这两个类在功能上的差异，请参考 Angela Shi 撰写的“SVM with Scikit-Learn: What You Should Know”一文（https://oreil.ly/OQl1k）。

混淆矩阵是对模型准确率进行可视化的一个好方法。让我们拆分数据集，用 80%的图像训练一个优化的线性模型，再用剩余 20%测试它，并在混淆矩阵中显示结果。

第一步是拆分数据集。注意 stratify = y 参数，它确保在训练数据集和测试数据集中，每个类别的样本比例与原始数据集相同。在本例中，训练数据集为 5 个人中的每个人包含了 20 个样本。

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y, train_size=0.8,
                                                    stratify=y, random_state=0)
```

现在，用网格查找所揭示的最佳 C 值训练一个线性 SVM：

```
scaler = StandardScaler()
svc = SVC(kernel='linear', C=0.1)
pipe = make_pipeline(scaler, svc)
pipe.fit(x_train, y_train)
```

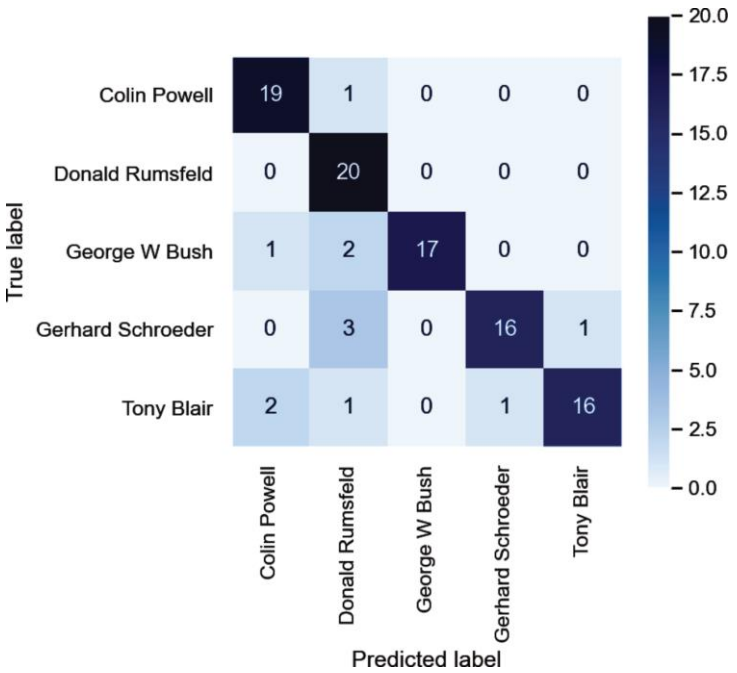
交叉验证该模型以确认其准确率：

```
from sklearn.model_selection import cross_val_score

cross_val_score(pipe, x, y, cv=5).mean()
```

使用混淆矩阵来观察模型在测试数据上的表现：

```
from sklearn.metrics import ConfusionMatrixDisplay as cmd

fig, ax = plt.subplots(figsize=(6, 6))
ax.grid(False)
cmd.from_estimator(pipe, x_test, y_test, display_labels=faces.target_names,
                   cmap='Blues', xticks_rotation='vertical', ax=ax)
```

下面是输出结果：



该模型在 20 次中正确识别了 Colin Powell 19 次，Donald Rumsfeld 20 次，以此类推。这并不坏。这也是体会支持向量机（SVM）实际表现的一个好例子。

5.6 小结

在对数据集进行拟合的时候，支持向量机（SVM）的表现经常比其他学习算法更好。支持向量机是最大边缘（maximum-margin）分类器，使用所谓的“核技巧”（kernel tricks）来模拟为数据增加维度。它的理论是，如果 *n* 大于 *m*，那么在 *m* 维上不能线性分离的数据在 *n* 维上也许是可以分离的。SVM 最常的用途就是分类，但它们也可用于回归。作为一个实验，尝试用 `SVR` 替换第 2 章打车费用例子中的 `GradientBoostingRegressor`，并使用 `GridSearchCV` 来优化模型的超参数。哪个模型能产生最高的交叉验证决定系数？

使用归一化为单位方差的数据，SVM 的训练效果通常会更好。即使所有列中的值都有相似的范围，这样处理也能获得更好的效果。如果没有相似的范围，那么更是如此。可以使用 Scikit 的 `StandardScaler` 类向数据应用单位方差。单位方差是通过将一列中的值减去该列中所有值的平均值，结果再除以标准差来获得的。Scikit 的 `make_pipeline` 函数允许将 `StandardScaler` 等转换器和 `SVC` 等分类器合并为一个逻辑单元（管道化），以确保传递给 `fit`、`predict` 和 `predict_proba` 的数据经历同样的转换。

SVM 需要调优（tuning）以获得最佳准确率。调优意味着为 `C`、`gamma` 和 `kernel` 等参数找到恰当的值，它需要尝试不同的参数值组合并评估结果。Scikit 提供了 `GridSearchCV` 等类在这方面提供帮助。但是，由于要为每个独特的参数值组合训练一次模型，所以会增加训练时间。

SVM 能为复杂的数据集拟合数学模型，这已经很神奇了。但在我看来，与主成分分析（Principal Component Analysis，PCA）所做的数字体操相比，它还不够神奇。PCA 解决了机器学习中经常遇到的各种问题。我在介绍 PCA 时经常会告诉听众，它才是机器学习中最不为人所知的秘密。但是，在第 6 章之后，它就不再是一个秘密了。