

Introduction to Machine Learning with Python

A GUIDE FOR DATA SCIENTISTS

Andreas C. Müller & Sarah Guido

Introduction to Machine Learning with Python

Machine learning has become an integral part of many commercial applications and research projects, but this field is not exclusive to large companies with extensive research teams. If you use Python, even as a beginner, this book will teach you practical ways to build your own machine learning solutions. With all the data available today, machine learning applications are limited only by your imagination.

You'll learn the steps necessary to create a successful machine learning application with Python and the scikit-learn library. Authors Andreas Müller and Sarah Guido focus on the practical aspects of using machine learning algorithms, rather than the math behind them. Familiarity with the NumPy and matplotlib libraries will help you get even more from this book.

With this book, you'll learn:

- Fundamental concepts and applications of machine learning
- Advantages and shortcomings of widely used machine learning algorithms
- How to represent data processed by machine learning, including which data aspects to focus on
- Advanced methods for model evaluation and parameter tuning
- The concept of pipelines for chaining models and encapsulating your workflow
- Methods for working with text data, including text-specific processing techniques
- Suggestions for improving your machine learning and data science skills

“This book is a fantastic, super-practical resource for anyone who wants to start using machine learning in Python—I just wish it had existed when I started using scikit-learn!”

—Hanna Wallach

Senior Researcher, Microsoft Research

Andreas Müller earned a PhD in machine learning from the University of Bonn. After working as a machine learning researcher on computer vision applications at Amazon, he joined the Center for Data Science at New York University. He's also a maintainer and core contributor to scikit-learn.

Sarah Guido is a data scientist who has spent a lot of time working with startups, and most recently served as Lead Data Scientist at Bitly. An accomplished conference speaker, Sarah earned a Master of Science in Information from the University of Michigan.

PYTHON/MACHINE LEARNING

US \$49.99

CAN \$57.99

ISBN: 978-1-449-36941-5



5 4 9 9 9
9 781449 369415



Twitter: @oreillymedia
facebook.com/oreilly

Introduction to Machine Learning with Python

A Guide for Data Scientists

Andreas C. Müller and Sarah Guido

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Introduction to Machine Learning with Python

by Andreas C. Müller and Sarah Guido

Copyright © 2017 Sarah Guido, Andreas Müller. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Dawn Schanafelt

Indexer: Judy McConville

Production Editor: Kristen Brown

Interior Designer: David Futato

Copyeditor: Rachel Head

Cover Designer: Karen Montgomery

Proofreader: Jasmine Kwityn

Illustrator: Rebecca Demarest

October 2016: First Edition

Revision History for the First Edition

2016-09-22: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781449369415> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Introduction to Machine Learning with Python*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-449-36941-5

[LSI]

Table of Contents

Preface.....	vii
1. Introduction.....	1
Why Machine Learning?	1
Problems Machine Learning Can Solve	2
Knowing Your Task and Knowing Your Data	4
Why Python?	5
scikit-learn	5
Installing scikit-learn	6
Essential Libraries and Tools	7
Jupyter Notebook	7
NumPy	7
SciPy	8
matplotlib	9
pandas	10
mlearn	11
Python 2 Versus Python 3	12
Versions Used in this Book	12
A First Application: Classifying Iris Species	13
Meet the Data	14
Measuring Success: Training and Testing Data	17
First Things First: Look at Your Data	19
Building Your First Model: k-Nearest Neighbors	20
Making Predictions	22
Evaluating the Model	22
Summary and Outlook	23

2. Supervised Learning.....	25
Classification and Regression	25
Generalization, Overfitting, and Underfitting	26
Relation of Model Complexity to Dataset Size	29
Supervised Machine Learning Algorithms	29
Some Sample Datasets	30
k-Nearest Neighbors	35
Linear Models	45
Naive Bayes Classifiers	68
Decision Trees	70
Ensembles of Decision Trees	83
Kernelized Support Vector Machines	92
Neural Networks (Deep Learning)	104
Uncertainty Estimates from Classifiers	119
The Decision Function	120
Predicting Probabilities	122
Uncertainty in Multiclass Classification	124
Summary and Outlook	127
3. Unsupervised Learning and Preprocessing.....	131
Types of Unsupervised Learning	131
Challenges in Unsupervised Learning	132
Preprocessing and Scaling	132
Different Kinds of Preprocessing	133
Applying Data Transformations	134
Scaling Training and Test Data the Same Way	136
The Effect of Preprocessing on Supervised Learning	138
Dimensionality Reduction, Feature Extraction, and Manifold Learning	140
Principal Component Analysis (PCA)	140
Non-Negative Matrix Factorization (NMF)	156
Manifold Learning with t-SNE	163
Clustering	168
k-Means Clustering	168
Agglomerative Clustering	182
DBSCAN	187
Comparing and Evaluating Clustering Algorithms	191
Summary of Clustering Methods	207
Summary and Outlook	208
4. Representing Data and Engineering Features.....	211
Categorical Variables	212
One-Hot-Encoding (Dummy Variables)	213

Numbers Can Encode Categoricals	218
Binning, Discretization, Linear Models, and Trees	220
Interactions and Polynomials	224
Univariate Nonlinear Transformations	232
Automatic Feature Selection	236
Univariate Statistics	236
Model-Based Feature Selection	238
Iterative Feature Selection	240
Utilizing Expert Knowledge	242
Summary and Outlook	250
5. Model Evaluation and Improvement.....	251
Cross-Validation	252
Cross-Validation in scikit-learn	253
Benefits of Cross-Validation	254
Stratified k-Fold Cross-Validation and Other Strategies	254
Grid Search	260
Simple Grid Search	261
The Danger of Overfitting the Parameters and the Validation Set	261
Grid Search with Cross-Validation	263
Evaluation Metrics and Scoring	275
Keep the End Goal in Mind	275
Metrics for Binary Classification	276
Metrics for Multiclass Classification	296
Regression Metrics	299
Using Evaluation Metrics in Model Selection	300
Summary and Outlook	302
6. Algorithm Chains and Pipelines.....	305
Parameter Selection with Preprocessing	306
Building Pipelines	308
Using Pipelines in Grid Searches	309
The General Pipeline Interface	312
Convenient Pipeline Creation with <code>make_pipeline</code>	313
Accessing Step Attributes	314
Accessing Attributes in a Grid-Searched Pipeline	315
Grid-Searching Preprocessing Steps and Model Parameters	317
Grid-Searching Which Model To Use	319
Summary and Outlook	320
7. Working with Text Data.....	323
Types of Data Represented as Strings	323

Example Application: Sentiment Analysis of Movie Reviews	325
Representing Text Data as a Bag of Words	327
Applying Bag-of-Words to a Toy Dataset	329
Bag-of-Words for Movie Reviews	330
Stopwords	334
Rescaling the Data with tf-idf	336
Investigating Model Coefficients	338
Bag-of-Words with More Than One Word (n-Grams)	339
Advanced Tokenization, Stemming, and Lemmatization	344
Topic Modeling and Document Clustering	347
Latent Dirichlet Allocation	348
Summary and Outlook	355
8. Wrapping Up.....	357
Approaching a Machine Learning Problem	357
Humans in the Loop	358
From Prototype to Production	359
Testing Production Systems	359
Building Your Own Estimator	360
Where to Go from Here	361
Theory	361
Other Machine Learning Frameworks and Packages	362
Ranking, Recommender Systems, and Other Kinds of Learning	363
Probabilistic Modeling, Inference, and Probabilistic Programming	363
Neural Networks	364
Scaling to Larger Datasets	364
Honing Your Skills	365
Conclusion	366
Index.....	367

Preface

Machine learning is an integral part of many commercial applications and research projects today, in areas ranging from medical diagnosis and treatment to finding your friends on social networks. Many people think that machine learning can only be applied by large companies with extensive research teams. In this book, we want to show you how easy it can be to build machine learning solutions yourself, and how to best go about it. With the knowledge in this book, you can build your own system for finding out how people feel on Twitter, or making predictions about global warming. The applications of machine learning are endless and, with the amount of data available today, mostly limited by your imagination.

Who Should Read This Book

This book is for current and aspiring machine learning practitioners looking to implement solutions to real-world machine learning problems. This is an introductory book requiring no previous knowledge of machine learning or artificial intelligence (AI). We focus on using Python and the `scikit-learn` library, and work through all the steps to create a successful machine learning application. The methods we introduce will be helpful for scientists and researchers, as well as data scientists working on commercial applications. You will get the most out of the book if you are somewhat familiar with Python and the NumPy and `matplotlib` libraries.

We made a conscious effort not to focus too much on the math, but rather on the practical aspects of using machine learning algorithms. As mathematics (probability theory, in particular) is the foundation upon which machine learning is built, we won't go into the analysis of the algorithms in great detail. If you are interested in the mathematics of machine learning algorithms, we recommend the book *The Elements of Statistical Learning* (Springer) by Trevor Hastie, Robert Tibshirani, and Jerome Friedman, which is available for free at [the authors' website](#). We will also not describe how to write machine learning algorithms from scratch, and will instead focus on

how to use the large array of models already implemented in `scikit-learn` and other libraries.

Why We Wrote This Book

There are many books on machine learning and AI. However, all of them are meant for graduate students or PhD students in computer science, and they're full of advanced mathematics. This is in stark contrast with how machine learning is being used, as a commodity tool in research and commercial applications. Today, applying machine learning does not require a PhD. However, there are few resources out there that fully cover all the important aspects of implementing machine learning in practice, without requiring you to take advanced math courses. We hope this book will help people who want to apply machine learning without reading up on years' worth of calculus, linear algebra, and probability theory.

Navigating This Book

This book is organized roughly as follows:

- [Chapter 1](#) introduces the fundamental concepts of machine learning and its applications, and describes the setup we will be using throughout the book.
- Chapters [2](#) and [3](#) describe the actual machine learning algorithms that are most widely used in practice, and discuss their advantages and shortcomings.
- [Chapter 4](#) discusses the importance of how we represent data that is processed by machine learning, and what aspects of the data to pay attention to.
- [Chapter 5](#) covers advanced methods for model evaluation and parameter tuning, with a particular focus on cross-validation and grid search.
- [Chapter 6](#) explains the concept of pipelines for chaining models and encapsulating your workflow.
- [Chapter 7](#) shows how to apply the methods described in earlier chapters to text data, and introduces some text-specific processing techniques.
- [Chapter 8](#) offers a high-level overview, and includes references to more advanced topics.

While Chapters [2](#) and [3](#) provide the actual algorithms, understanding all of these algorithms might not be necessary for a beginner. If you need to build a machine learning system ASAP, we suggest starting with [Chapter 1](#) and the opening sections of [Chapter 2](#), which introduce all the core concepts. You can then skip to “[Summary and Outlook](#)” on page [127](#) in [Chapter 2](#), which includes a list of all the supervised models that we cover. Choose the model that best fits your needs and flip back to read the

section devoted to it for details. Then you can use the techniques in [Chapter 5](#) to evaluate and tune your model.

Online Resources

While studying this book, definitely refer to the [scikit-learn](#) website for more in-depth documentation of the classes and functions, and many examples. There is also a video course created by Andreas Müller, “Advanced Machine Learning with scikit-learn,” that supplements this book. You can find it at http://bit.ly/advanced_machine_learning_scikit-learn.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords. Also used for commands and module and package names.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip or suggestion.



This element signifies a general note.



This icon indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, IPython notebooks, etc.) is available for download at https://github.com/amueller/introduction_to_ml_with_python.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*An Introduction to Machine Learning with Python* by Andreas C. Müller and Sarah Guido (O'Reilly). Copyright 2017 Sarah Guido and Andreas Müller, 978-1-449-36941-5.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online



Safari Books Online is an on-demand digital library that delivers expert **content** in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of **plans and pricing** for **enterprise**, **government**, **education**, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que,

Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds **more**. For more information about Safari Books Online, please visit us [online](#).

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <http://bit.ly/intro-machine-learning-python>.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

From Andreas

Without the help and support of a large group of people, this book would never have existed.

I would like to thank the editors, Meghan Blanchette, Brian MacDonald, and in particular Dawn Schanafelt, for helping Sarah and me make this book a reality.

I want to thank my reviewers, Thomas Caswell, Olivier Grisel, Stefan van der Walt, and John Myles White, who took the time to read the early versions of this book and provided me with invaluable feedback—in addition to being some of the cornerstones of the scientific open source ecosystem.

I am forever thankful for the welcoming open source scientific Python community, especially the contributors to `scikit-learn`. Without the support and help from this community, in particular from Gael Varoquaux, Alex Gramfort, and Olivier Grisel, I would never have become a core contributor to `scikit-learn` or learned to understand this package as well as I do now. My thanks also go out to all the other contributors who donate their time to improve and maintain this package.

I'm also thankful for the discussions with many of my colleagues and peers that helped me understand the challenges of machine learning and gave me ideas for structuring a textbook. Among the people I talk to about machine learning, I specifically want to thank Brian McFee, Daniela Huttenkopp, Joel Nothman, Gilles Louppe, Hugo Bowne-Anderson, Sven Kreis, Alice Zheng, Kyunghyun Cho, Pablo Baber, and Dan Cervone.

My thanks also go out to Rachel Rakov, who was an eager beta tester and proofreader of an early version of this book, and helped me shape it in many ways.

On the personal side, I want to thank my parents, Harald and Margot, and my sister, Miriam, for their continuing support and encouragement. I also want to thank the many people in my life whose love and friendship gave me the energy and support to undertake such a challenging task.

From Sarah

I would like to thank Meg Blanchette, without whose help and guidance this project would not have even existed. Thanks to Celia La and Brian Carlson for reading in the early days. Thanks to the O'Reilly folks for their endless patience. And finally, thanks to DTS, for your everlasting and endless support.

CHAPTER 1

Introduction

Machine learning is about extracting knowledge from data. It is a research field at the intersection of statistics, artificial intelligence, and computer science and is also known as predictive analytics or statistical learning. The application of machine learning methods has in recent years become ubiquitous in everyday life. From automatic recommendations of which movies to watch, to what food to order or which products to buy, to personalized online radio and recognizing your friends in your photos, many modern websites and devices have machine learning algorithms at their core. When you look at a complex website like Facebook, Amazon, or Netflix, it is very likely that every part of the site contains multiple machine learning models.

Outside of commercial applications, machine learning has had a tremendous influence on the way data-driven research is done today. The tools introduced in this book have been applied to diverse scientific problems such as understanding stars, finding distant planets, discovering new particles, analyzing DNA sequences, and providing personalized cancer treatments.

Your application doesn't need to be as large-scale or world-changing as these examples in order to benefit from machine learning, though. In this chapter, we will explain why machine learning has become so popular and discuss what kinds of problems can be solved using machine learning. Then, we will show you how to build your first machine learning model, introducing important concepts along the way.

Why Machine Learning?

In the early days of “intelligent” applications, many systems used handcoded rules of “if” and “else” decisions to process data or adjust to user input. Think of a spam filter whose job is to move the appropriate incoming email messages to a spam folder. You could make up a blacklist of words that would result in an email being marked as

spam. This would be an example of using an expert-designed rule system to design an “intelligent” application. Manually crafting decision rules is feasible for some applications, particularly those in which humans have a good understanding of the process to model. However, using handcoded rules to make decisions has two major disadvantages:

- The logic required to make a decision is specific to a single domain and task. Changing the task even slightly might require a rewrite of the whole system.
- Designing rules requires a deep understanding of how a decision should be made by a human expert.

One example of where this handcoded approach will fail is in detecting faces in images. Today, every smartphone can detect a face in an image. However, face detection was an unsolved problem until as recently as 2001. The main problem is that the way in which pixels (which make up an image in a computer) are “perceived” by the computer is very different from how humans perceive a face. This difference in representation makes it basically impossible for a human to come up with a good set of rules to describe what constitutes a face in a digital image.

Using machine learning, however, simply presenting a program with a large collection of images of faces is enough for an algorithm to determine what characteristics are needed to identify a face.

Problems Machine Learning Can Solve

The most successful kinds of machine learning algorithms are those that automate decision-making processes by generalizing from known examples. In this setting, which is known as *supervised learning*, the user provides the algorithm with pairs of inputs and desired outputs, and the algorithm finds a way to produce the desired output given an input. In particular, the algorithm is able to create an output for an input it has never seen before without any help from a human. Going back to our example of spam classification, using machine learning, the user provides the algorithm with a large number of emails (which are the input), together with information about whether any of these emails are spam (which is the desired output). Given a new email, the algorithm will then produce a prediction as to whether the new email is spam.

Machine learning algorithms that learn from input/output pairs are called supervised learning algorithms because a “teacher” provides supervision to the algorithms in the form of the desired outputs for each example that they learn from. While creating a dataset of inputs and outputs is often a laborious manual process, supervised learning algorithms are well understood and their performance is easy to measure. If your application can be formulated as a supervised learning problem, and you are able to

create a dataset that includes the desired outcome, machine learning will likely be able to solve your problem.

Examples of supervised machine learning tasks include:

Identifying the zip code from handwritten digits on an envelope

Here the input is a scan of the handwriting, and the desired output is the actual digits in the zip code. To create a dataset for building a machine learning model, you need to collect many envelopes. Then you can read the zip codes yourself and store the digits as your desired outcomes.

Determining whether a tumor is benign based on a medical image

Here the input is the image, and the output is whether the tumor is benign. To create a dataset for building a model, you need a database of medical images. You also need an expert opinion, so a doctor needs to look at all of the images and decide which tumors are benign and which are not. It might even be necessary to do additional diagnosis beyond the content of the image to determine whether the tumor in the image is cancerous or not.

Detecting fraudulent activity in credit card transactions

Here the input is a record of the credit card transaction, and the output is whether it is likely to be fraudulent or not. Assuming that you are the entity distributing the credit cards, collecting a dataset means storing all transactions and recording if a user reports any transaction as fraudulent.

An interesting thing to note about these examples is that although the inputs and outputs look fairly straightforward, the data collection process for these three tasks is vastly different. While reading envelopes is laborious, it is easy and cheap. Obtaining medical imaging and diagnoses, on the other hand, requires not only expensive machinery but also rare and expensive expert knowledge, not to mention the ethical concerns and privacy issues. In the example of detecting credit card fraud, data collection is much simpler. Your customers will provide you with the desired output, as they will report fraud. All you have to do to obtain the input/output pairs of fraudulent and nonfraudulent activity is wait.

Unsupervised algorithms are the other type of algorithm that we will cover in this book. In unsupervised learning, only the input data is known, and no known output data is given to the algorithm. While there are many successful applications of these methods, they are usually harder to understand and evaluate.

Examples of unsupervised learning include:

Identifying topics in a set of blog posts

If you have a large collection of text data, you might want to summarize it and find prevalent themes in it. You might not know beforehand what these topics are, or how many topics there might be. Therefore, there are no known outputs.

Segmenting customers into groups with similar preferences

Given a set of customer records, you might want to identify which customers are similar, and whether there are groups of customers with similar preferences. For a shopping site, these might be “parents,” “bookworms,” or “gamers.” Because you don’t know in advance what these groups might be, or even how many there are, you have no known outputs.

Detecting abnormal access patterns to a website

To identify abuse or bugs, it is often helpful to find access patterns that are different from the norm. Each abnormal pattern might be very different, and you might not have any recorded instances of abnormal behavior. Because in this example you only observe traffic, and you don’t know what constitutes normal and abnormal behavior, this is an unsupervised problem.

For both supervised and unsupervised learning tasks, it is important to have a representation of your input data that a computer can understand. Often it is helpful to think of your data as a table. Each data point that you want to reason about (each email, each customer, each transaction) is a row, and each property that describes that data point (say, the age of a customer or the amount or location of a transaction) is a column. You might describe users by their age, their gender, when they created an account, and how often they have bought from your online shop. You might describe the image of a tumor by the grayscale values of each pixel, or maybe by using the size, shape, and color of the tumor.

Each entity or row here is known as a *sample* (or data point) in machine learning, while the columns—the properties that describe these entities—are called *features*.

Later in this book we will go into more detail on the topic of building a good representation of your data, which is called *feature extraction* or *feature engineering*. You should keep in mind, however, that no machine learning algorithm will be able to make a prediction on data for which it has no information. For example, if the only feature that you have for a patient is their last name, no algorithm will be able to predict their gender. This information is simply not contained in your data. If you add another feature that contains the patient’s first name, you will have much better luck, as it is often possible to tell the gender by a person’s first name.

Knowing Your Task and Knowing Your Data

Quite possibly the most important part in the machine learning process is understanding the data you are working with and how it relates to the task you want to solve. It will not be effective to randomly choose an algorithm and throw your data at it. It is necessary to understand what is going on in your dataset before you begin building a model. Each algorithm is different in terms of what kind of data and what problem setting it works best for. While you are building a machine learning solution, you should answer, or at least keep in mind, the following questions:

- What question(s) am I trying to answer? Do I think the data collected can answer that question?
- What is the best way to phrase my question(s) as a machine learning problem?
- Have I collected enough data to represent the problem I want to solve?
- What features of the data did I extract, and will these enable the right predictions?
- How will I measure success in my application?
- How will the machine learning solution interact with other parts of my research or business product?

In a larger context, the algorithms and methods in machine learning are only one part of a greater process to solve a particular problem, and it is good to keep the big picture in mind at all times. Many people spend a lot of time building complex machine learning solutions, only to find out they don't solve the right problem.

When going deep into the technical aspects of machine learning (as we will in this book), it is easy to lose sight of the ultimate goals. While we will not discuss the questions listed here in detail, we still encourage you to keep in mind all the assumptions that you might be making, explicitly or implicitly, when you start building machine learning models.

Why Python?

Python has become the lingua franca for many data science applications. It combines the power of general-purpose programming languages with the ease of use of domain-specific scripting languages like MATLAB or R. Python has libraries for data loading, visualization, statistics, natural language processing, image processing, and more. This vast toolbox provides data scientists with a large array of general- and special-purpose functionality. One of the main advantages of using Python is the ability to interact directly with the code, using a terminal or other tools like the Jupyter Notebook, which we'll look at shortly. Machine learning and data analysis are fundamentally iterative processes, in which the data drives the analysis. It is essential for these processes to have tools that allow quick iteration and easy interaction.

As a general-purpose programming language, Python also allows for the creation of complex graphical user interfaces (GUIs) and web services, and for integration into existing systems.

scikit-learn

scikit-learn is an open source project, meaning that it is free to use and distribute, and anyone can easily obtain the source code to see what is going on behind the

scenes. The `scikit-learn` project is constantly being developed and improved, and it has a very active user community. It contains a number of state-of-the-art machine learning algorithms, as well as comprehensive [documentation](#) about each algorithm. `scikit-learn` is a very popular tool, and the most prominent Python library for machine learning. It is widely used in industry and academia, and a wealth of tutorials and code snippets are available online. `scikit-learn` works well with a number of other scientific Python tools, which we will discuss later in this chapter.

While reading this, we recommend that you also browse the [scikit-learn user guide](#) and API documentation for additional details on and many more options for each algorithm. The online documentation is very thorough, and this book will provide you with all the prerequisites in machine learning to understand it in detail.

Installing scikit-learn

`scikit-learn` depends on two other Python packages, `NumPy` and `SciPy`. For plotting and interactive development, you should also install `matplotlib`, `IPython`, and the Jupyter Notebook. We recommend using one of the following prepackaged Python distributions, which will provide the necessary packages:

Anaconda

A Python distribution made for large-scale data processing, predictive analytics, and scientific computing. Anaconda comes with `NumPy`, `SciPy`, `matplotlib`, `pandas`, `IPython`, Jupyter Notebook, and `scikit-learn`. Available on Mac OS, Windows, and Linux, it is a very convenient solution and is the one we suggest for people without an existing installation of the scientific Python packages. Anaconda now also includes the commercial Intel MKL library for free. Using MKL (which is done automatically when Anaconda is installed) can give significant speed improvements for many algorithms in `scikit-learn`.

Enthought Canopy

Another Python distribution for scientific computing. This comes with `NumPy`, `SciPy`, `matplotlib`, `pandas`, and `IPython`, but the free version does not come with `scikit-learn`. If you are part of an academic, degree-granting institution, you can request an academic license and get free access to the paid subscription version of Enthought Canopy. Enthought Canopy is available for Python 2.7.x, and works on Mac OS, Windows, and Linux.

Python(x,y)

A free Python distribution for scientific computing, specifically for Windows. Python(x,y) comes with `NumPy`, `SciPy`, `matplotlib`, `pandas`, `IPython`, and `scikit-learn`.

If you already have a Python installation set up, you can use `pip` to install all of these packages:

```
$ pip install numpy scipy matplotlib ipython scikit-learn pandas
```

Essential Libraries and Tools

Understanding what `scikit-learn` is and how to use it is important, but there are a few other libraries that will enhance your experience. `scikit-learn` is built on top of the NumPy and SciPy scientific Python libraries. In addition to NumPy and SciPy, we will be using `pandas` and `matplotlib`. We will also introduce the Jupyter Notebook, which is a browser-based interactive programming environment. Briefly, here is what you should know about these tools in order to get the most out of `scikit-learn`.¹

Jupyter Notebook

The Jupyter Notebook is an interactive environment for running code in the browser. It is a great tool for exploratory data analysis and is widely used by data scientists. While the Jupyter Notebook supports many programming languages, we only need the Python support. The Jupyter Notebook makes it easy to incorporate code, text, and images, and all of this book was in fact written as a Jupyter Notebook. All of the code examples we include can be downloaded from [GitHub](#).

NumPy

NumPy is one of the fundamental packages for scientific computing in Python. It contains functionality for multidimensional arrays, high-level mathematical functions such as linear algebra operations and the Fourier transform, and pseudorandom number generators.

In `scikit-learn`, the NumPy array is the fundamental data structure. `scikit-learn` takes in data in the form of NumPy arrays. Any data you're using will have to be converted to a NumPy array. The core functionality of NumPy is the `ndarray` class, a multidimensional (n -dimensional) array. All elements of the array must be of the same type. A NumPy array looks like this:

`In[2]:`

```
import numpy as np

x = np.array([[1, 2, 3], [4, 5, 6]])
print("x:\n{}".format(x))
```

¹ If you are unfamiliar with NumPy or `matplotlib`, we recommend reading the first chapter of the [SciPy Lecture Notes](#).

Out[2]:

```
x:  
[[1 2 3]  
 [4 5 6]]
```

We will be using NumPy *a lot* in this book, and we will refer to objects of the NumPy `ndarray` class as “NumPy arrays” or just “arrays.”

SciPy

SciPy is a collection of functions for scientific computing in Python. It provides, among other functionality, advanced linear algebra routines, mathematical function optimization, signal processing, special mathematical functions, and statistical distributions. `scikit-learn` draws from SciPy’s collection of functions for implementing its algorithms. The most important part of SciPy for us is `scipy.sparse`: this provides *sparse matrices*, which are another representation that is used for data in `scikit-learn`. Sparse matrices are used whenever we want to store a 2D array that contains mostly zeros:

In[3]:

```
from scipy import sparse  
  
# Create a 2D NumPy array with a diagonal of ones, and zeros everywhere else  
eye = np.eye(4)  
print("NumPy array:\n{}".format(eye))
```

Out[3]:

```
NumPy array:  
[[ 1.  0.  0.  0.]  
 [ 0.  1.  0.  0.]  
 [ 0.  0.  1.  0.]  
 [ 0.  0.  0.  1.]]
```

In[4]:

```
# Convert the NumPy array to a SciPy sparse matrix in CSR format  
# Only the nonzero entries are stored  
sparse_matrix = sparse.csr_matrix(eye)  
print("\nSciPy sparse CSR matrix:\n{}".format(sparse_matrix))
```

Out[4]:

```
SciPy sparse CSR matrix:  
(0, 0)    1.0  
(1, 1)    1.0  
(2, 2)    1.0  
(3, 3)    1.0
```

Usually it is not possible to create dense representations of sparse data (as they would not fit into memory), so we need to create sparse representations directly. Here is a way to create the same sparse matrix as before, using the COO format:

In[5]:

```
data = np.ones(4)
row_indices = np.arange(4)
col_indices = np.arange(4)
eye_coo = sparse.coo_matrix((data, (row_indices, col_indices)))
print("COO representation:\n{}".format(eye_coo))
```

Out[5]:

```
COO representation:
 (0, 0)    1.0
 (1, 1)    1.0
 (2, 2)    1.0
 (3, 3)    1.0
```

More details on SciPy sparse matrices can be found in the [SciPy Lecture Notes](#).

matplotlib

`matplotlib` is the primary scientific plotting library in Python. It provides functions for making publication-quality visualizations such as line charts, histograms, scatter plots, and so on. Visualizing your data and different aspects of your analysis can give you important insights, and we will be using `matplotlib` for all our visualizations. When working inside the Jupyter Notebook, you can show figures directly in the browser by using the `%matplotlib notebook` and `%matplotlib inline` commands. We recommend using `%matplotlib notebook`, which provides an interactive environment (though we are using `%matplotlib inline` to produce this book). For example, this code produces the plot in [Figure 1-1](#):

In[6]:

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

# Generate a sequence of numbers from -10 to 10 with 100 steps in between
x = np.linspace(-10, 10, 100)
# Create a second array using sine
y = np.sin(x)
# The plot function makes a line chart of one array against another
plt.plot(x, y, marker="x")
```

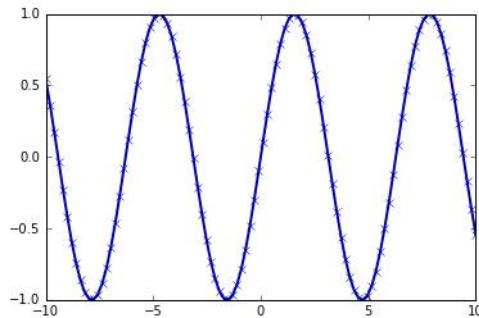


Figure 1-1. Simple line plot of the sine function using matplotlib

pandas

pandas is a Python library for data wrangling and analysis. It is built around a data structure called the `DataFrame` that is modeled after the R `DataFrame`. Simply put, a pandas `DataFrame` is a table, similar to an Excel spreadsheet. pandas provides a great range of methods to modify and operate on this table; in particular, it allows SQL-like queries and joins of tables. In contrast to NumPy, which requires that all entries in an array be of the same type, pandas allows each column to have a separate type (for example, integers, dates, floating-point numbers, and strings). Another valuable tool provided by pandas is its ability to ingest from a great variety of file formats and databases, like SQL, Excel files, and comma-separated values (CSV) files. Going into detail about the functionality of pandas is out of the scope of this book. However, *Python for Data Analysis* by Wes McKinney (O'Reilly, 2012) provides a great guide. Here is a small example of creating a `DataFrame` using a dictionary:

In[7]:

```
import pandas as pd

# create a simple dataset of people
data = {'Name': ["John", "Anna", "Peter", "Linda"],
        'Location' : ["New York", "Paris", "Berlin", "London"],
        'Age' : [24, 13, 53, 33]
       }

data_pandas = pd.DataFrame(data)
# IPython.display allows "pretty printing" of dataframes
# in the Jupyter notebook
display(data_pandas)
```

This produces the following output:

	Age	Location	Name
0	24	New York	John
1	13	Paris	Anna
2	53	Berlin	Peter
3	33	London	Linda

There are several possible ways to query this table. For example:

In[8]:

```
# Select all rows that have an age column greater than 30
display(data_pandas[data_pandas.Age > 30])
```

This produces the following result:

	Age	Location	Name
2	53	Berlin	Peter
3	33	London	Linda

mglearn

This book comes with accompanying code, which you can find on [GitHub](#). The accompanying code includes not only all the examples shown in this book, but also the `mglearn` library. This is a library of utility functions we wrote for this book, so that we don't clutter up our code listings with details of plotting and data loading. If you're interested, you can look up all the functions in the repository, but the details of the `mglearn` module are not really important to the material in this book. If you see a call to `mglearn` in the code, it is usually a way to make a pretty picture quickly, or to get our hands on some interesting data.



Throughout the book we make ample use of NumPy, `matplotlib` and `pandas`. All the code will assume the following imports:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import mglearn
```

We also assume that you will run the code in a Jupyter Notebook with the `%matplotlib notebook` or `%matplotlib inline` magic enabled to show plots. If you are not using the notebook or these magic commands, you will have to call `plt.show` to actually show any of the figures.

Python 2 Versus Python 3

There are two major versions of Python that are widely used at the moment: Python 2 (more precisely, 2.7) and Python 3 (with the latest release being 3.5 at the time of writing). This sometimes leads to some confusion. Python 2 is no longer actively developed, but because Python 3 contains major changes, Python 2 code usually does not run on Python 3. If you are new to Python, or are starting a new project from scratch, we highly recommend using the latest version of Python 3 without changes. If you have a large codebase that you rely on that is written for Python 2, you are excused from upgrading for now. However, you should try to migrate to Python 3 as soon as possible. When writing any new code, it is for the most part quite easy to write code that runs under Python 2 and Python 3.² If you don't have to interface with legacy software, you should definitely use Python 3. All the code in this book is written in a way that works for both versions. However, the exact output might differ slightly under Python 2.

Versions Used in this Book

We are using the following versions of the previously mentioned libraries in this book:

In[9]:

```
import sys
print("Python version: {}".format(sys.version))

import pandas as pd
print("pandas version: {}".format(pd.__version__))

import matplotlib
print("matplotlib version: {}".format(matplotlib.__version__))

import numpy as np
print("NumPy version: {}".format(np.__version__))

import scipy as sp
print("SciPy version: {}".format(sp.__version__))

import IPython
print("IPython version: {}".format(IPython.__version__))

import sklearn
print("scikit-learn version: {}".format(sklearn.__version__))
```

² The [six package](#) can be very handy for that.

Out[9]:

```
Python version: 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul 2 2016, 17:53:06)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
pandas version: 0.18.1
matplotlib version: 1.5.1
NumPy version: 1.11.1
SciPy version: 0.17.1
IPython version: 5.1.0
scikit-learn version: 0.18
```

While it is not important to match these versions exactly, you should have a version of `scikit-learn` that is as least as recent as the one we used.

Now that we have everything set up, let's dive into our first application of machine learning.



This book assumes that you have version 0.18 or later of scikit-learn. The `model_selection` module was added in 0.18, and if you use an earlier version of scikit-learn, you will need to adjust the imports from this module.

A First Application: Classifying Iris Species

In this section, we will go through a simple machine learning application and create our first model. In the process, we will introduce some core concepts and terms.

Let's assume that a hobby botanist is interested in distinguishing the species of some iris flowers that she has found. She has collected some measurements associated with each iris: the length and width of the petals and the length and width of the sepals, all measured in centimeters (see [Figure 1-2](#)).

She also has the measurements of some irises that have been previously identified by an expert botanist as belonging to the species *setosa*, *versicolor*, or *virginica*. For these measurements, she can be certain of which species each iris belongs to. Let's assume that these are the only species our hobby botanist will encounter in the wild.

Our goal is to build a machine learning model that can learn from the measurements of these irises whose species is known, so that we can predict the species for a new iris.

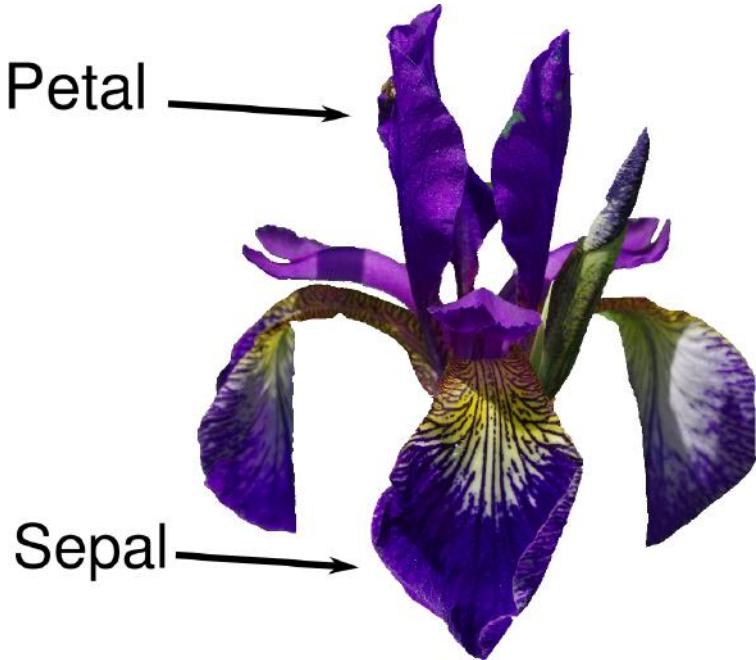


Figure 1-2. Parts of the iris flower

Because we have measurements for which we know the correct species of iris, this is a supervised learning problem. In this problem, we want to predict one of several options (the species of iris). This is an example of a *classification* problem. The possible outputs (different species of irises) are called *classes*. Every iris in the dataset belongs to one of three classes, so this problem is a three-class classification problem.

The desired output for a single data point (an iris) is the species of this flower. For a particular data point, the species it belongs to is called its *label*.

Meet the Data

The data we will use for this example is the Iris dataset, a classical dataset in machine learning and statistics. It is included in `scikit-learn` in the `datasets` module. We can load it by calling the `load_iris` function:

In[10]:

```
from sklearn.datasets import load_iris  
iris_dataset = load_iris()
```

The `iris` object that is returned by `load_iris` is a `Bunch` object, which is very similar to a dictionary. It contains keys and values:

In[11]:

```
print("Keys of iris_dataset: \n{}".format(iris_dataset.keys()))
```

Out[11]:

```
Keys of iris_dataset:  
dict_keys(['target_names', 'feature_names', 'DESCR', 'data', 'target'])
```

The value of the key DESCR is a short description of the dataset. We show the beginning of the description here (feel free to look up the rest yourself):

In[12]:

```
print(iris_dataset['DESCR'][:193] + "\n...")
```

Out[12]:

```
Iris Plants Database  
=====
```

Notes

```
----
```

Data Set Characteristics:

```
:Number of Instances: 150 (50 in each of three classes)  
:Number of Attributes: 4 numeric, predictive att
```

```
...  
----
```

The value of the key target_names is an array of strings, containing the species of flower that we want to predict:

In[13]:

```
print("Target names: {}".format(iris_dataset['target_names']))
```

Out[13]:

```
Target names: ['setosa' 'versicolor' 'virginica']
```

The value of feature_names is a list of strings, giving the description of each feature:

In[14]:

```
print("Feature names: \n{}".format(iris_dataset['feature_names']))
```

Out[14]:

```
Feature names:  
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',  
'petal width (cm)']
```

The data itself is contained in the target and data fields. data contains the numeric measurements of sepal length, sepal width, petal length, and petal width in a NumPy array:

In[15]:

```
print("Type of data: {}".format(type(iris_dataset['data'])))
```

Out[15]:

```
Type of data: <class 'numpy.ndarray'>
```

The rows in the `data` array correspond to flowers, while the columns represent the four measurements that were taken for each flower:

In[16]:

```
print("Shape of data: {}".format(iris_dataset['data'].shape))
```

Out[16]:

```
Shape of data: (150, 4)
```

We see that the array contains measurements for 150 different flowers. Remember that the individual items are called *samples* in machine learning, and their properties are called *features*. The *shape* of the `data` array is the number of samples multiplied by the number of features. This is a convention in `scikit-learn`, and your data will always be assumed to be in this shape. Here are the feature values for the first five samples:

In[17]:

```
print("First five columns of data:\n{}".format(iris_dataset['data'][:5]))
```

Out[17]:

```
First five columns of data:  
[[ 5.1  3.5  1.4  0.2]  
 [ 4.9  3.   1.4  0.2]  
 [ 4.7  3.2  1.3  0.2]  
 [ 4.6  3.1  1.5  0.2]  
 [ 5.   3.6  1.4  0.2]]
```

From this data, we can see that all of the first five flowers have a petal width of 0.2 cm and that the first flower has the longest sepal, at 5.1 cm.

The `target` array contains the species of each of the flowers that were measured, also as a NumPy array:

In[18]:

```
print("Type of target: {}".format(type(iris_dataset['target'])))
```

Out[18]:

```
Type of target: <class 'numpy.ndarray'>
```

`target` is a one-dimensional array, with one entry per flower:

In[19]:

```
print("Shape of target: {}".format(iris_dataset['target'].shape))
```

Out[19]:

```
Shape of target: (150,)
```

The species are encoded as integers from 0 to 2:

In[20]:

```
print("Target:\n{}".format(iris_dataset['target']))
```

Out[20]:

```
Target:  
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 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 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2  
 2 2]
```

The meanings of the numbers are given by the `iris['target_names']` array: 0 means *setosa*, 1 means *versicolor*, and 2 means *virginica*.

Measuring Success: Training and Testing Data

We want to build a machine learning model from this data that can predict the species of iris for a new set of measurements. But before we can apply our model to new measurements, we need to know whether it actually works—that is, whether we should trust its predictions.

Unfortunately, we cannot use the data we used to build the model to evaluate it. This is because our model can always simply remember the whole training set, and will therefore always predict the correct label for any point in the training set. This “remembering” does not indicate to us whether our model will *generalize* well (in other words, whether it will also perform well on new data).

To assess the model’s performance, we show it new data (data that it hasn’t seen before) for which we have labels. This is usually done by splitting the labeled data we have collected (here, our 150 flower measurements) into two parts. One part of the data is used to build our machine learning model, and is called the *training data* or *training set*. The rest of the data will be used to assess how well the model works; this is called the *test data*, *test set*, or *hold-out set*.

`scikit-learn` contains a function that shuffles the dataset and splits it for you: the `train_test_split` function. This function extracts 75% of the rows in the data as the training set, together with the corresponding labels for this data. The remaining 25% of the data, together with the remaining labels, is declared as the test set. Deciding

how much data you want to put into the training and the test set respectively is somewhat arbitrary, but using a test set containing 25% of the data is a good rule of thumb.

In `scikit-learn`, data is usually denoted with a capital `X`, while labels are denoted by a lowercase `y`. This is inspired by the standard formulation $f(x)=y$ in mathematics, where x is the input to a function and y is the output. Following more conventions from mathematics, we use a capital `X` because the data is a two-dimensional array (a matrix) and a lowercase `y` because the target is a one-dimensional array (a vector).

Let's call `train_test_split` on our data and assign the outputs using this nomenclature:

In[21]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    iris_dataset['data'], iris_dataset['target'], random_state=0)
```

Before making the split, the `train_test_split` function shuffles the dataset using a pseudorandom number generator. If we just took the last 25% of the data as a test set, all the data points would have the label 2, as the data points are sorted by the label (see the output for `iris['target']` shown earlier). Using a test set containing only one of the three classes would not tell us much about how well our model generalizes, so we shuffle our data to make sure the test data contains data from all classes.

To make sure that we will get the same output if we run the same function several times, we provide the pseudorandom number generator with a fixed seed using the `random_state` parameter. This will make the outcome deterministic, so this line will always have the same outcome. We will always fix the `random_state` in this way when using randomized procedures in this book.

The output of the `train_test_split` function is `X_train`, `X_test`, `y_train`, and `y_test`, which are all NumPy arrays. `X_train` contains 75% of the rows of the dataset, and `X_test` contains the remaining 25%:

In[22]:

```
print("X_train shape: {}".format(X_train.shape))
print("y_train shape: {}".format(y_train.shape))
```

Out[22]:

```
X_train shape: (112, 4)
y_train shape: (112,)
```

In[23]:

```
print("X_test shape: {}".format(X_test.shape))
print("y_test shape: {}".format(y_test.shape))
```

Out[23]:

```
X_test shape: (38, 4)
y_test shape: (38,)
```

First Things First: Look at Your Data

Before building a machine learning model it is often a good idea to inspect the data, to see if the task is easily solvable without machine learning, or if the desired information might not be contained in the data.

Additionally, inspecting your data is a good way to find abnormalities and peculiarities. Maybe some of your irises were measured using inches and not centimeters, for example. In the real world, inconsistencies in the data and unexpected measurements are very common.

One of the best ways to inspect data is to visualize it. One way to do this is by using a *scatter plot*. A scatter plot of the data puts one feature along the x-axis and another along the y-axis, and draws a dot for each data point. Unfortunately, computer screens have only two dimensions, which allows us to plot only two (or maybe three) features at a time. It is difficult to plot datasets with more than three features this way. One way around this problem is to do a *pair plot*, which looks at all possible pairs of features. If you have a small number of features, such as the four we have here, this is quite reasonable. You should keep in mind, however, that a pair plot does not show the interaction of all of features at once, so some interesting aspects of the data may not be revealed when visualizing it this way.

Figure 1-3 is a pair plot of the features in the training set. The data points are colored according to the species the iris belongs to. To create the plot, we first convert the NumPy array into a pandas DataFrame. pandas has a function to create pair plots called `scatter_matrix`. The diagonal of this matrix is filled with histograms of each feature:

In[24]:

```
# create dataframe from data in X_train
# label the columns using the strings in iris_dataset.feature_names
iris_dataframe = pd.DataFrame(X_train, columns=iris_dataset.feature_names)
# create a scatter matrix from the dataframe, color by y_train
grr = pd.scatter_matrix(iris_dataframe, c=y_train, figsize=(15, 15), marker='o',
                        hist_kwds={'bins': 20}, s=60, alpha=.8, cmap=mglearn.cm3)
```

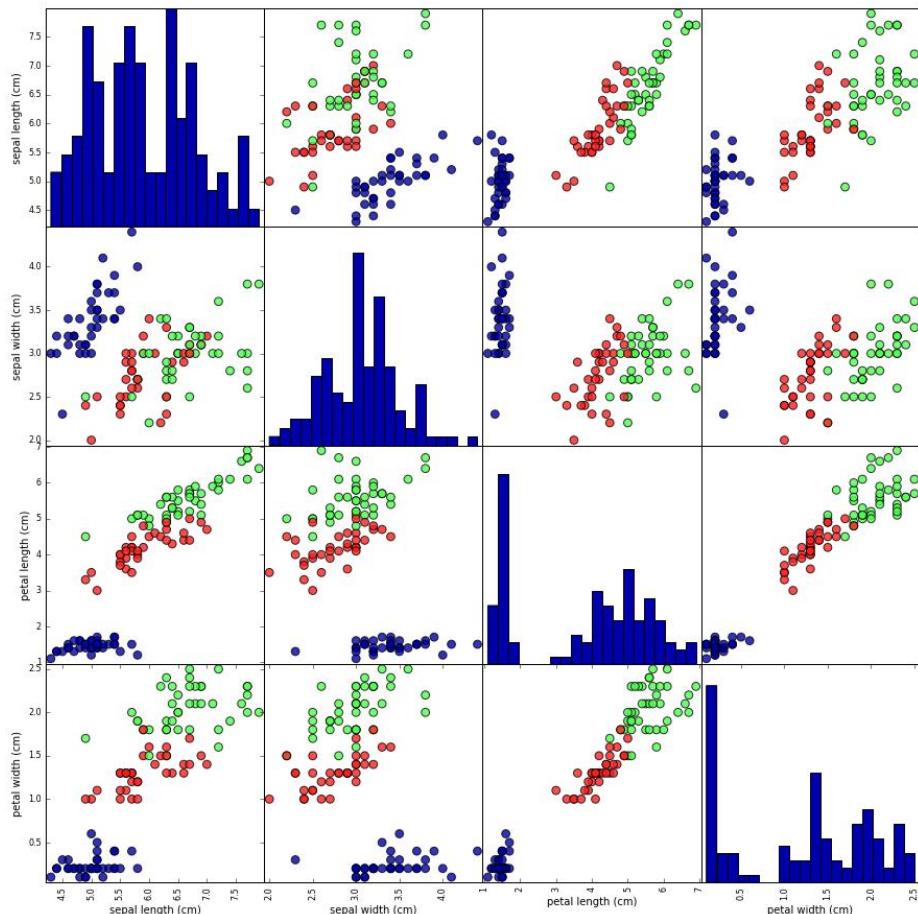


Figure 1-3. Pair plot of the Iris dataset, colored by class label

From the plots, we can see that the three classes seem to be relatively well separated using the sepal and petal measurements. This means that a machine learning model will likely be able to learn to separate them.

Building Your First Model: k-Nearest Neighbors

Now we can start building the actual machine learning model. There are many classification algorithms in `scikit-learn` that we could use. Here we will use a k -nearest neighbors classifier, which is easy to understand. Building this model only consists of storing the training set. To make a prediction for a new data point, the algorithm finds the point in the training set that is closest to the new point. Then it assigns the label of this training point to the new data point.

The k in k -nearest neighbors signifies that instead of using only the closest neighbor to the new data point, we can consider any fixed number k of neighbors in the training (for example, the closest three or five neighbors). Then, we can make a prediction using the majority class among these neighbors. We will go into more detail about this in [Chapter 2](#); for now, we'll use only a single neighbor.

All machine learning models in `scikit-learn` are implemented in their own classes, which are called `Estimator` classes. The k -nearest neighbors classification algorithm is implemented in the `KNeighborsClassifier` class in the `neighbors` module. Before we can use the model, we need to instantiate the class into an object. This is when we will set any parameters of the model. The most important parameter of `KNeighborsClassifier` is the number of neighbors, which we will set to 1:

In[25]:

```
from sklearn.neighbors import KNeighborsClassifier  
knn = KNeighborsClassifier(n_neighbors=1)
```

The `knn` object encapsulates the algorithm that will be used to build the model from the training data, as well the algorithm to make predictions on new data points. It will also hold the information that the algorithm has extracted from the training data. In the case of `KNeighborsClassifier`, it will just store the training set.

To build the model on the training set, we call the `fit` method of the `knn` object, which takes as arguments the NumPy array `X_train` containing the training data and the NumPy array `y_train` of the corresponding training labels:

In[26]:

```
knn.fit(X_train, y_train)
```

Out[26]:

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',  
                     metric_params=None, n_jobs=1, n_neighbors=1, p=2,  
                     weights='uniform')
```

The `fit` method returns the `knn` object itself (and modifies it in place), so we get a string representation of our classifier. The representation shows us which parameters were used in creating the model. Nearly all of them are the default values, but you can also find `n_neighbors=1`, which is the parameter that we passed. Most models in `scikit-learn` have many parameters, but the majority of them are either speed optimizations or for very special use cases. You don't have to worry about the other parameters shown in this representation. Printing a `scikit-learn` model can yield very long strings, but don't be intimidated by these. We will cover all the important parameters in [Chapter 2](#). In the remainder of this book, we will not show the output of `fit` because it doesn't contain any new information.

Making Predictions

We can now make predictions using this model on new data for which we might not know the correct labels. Imagine we found an iris in the wild with a sepal length of 5 cm, a sepal width of 2.9 cm, a petal length of 1 cm, and a petal width of 0.2 cm. What species of iris would this be? We can put this data into a NumPy array, again by calculating the shape—that is, the number of samples (1) multiplied by the number of features (4):

In[27]:

```
X_new = np.array([[5, 2.9, 1, 0.2]])
print("X_new.shape: {}".format(X_new.shape))
```

Out[27]:

```
X_new.shape: (1, 4)
```

Note that we made the measurements of this single flower into a row in a two-dimensional NumPy array, as scikit-learn always expects two-dimensional arrays for the data.

To make a prediction, we call the `predict` method of the `knn` object:

In[28]:

```
prediction = knn.predict(X_new)
print("Prediction: {}".format(prediction))
print("Predicted target name: {}".format(
    iris_dataset['target_names'][prediction]))
```

Out[28]:

```
Prediction: [0]
Predicted target name: ['setosa']
```

Our model predicts that this new iris belongs to the class 0, meaning its species is *setosa*. But how do we know whether we can trust our model? We don't know the correct species of this sample, which is the whole point of building the model!

Evaluating the Model

This is where the test set that we created earlier comes in. This data was not used to build the model, but we do know what the correct species is for each iris in the test set.

Therefore, we can make a prediction for each iris in the test data and compare it against its label (the known species). We can measure how well the model works by computing the *accuracy*, which is the fraction of flowers for which the right species was predicted:

In[29]:

```
y_pred = knn.predict(X_test)
print("Test set predictions:\n {}".format(y_pred))
```

Out[29]:

```
Test set predictions:
 [2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2 1 0 2 2 1 0 2]
```

In[30]:

```
print("Test set score: {:.2f}".format(np.mean(y_pred == y_test)))
```

Out[30]:

```
Test set score: 0.97
```

We can also use the `score` method of the `knn` object, which will compute the test set accuracy for us:

In[31]:

```
print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))
```

Out[31]:

```
Test set score: 0.97
```

For this model, the test set accuracy is about 0.97, which means we made the right prediction for 97% of the irises in the test set. Under some mathematical assumptions, this means that we can expect our model to be correct 97% of the time for new irises. For our hobby botanist application, this high level of accuracy means that our model may be trustworthy enough to use. In later chapters we will discuss how we can improve performance, and what caveats there are in tuning a model.

Summary and Outlook

Let's summarize what we learned in this chapter. We started with a brief introduction to machine learning and its applications, then discussed the distinction between supervised and unsupervised learning and gave an overview of the tools we'll be using in this book. Then, we formulated the task of predicting which species of iris a particular flower belongs to by using physical measurements of the flower. We used a dataset of measurements that was annotated by an expert with the correct species to build our model, making this a supervised learning task. There were three possible species, *setosa*, *versicolor*, or *virginica*, which made the task a three-class classification problem. The possible species are called *classes* in the classification problem, and the species of a single iris is called its *label*.

The Iris dataset consists of two NumPy arrays: one containing the data, which is referred to as `X` in `scikit-learn`, and one containing the correct or desired outputs,

which is called `y`. The array `X` is a two-dimensional array of features, with one row per data point and one column per feature. The array `y` is a one-dimensional array, which here contains one class label, an integer ranging from 0 to 2, for each of the samples.

We split our dataset into a *training set*, to build our model, and a *test set*, to evaluate how well our model will generalize to new, previously unseen data.

We chose the *k*-nearest neighbors classification algorithm, which makes predictions for a new data point by considering its closest neighbor(s) in the training set. This is implemented in the `KNeighborsClassifier` class, which contains the algorithm that builds the model as well as the algorithm that makes a prediction using the model. We instantiated the class, setting parameters. Then we built the model by calling the `fit` method, passing the training data (`X_train`) and training outputs (`y_train`) as parameters. We evaluated the model using the `score` method, which computes the accuracy of the model. We applied the `score` method to the test set data and the test set labels and found that our model is about 97% accurate, meaning it is correct 97% of the time on the test set.

This gave us the confidence to apply the model to new data (in our example, new flower measurements) and trust that the model will be correct about 97% of the time.

Here is a summary of the code needed for the whole training and evaluation procedure:

In[32]:

```
X_train, X_test, y_train, y_test = train_test_split(  
    iris_dataset['data'], iris_dataset['target'], random_state=0)  
  
knn = KNeighborsClassifier(n_neighbors=1)  
knn.fit(X_train, y_train)  
  
print("Test set score: {:.2f}".format(knn.score(X_test, y_test)))
```

Out[32]:

```
Test set score: 0.97
```

This snippet contains the core code for applying any machine learning algorithm using `scikit-learn`. The `fit`, `predict`, and `score` methods are the common interface to supervised models in `scikit-learn`, and with the concepts introduced in this chapter, you can apply these models to many machine learning tasks. In the next chapter, we will go into more depth about the different kinds of supervised models in `scikit-learn` and how to apply them successfully.

CHAPTER 2

Supervised Learning

As we mentioned earlier, supervised machine learning is one of the most commonly used and successful types of machine learning. In this chapter, we will describe supervised learning in more detail and explain several popular supervised learning algorithms. We already saw an application of supervised machine learning in [Chapter 1](#): classifying iris flowers into several species using physical measurements of the flowers.

Remember that supervised learning is used whenever we want to predict a certain outcome from a given input, and we have examples of input/output pairs. We build a machine learning model from these input/output pairs, which comprise our training set. Our goal is to make accurate predictions for new, never-before-seen data. Supervised learning often requires human effort to build the training set, but afterward automates and often speeds up an otherwise laborious or infeasible task.

Classification and Regression

There are two major types of supervised machine learning problems, called *classification* and *regression*.

In classification, the goal is to predict a *class label*, which is a choice from a predefined list of possibilities. In [Chapter 1](#) we used the example of classifying irises into one of three possible species. Classification is sometimes separated into *binary classification*, which is the special case of distinguishing between exactly two classes, and *multiclass classification*, which is classification between more than two classes. You can think of binary classification as trying to answer a yes/no question. Classifying emails as either spam or not spam is an example of a binary classification problem. In this binary classification task, the yes/no question being asked would be “Is this email spam?”



In binary classification we often speak of one class being the *positive* class and the other class being the *negative* class. Here, positive doesn't represent having benefit or value, but rather what the object of the study is. So, when looking for spam, "positive" could mean the spam class. Which of the two classes is called positive is often a subjective matter, and specific to the domain.

The iris example, on the other hand, is an example of a multiclass classification problem. Another example is predicting what language a website is in from the text on the website. The classes here would be a pre-defined list of possible languages.

For regression tasks, the goal is to predict a continuous number, or a *floating-point number* in programming terms (or *real number* in mathematical terms). Predicting a person's annual income from their education, their age, and where they live is an example of a regression task. When predicting income, the predicted value is an *amount*, and can be any number in a given range. Another example of a regression task is predicting the yield of a corn farm given attributes such as previous yields, weather, and number of employees working on the farm. The yield again can be an arbitrary number.

An easy way to distinguish between classification and regression tasks is to ask whether there is some kind of continuity in the output. If there is continuity between possible outcomes, then the problem is a regression problem. Think about predicting annual income. There is a clear continuity in the output. Whether a person makes \$40,000 or \$40,001 a year does not make a tangible difference, even though these are different amounts of money; if our algorithm predicts \$39,999 or \$40,001 when it should have predicted \$40,000, we don't mind that much.

By contrast, for the task of recognizing the language of a website (which is a classification problem), there is no matter of degree. A website is in one language, or it is in another. There is no continuity between languages, and there is no language that is *between* English and French.¹

Generalization, Overfitting, and Underfitting

In supervised learning, we want to build a model on the training data and then be able to make accurate predictions on new, unseen data that has the same characteristics as the training set that we used. If a model is able to make accurate predictions on unseen data, we say it is able to *generalize* from the training set to the test set. We want to build a model that is able to generalize as accurately as possible.

¹ We ask linguists to excuse the simplified presentation of languages as distinct and fixed entities.

Usually we build a model in such a way that it can make accurate predictions on the training set. If the training and test sets have enough in common, we expect the model to also be accurate on the test set. However, there are some cases where this can go wrong. For example, if we allow ourselves to build very complex models, we can always be as accurate as we like on the training set.

Let's take a look at a made-up example to illustrate this point. Say a novice data scientist wants to predict whether a customer will buy a boat, given records of previous boat buyers and customers who we know are not interested in buying a boat.² The goal is to send out promotional emails to people who are likely to actually make a purchase, but not bother those customers who won't be interested.

Suppose we have the customer records shown in [Table 2-1](#).

Table 2-1. Example data about customers

Age	Number of cars owned	Owns house	Number of children	Marital status	Owns a dog	Bought a boat
66	1	yes	2	widowed	no	yes
52	2	yes	3	married	no	yes
22	0	no	0	married	yes	no
25	1	no	1	single	no	no
44	0	no	2	divorced	yes	no
39	1	yes	2	married	yes	no
26	1	no	2	single	no	no
40	3	yes	1	married	yes	no
53	2	yes	2	divorced	no	yes
64	2	yes	3	divorced	no	no
58	2	yes	2	married	yes	yes
33	1	no	1	single	no	no

After looking at the data for a while, our novice data scientist comes up with the following rule: “If the customer is older than 45, and has less than 3 children or is not divorced, then they want to buy a boat.” When asked how well this rule of his does, our data scientist answers, “It’s 100 percent accurate!” And indeed, on the data that is in the table, the rule is perfectly accurate. There are many possible rules we could come up with that would explain perfectly if someone in this dataset wants to buy a boat. No age appears twice in the data, so we could say people who are 66, 52, 53, or

² In the real world, this is actually a tricky problem. While we know that the other customers haven’t bought a boat from us yet, they might have bought one from someone else, or they may still be saving and plan to buy one in the future.

58 years old want to buy a boat, while all others don't. While we can make up many rules that work well on this data, remember that we are not interested in making predictions for this dataset; we already know the answers for these customers. We want to know if *new customers* are likely to buy a boat. We therefore want to find a rule that will work well for new customers, and achieving 100 percent accuracy on the training set does not help us there. We might not expect that the rule our data scientist came up with will work very well on new customers. It seems too complex, and it is supported by very little data. For example, the "or is not divorced" part of the rule hinges on a single customer.

The only measure of whether an algorithm will perform well on new data is the evaluation on the test set. However, intuitively³ we expect simple models to generalize better to new data. If the rule was "People older than 50 want to buy a boat," and this would explain the behavior of all the customers, we would trust it more than the rule involving children and marital status in addition to age. Therefore, we always want to find the simplest model. Building a model that is too complex for the amount of information we have, as our novice data scientist did, is called *overfitting*. Overfitting occurs when you fit a model too closely to the particularities of the training set and obtain a model that works well on the training set but is not able to generalize to new data. On the other hand, if your model is too simple—say, "Everybody who owns a house buys a boat"—then you might not be able to capture all the aspects of and variability in the data, and your model will do badly even on the training set. Choosing too simple a model is called *underfitting*.

The more complex we allow our model to be, the better we will be able to predict on the training data. However, if our model becomes too complex, we start focusing too much on each individual data point in our training set, and the model will not generalize well to new data.

There is a sweet spot in between that will yield the best generalization performance. This is the model we want to find.

The trade-off between overfitting and underfitting is illustrated in [Figure 2-1](#).

³ And also provably, with the right math.

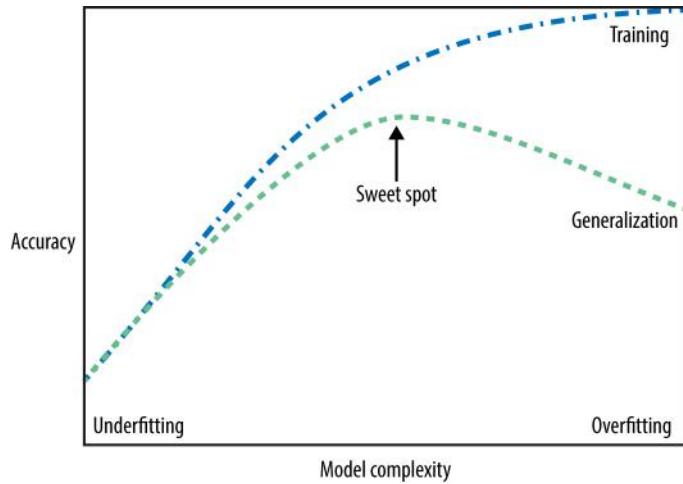


Figure 2-1. Trade-off of model complexity against training and test accuracy

Relation of Model Complexity to Dataset Size

It's important to note that model complexity is intimately tied to the variation of inputs contained in your training dataset: the larger variety of data points your dataset contains, the more complex a model you can use without overfitting. Usually, collecting more data points will yield more variety, so larger datasets allow building more complex models. However, simply duplicating the same data points or collecting very similar data will not help.

Going back to the boat selling example, if we saw 10,000 more rows of customer data, and all of them complied with the rule “If the customer is older than 45, and has less than 3 children or is not divorced, then they want to buy a boat,” we would be much more likely to believe this to be a good rule than when it was developed using only the 12 rows in [Table 2-1](#).

Having more data and building appropriately more complex models can often work wonders for supervised learning tasks. In this book, we will focus on working with datasets of fixed sizes. In the real world, you often have the ability to decide how much data to collect, which might be more beneficial than tweaking and tuning your model. Never underestimate the power of more data.

Supervised Machine Learning Algorithms

We will now review the most popular machine learning algorithms and explain how they learn from data and how they make predictions. We will also discuss how the concept of model complexity plays out for each of these models, and provide an over-

view of how each algorithm builds a model. We will examine the strengths and weaknesses of each algorithm, and what kind of data they can best be applied to. We will also explain the meaning of the most important parameters and options.⁴ Many algorithms have a classification and a regression variant, and we will describe both.

It is not necessary to read through the descriptions of each algorithm in detail, but understanding the models will give you a better feeling for the different ways machine learning algorithms can work. This chapter can also be used as a reference guide, and you can come back to it when you are unsure about the workings of any of the algorithms.

Some Sample Datasets

We will use several datasets to illustrate the different algorithms. Some of the datasets will be small and synthetic (meaning made-up), designed to highlight particular aspects of the algorithms. Other datasets will be large, real-world examples.

An example of a synthetic two-class classification dataset is the `forge` dataset, which has two features. The following code creates a scatter plot (Figure 2-2) visualizing all of the data points in this dataset. The plot has the first feature on the x-axis and the second feature on the y-axis. As is always the case in scatter plots, each data point is represented as one dot. The color and shape of the dot indicates its class:

In[2]:

```
# generate dataset
X, y = mglearn.datasets.make_forge()
# plot dataset
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.legend(["Class 0", "Class 1"], loc=4)
plt.xlabel("First feature")
plt.ylabel("Second feature")
print("X.shape: {}".format(X.shape))
```

Out[2]:

```
X.shape: (26, 2)
```

⁴ Discussing all of them is beyond the scope of the book, and we refer you to the [scikit-learn documentation](#) for more details.

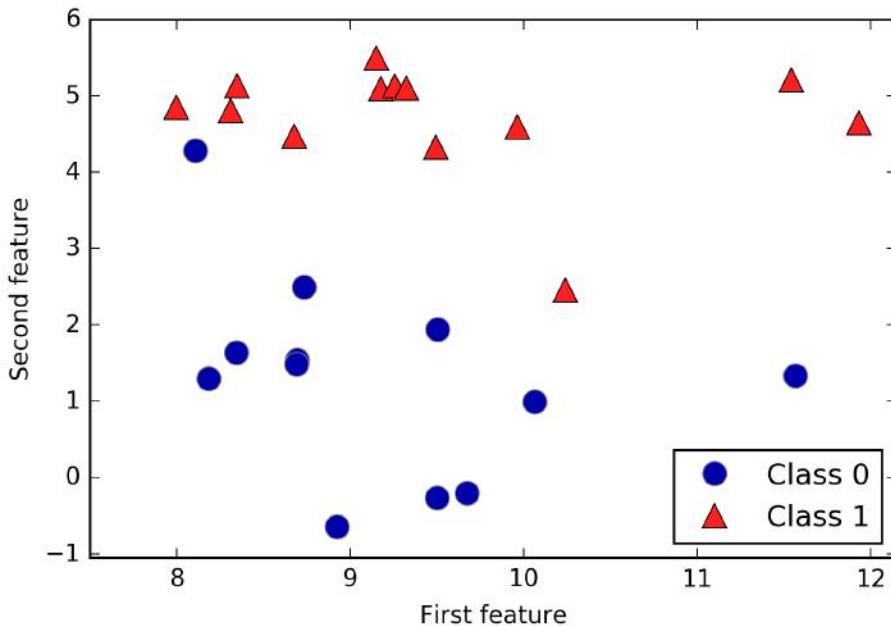


Figure 2-2. Scatter plot of the forge dataset

As you can see from `X.shape`, this dataset consists of 26 data points, with 2 features.

To illustrate regression algorithms, we will use the synthetic wave dataset. The wave dataset has a single input feature and a continuous target variable (or *response*) that we want to model. The plot created here (Figure 2-3) shows the single feature on the x-axis and the regression target (the output) on the y-axis:

In[3]:

```
X, y = mglearn.datasets.make_wave(n_samples=40)
plt.plot(X, y, 'o')
plt.ylim(-3, 3)
plt.xlabel("Feature")
plt.ylabel("Target")
```

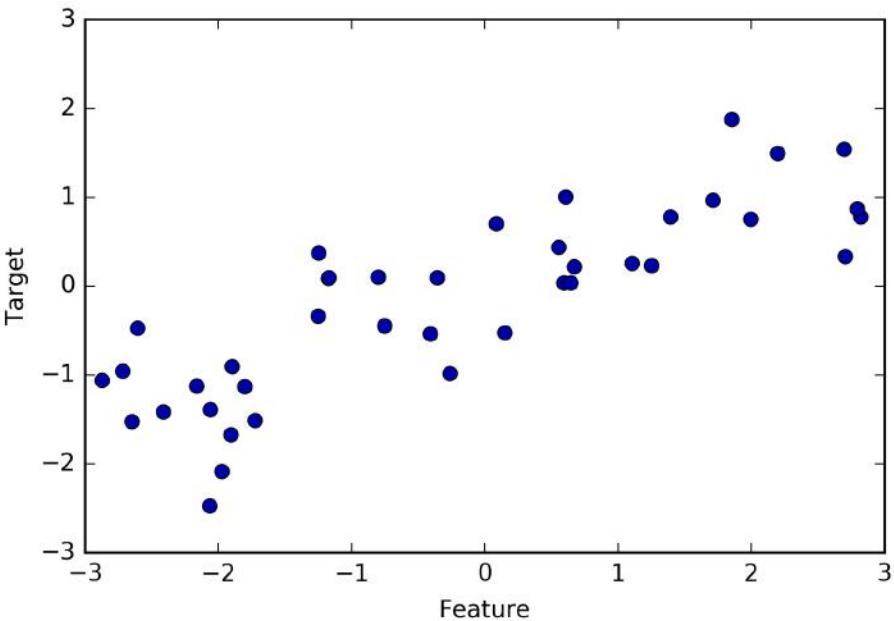


Figure 2-3. Plot of the wave dataset, with the x-axis showing the feature and the y-axis showing the regression target

We are using these very simple, low-dimensional datasets because we can easily visualize them—a printed page has two dimensions, so data with more than two features is hard to show. Any intuition derived from datasets with few features (also called *low-dimensional* datasets) might not hold in datasets with many features (*high-dimensional* datasets). As long as you keep that in mind, inspecting algorithms on low-dimensional datasets can be very instructive.

We will complement these small synthetic datasets with two real-world datasets that are included in `scikit-learn`. One is the Wisconsin Breast Cancer dataset (`cancer`, for short), which records clinical measurements of breast cancer tumors. Each tumor is labeled as “benign” (for harmless tumors) or “malignant” (for cancerous tumors), and the task is to learn to predict whether a tumor is malignant based on the measurements of the tissue.

The data can be loaded using the `load_breast_cancer` function from `scikit-learn`:

In[4]:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
print("cancer.keys(): \n{}".format(cancer.keys()))
```

Out[4]:

```
cancer.keys():
dict_keys(['feature_names', 'data', 'DESCR', 'target', 'target_names'])
```



Datasets that are included in `scikit-learn` are usually stored as `Bunch` objects, which contain some information about the dataset as well as the actual data. All you need to know about `Bunch` objects is that they behave like dictionaries, with the added benefit that you can access values using a dot (as in `bunch.key` instead of `bunch['key']`).

The dataset consists of 569 data points, with 30 features each:

In[5]:

```
print("Shape of cancer data: {}".format(cancer.data.shape))
```

Out[5]:

```
Shape of cancer data: (569, 30)
```

Of these 569 data points, 212 are labeled as malignant and 357 as benign:

In[6]:

```
print("Sample counts per class:\n{}".format(
    {n: v for n, v in zip(cancer.target_names, np.bincount(cancer.target))}))
```

Out[6]:

```
Sample counts per class:
{'benign': 357, 'malignant': 212}
```

To get a description of the semantic meaning of each feature, we can have a look at the `feature_names` attribute:

In[7]:

```
print("Feature names:\n{}".format(cancer.feature_names))
```

Out[7]:

```
Feature names:
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

You can find out more about the data by reading `cancer.DESCR` if you are interested.

We will also be using a real-world regression dataset, the Boston Housing dataset. The task associated with this dataset is to predict the median value of homes in several Boston neighborhoods in the 1970s, using information such as crime rate, proximity to the Charles River, highway accessibility, and so on. The dataset contains 506 data points, described by 13 features:

In[8]:

```
from sklearn.datasets import load_boston
boston = load_boston()
print("Data shape: {}".format(boston.data.shape))
```

Out[8]:

```
Data shape: (506, 13)
```

Again, you can get more information about the dataset by reading the `DESCR` attribute of `boston`. For our purposes here, we will actually expand this dataset by not only considering these 13 measurements as input features, but also looking at all products (also called *interactions*) between features. In other words, we will not only consider crime rate and highway accessibility as features, but also the product of crime rate and highway accessibility. Including derived feature like these is called *feature engineering*, which we will discuss in more detail in [Chapter 4](#). This derived dataset can be loaded using the `load_extended_boston` function:

In[9]:

```
X, y = mlearn.datasets.load_extended_boston()
print("X.shape: {}".format(X.shape))
```

Out[9]:

```
X.shape: (506, 104)
```

The resulting 104 features are the 13 original features together with the 91 possible combinations of two features within those 13.⁵

We will use these datasets to explain and illustrate the properties of the different machine learning algorithms. But for now, let's get to the algorithms themselves. First, we will revisit the *k*-nearest neighbors (*k*-NN) algorithm that we saw in the previous chapter.

⁵ This is called the binomial coefficient, which is the number of combinations of *k* elements that can be selected from a set of *n* elements. Often this is written as $\binom{n}{k}$ and spoken as “*n* choose *k*”—in this case, “13 choose 2.”

k-Nearest Neighbors

The k -NN algorithm is arguably the simplest machine learning algorithm. Building the model consists only of storing the training dataset. To make a prediction for a new data point, the algorithm finds the closest data points in the training dataset—its “nearest neighbors.”

k-Neighbors classification

In its simplest version, the k -NN algorithm only considers exactly one nearest neighbor, which is the closest training data point to the point we want to make a prediction for. The prediction is then simply the known output for this training point. [Figure 2-4](#) illustrates this for the case of classification on the `forge` dataset:

In[10]:

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

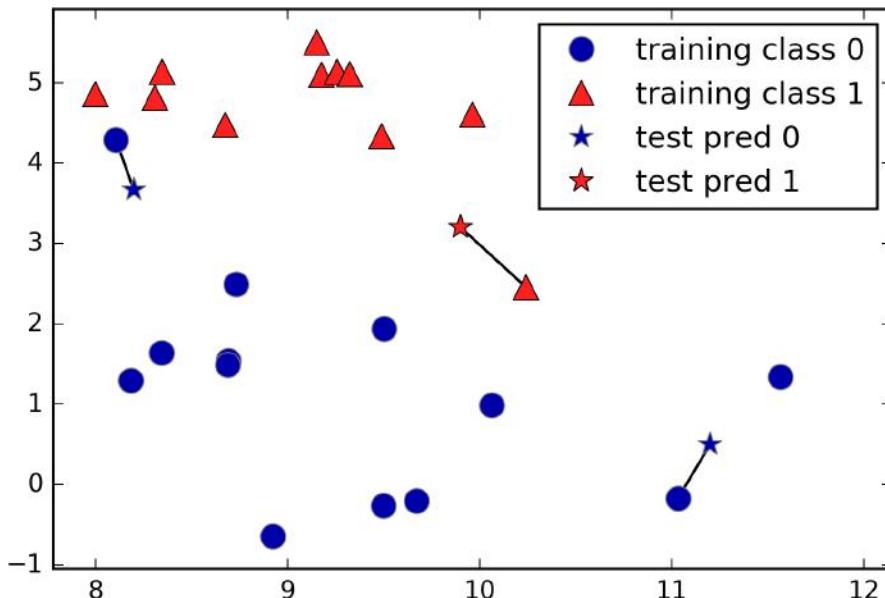


Figure 2-4. Predictions made by the one-nearest-neighbor model on the `forge` dataset

Here, we added three new data points, shown as stars. For each of them, we marked the closest point in the training set. The prediction of the one-nearest-neighbor algorithm is the label of that point (shown by the color of the cross).

Instead of considering only the closest neighbor, we can also consider an arbitrary number, k , of neighbors. This is where the name of the k -nearest neighbors algorithm comes from. When considering more than one neighbor, we use *voting* to assign a label. This means that for each test point, we count how many neighbors belong to class 0 and how many neighbors belong to class 1. We then assign the class that is more frequent: in other words, the majority class among the k -nearest neighbors. The following example ([Figure 2-5](#)) uses the three closest neighbors:

In[11]:

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

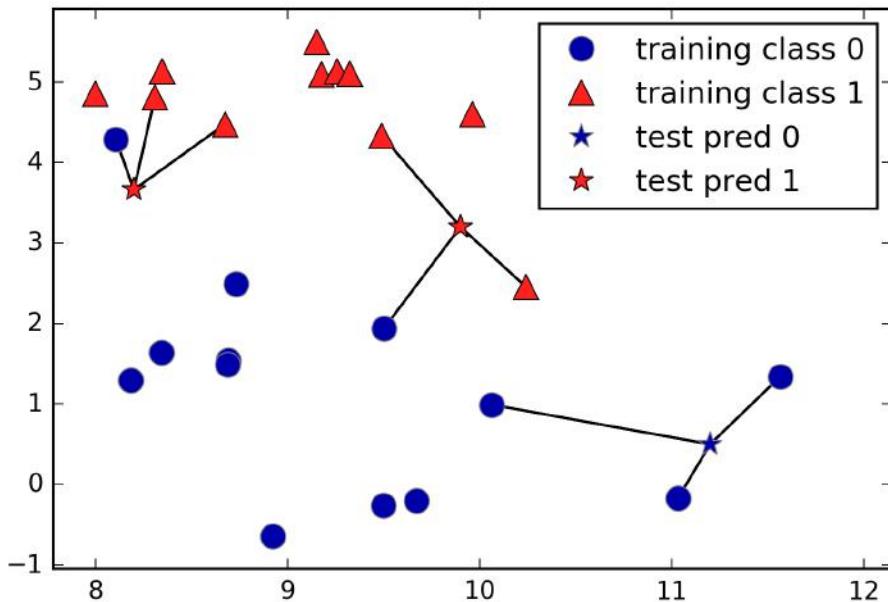


Figure 2-5. Predictions made by the three-nearest-neighbors model on the forge dataset

Again, the prediction is shown as the color of the cross. You can see that the prediction for the new data point at the top left is not the same as the prediction when we used only one neighbor.

While this illustration is for a binary classification problem, this method can be applied to datasets with any number of classes. For more classes, we count how many neighbors belong to each class and again predict the most common class.

Now let's look at how we can apply the k -nearest neighbors algorithm using `scikit-learn`. First, we split our data into a training and a test set so we can evaluate generalization performance, as discussed in [Chapter 1](#):

In[12]:

```
from sklearn.model_selection import train_test_split
X, y = mglearn.datasets.make_forge()

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Next, we import and instantiate the class. This is when we can set parameters, like the number of neighbors to use. Here, we set it to 3:

In[13]:

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=3)
```

Now, we fit the classifier using the training set. For `KNeighborsClassifier` this means storing the dataset, so we can compute neighbors during prediction:

In[14]:

```
clf.fit(X_train, y_train)
```

To make predictions on the test data, we call the `predict` method. For each data point in the test set, this computes its nearest neighbors in the training set and finds the most common class among these:

In[15]:

```
print("Test set predictions: {}".format(clf.predict(X_test)))
```

Out[15]:

```
Test set predictions: [1 0 1 0 1 0 0]
```

To evaluate how well our model generalizes, we can call the `score` method with the test data together with the test labels:

In[16]:

```
print("Test set accuracy: {:.2f}".format(clf.score(X_test, y_test)))
```

Out[16]:

```
Test set accuracy: 0.86
```

We see that our model is about 86% accurate, meaning the model predicted the class correctly for 86% of the samples in the test dataset.

Analyzing `KNeighborsClassifier`

For two-dimensional datasets, we can also illustrate the prediction for all possible test points in the xy -plane. We color the plane according to the class that would be assigned to a point in this region. This lets us view the *decision boundary*, which is the divide between where the algorithm assigns class 0 versus where it assigns class 1.

The following code produces the visualizations of the decision boundaries for one, three, and nine neighbors shown in [Figure 2-6](#):

In[17]:

```
fig, axes = plt.subplots(1, 3, figsize=(10, 3))

for n_neighbors, ax in zip([1, 3, 9], axes):
    # the fit method returns the object self, so we can instantiate
    # and fit in one line
    clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=True, eps=0.5, ax=ax, alpha=.4)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{} neighbor(s)".format(n_neighbors))
    ax.set_xlabel("feature 0")
    ax.set_ylabel("feature 1")
    axes[0].legend(loc=3)
```

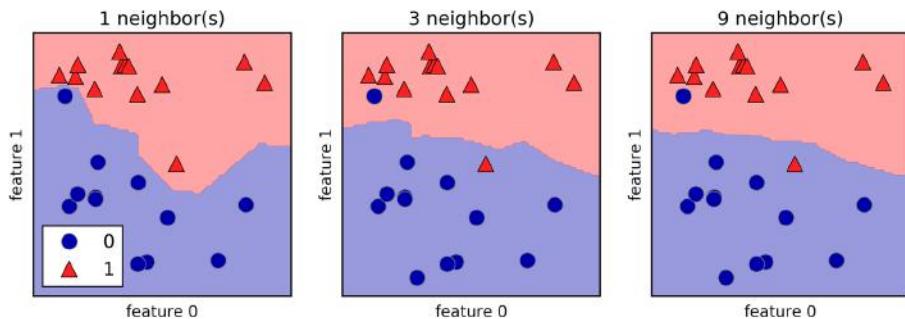


Figure 2-6. Decision boundaries created by the nearest neighbors model for different values of n_neighbors

As you can see on the left in the figure, using a single neighbor results in a decision boundary that follows the training data closely. Considering more and more neighbors leads to a smoother decision boundary. A smoother boundary corresponds to a simpler model. In other words, using few neighbors corresponds to high model complexity (as shown on the right side of [Figure 2-1](#)), and using many neighbors corresponds to low model complexity (as shown on the left side of [Figure 2-1](#)). If you consider the extreme case where the number of neighbors is the number of all data points in the training set, each test point would have exactly the same neighbors (all training points) and all predictions would be the same: the class that is most frequent in the training set.

Let's investigate whether we can confirm the connection between model complexity and generalization that we discussed earlier. We will do this on the real-world Breast Cancer dataset. We begin by splitting the dataset into a training and a test set. Then

we evaluate training and test set performance with different numbers of neighbors. The results are shown in [Figure 2-7](#):

In[18]:

```
from sklearn.datasets import load_breast_cancer

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=66)

training_accuracy = []
test_accuracy = []
# try n_neighbors from 1 to 10
neighbors_settings = range(1, 11)

for n_neighbors in neighbors_settings:
    # build the model
    clf = KNeighborsClassifier(n_neighbors=n_neighbors)
    clf.fit(X_train, y_train)
    # record training set accuracy
    training_accuracy.append(clf.score(X_train, y_train))
    # record generalization accuracy
    test_accuracy.append(clf.score(X_test, y_test))

plt.plot(neighbors_settings, training_accuracy, label="training accuracy")
plt.plot(neighbors_settings, test_accuracy, label="test accuracy")
plt.ylabel("Accuracy")
plt.xlabel("n_neighbors")
plt.legend()
```

The plot shows the training and test set accuracy on the y-axis against the setting of `n_neighbors` on the x-axis. While real-world plots are rarely very smooth, we can still recognize some of the characteristics of overfitting and underfitting (note that because considering fewer neighbors corresponds to a more complex model, the plot is horizontally flipped relative to the illustration in [Figure 2-1](#)). Considering a single nearest neighbor, the prediction on the training set is perfect. But when more neighbors are considered, the model becomes simpler and the training accuracy drops. The test set accuracy for using a single neighbor is lower than when using more neighbors, indicating that using the single nearest neighbor leads to a model that is too complex. On the other hand, when considering 10 neighbors, the model is too simple and performance is even worse. The best performance is somewhere in the middle, using around six neighbors. Still, it is good to keep the scale of the plot in mind. The worst performance is around 88% accuracy, which might still be acceptable.

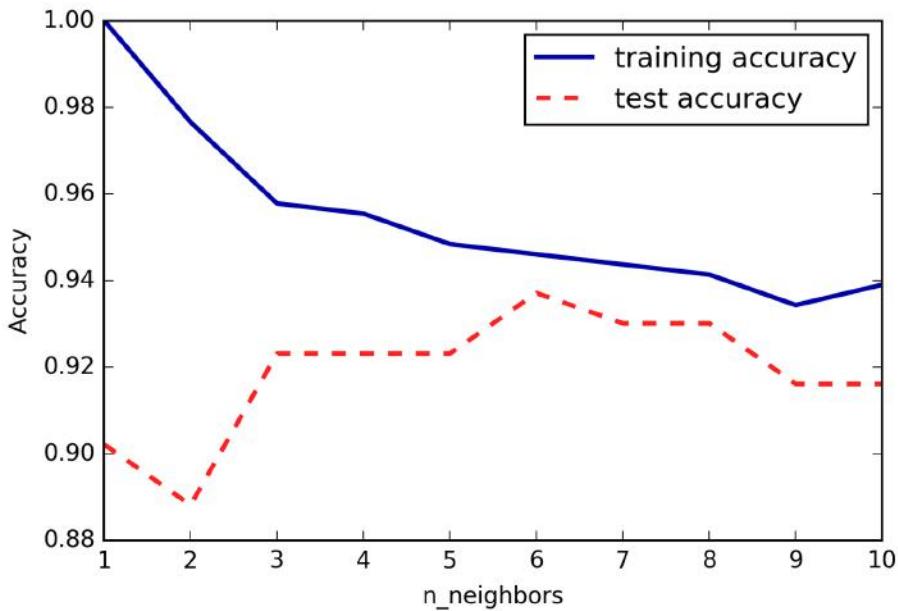


Figure 2-7. Comparison of training and test accuracy as a function of $n_{\text{neighbors}}$

k-neighbors regression

There is also a regression variant of the k -nearest neighbors algorithm. Again, let's start by using the single nearest neighbor, this time using the wave dataset. We've added three test data points as green stars on the x-axis. The prediction using a single neighbor is just the target value of the nearest neighbor. These are shown as blue stars in Figure 2-8:

In[19]:

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

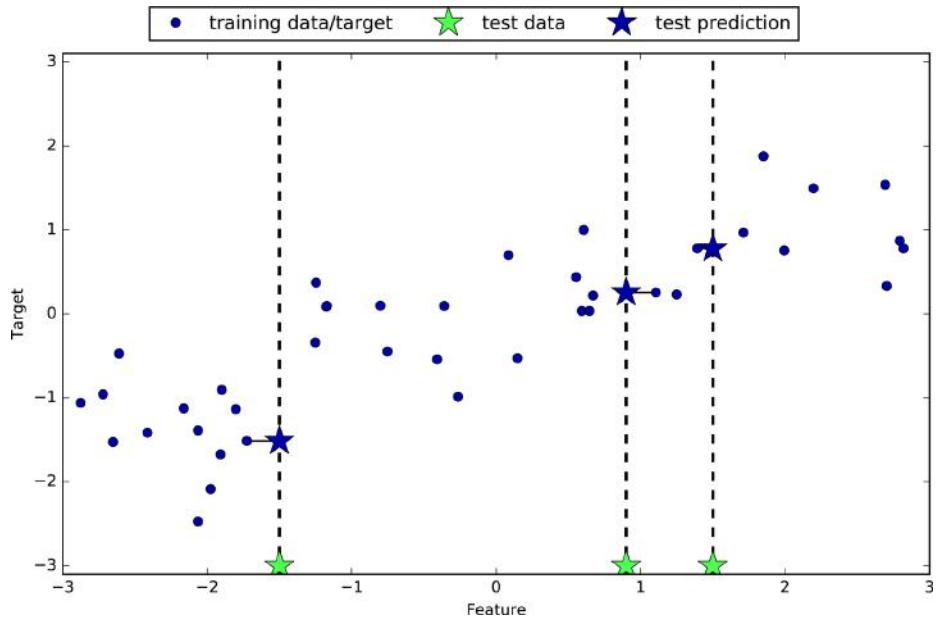


Figure 2-8. Predictions made by one-nearest-neighbor regression on the wave dataset

Again, we can use more than the single closest neighbor for regression. When using multiple nearest neighbors, the prediction is the average, or mean, of the relevant neighbors (Figure 2-9):

In[20]:

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

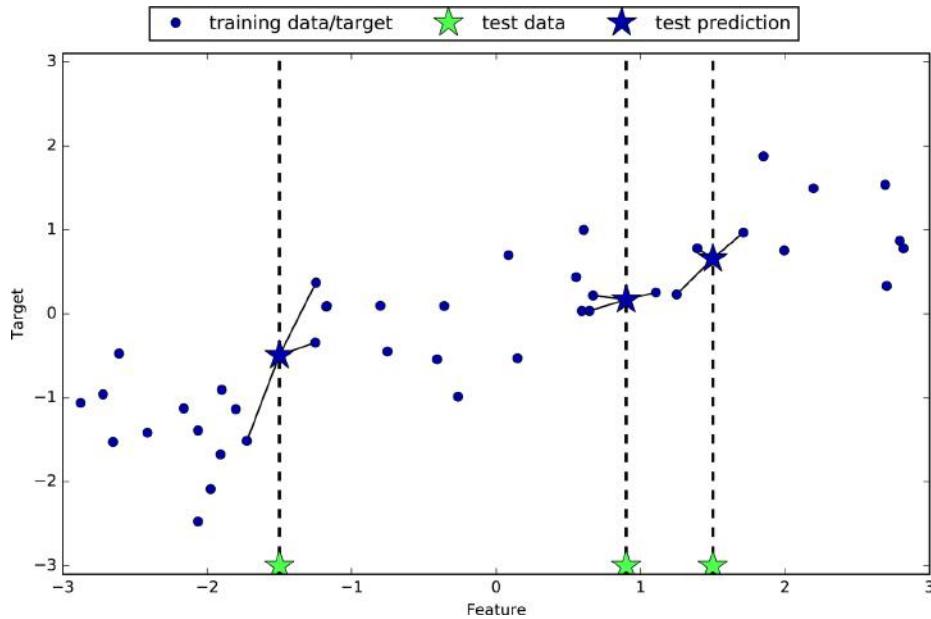


Figure 2-9. Predictions made by three-nearest-neighbors regression on the wave dataset

The k -nearest neighbors algorithm for regression is implemented in the `KNeighborsRegressor` class in scikit-learn. It's used similarly to `KNeighborsClassifier`:

In[21]:

```
from sklearn.neighbors import KNeighborsRegressor

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

# split the wave dataset into a training and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# instantiate the model and set the number of neighbors to consider to 3
reg = KNeighborsRegressor(n_neighbors=3)
# fit the model using the training data and training targets
reg.fit(X_train, y_train)
```

Now we can make predictions on the test set:

In[22]:

```
print("Test set predictions:\n{}".format(reg.predict(X_test)))
```

Out[22]:

```
Test set predictions:  
[-0.054  0.357  1.137 -1.894 -1.139 -1.631  0.357  0.912 -0.447 -1.139]
```

We can also evaluate the model using the `score` method, which for regressors returns the R^2 score. The R^2 score, also known as the coefficient of determination, is a measure of goodness of a prediction for a regression model, and yields a score between 0 and 1. A value of 1 corresponds to a perfect prediction, and a value of 0 corresponds to a constant model that just predicts the mean of the training set responses, `y_train`:

In[23]:

```
print("Test set R^2: {:.2f}".format(reg.score(X_test, y_test)))
```

Out[23]:

```
Test set R^2: 0.83
```

Here, the score is 0.83, which indicates a relatively good model fit.

Analyzing KNeighborsRegressor

For our one-dimensional dataset, we can see what the predictions look like for all possible feature values (Figure 2-10). To do this, we create a test dataset consisting of many points on the line:

In[24]:

```
fig, axes = plt.subplots(1, 3, figsize=(15, 4))  
# create 1,000 data points, evenly spaced between -3 and 3  
line = np.linspace(-3, 3, 1000).reshape(-1, 1)  
for n_neighbors, ax in zip([1, 3, 9], axes):  
    # make predictions using 1, 3, or 9 neighbors  
    reg = KNeighborsRegressor(n_neighbors=n_neighbors)  
    reg.fit(X_train, y_train)  
    ax.plot(line, reg.predict(line))  
    ax.plot(X_train, y_train, '^', c=mpl.cm.cm2(0), markersize=8)  
    ax.plot(X_test, y_test, 'v', c=mpl.cm.cm2(1), markersize=8)  
  
    ax.set_title(  
        "{} neighbor(s)\ntrain score: {:.2f} test score: {:.2f}".format(  
            n_neighbors, reg.score(X_train, y_train),  
            reg.score(X_test, y_test)))  
    ax.set_xlabel("Feature")  
    ax.set_ylabel("Target")  
axes[0].legend(["Model predictions", "Training data/target",  
    "Test data/target"], loc="best")
```

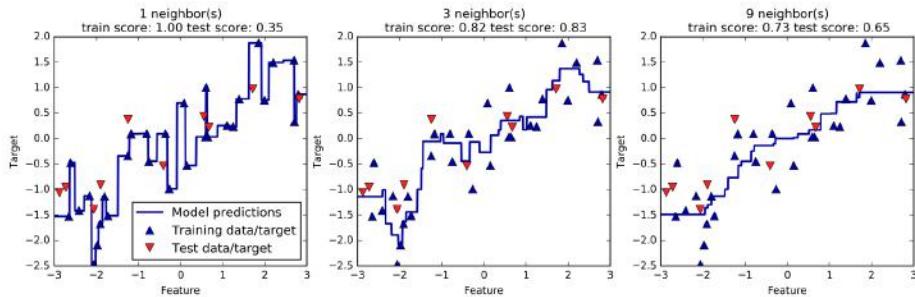


Figure 2-10. Comparing predictions made by nearest neighbors regression for different values of $n_{neighbors}$

As we can see from the plot, using only a single neighbor, each point in the training set has an obvious influence on the predictions, and the predicted values go through all of the data points. This leads to a very unsteady prediction. Considering more neighbors leads to smoother predictions, but these do not fit the training data as well.

Strengths, weaknesses, and parameters

In principle, there are two important parameters to the `KNeighbors` classifier: the number of neighbors and how you measure distance between data points. In practice, using a small number of neighbors like three or five often works well, but you should certainly adjust this parameter. Choosing the right distance measure is somewhat beyond the scope of this book. By default, Euclidean distance is used, which works well in many settings.

One of the strengths of k -NN is that the model is very easy to understand, and often gives reasonable performance without a lot of adjustments. Using this algorithm is a good baseline method to try before considering more advanced techniques. Building the nearest neighbors model is usually very fast, but when your training set is very large (either in number of features or in number of samples) prediction can be slow. When using the k -NN algorithm, it's important to preprocess your data (see [Chapter 3](#)). This approach often does not perform well on datasets with many features (hundreds or more), and it does particularly badly with datasets where most features are 0 most of the time (so-called *sparse datasets*).

So, while the nearest k -neighbors algorithm is easy to understand, it is not often used in practice, due to prediction being slow and its inability to handle many features. The method we discuss next has neither of these drawbacks.

Linear Models

Linear models are a class of models that are widely used in practice and have been studied extensively in the last few decades, with roots going back over a hundred years. Linear models make a prediction using a *linear function* of the input features, which we will explain shortly.

Linear models for regression

For regression, the general prediction formula for a linear model looks as follows:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \dots + w[p] * x[p] + b$$

Here, $x[0]$ to $x[p]$ denotes the features (in this example, the number of features is p) of a single data point, w and b are parameters of the model that are learned, and \hat{y} is the prediction the model makes. For a dataset with a single feature, this is:

$$\hat{y} = w[0] * x[0] + b$$

which you might remember from high school mathematics as the equation for a line. Here, $w[0]$ is the slope and b is the y -axis offset. For more features, w contains the slopes along each feature axis. Alternatively, you can think of the predicted response as being a weighted sum of the input features, with weights (which can be negative) given by the entries of w .

Trying to learn the parameters $w[0]$ and b on our one-dimensional wave dataset might lead to the following line (see [Figure 2-11](#)):

In[25]:

```
mlearn.plots.plot_linear_regression_wave()
```

Out[25]:

```
w[0]: 0.393906  b: -0.031804
```

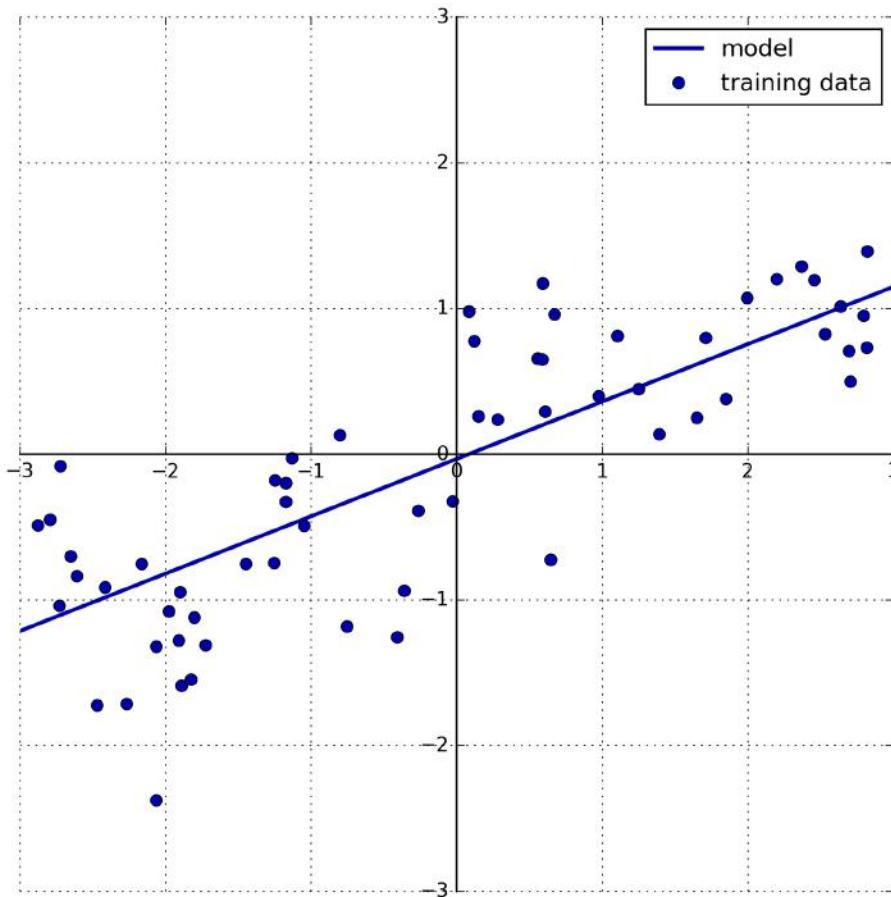


Figure 2-11. Predictions of a linear model on the wave dataset

We added a coordinate cross into the plot to make it easier to understand the line. Looking at $w[0]$ we see that the slope should be around 0.4, which we can confirm visually in the plot. The intercept is where the prediction line should cross the y-axis: this is slightly below zero, which you can also confirm in the image.

Linear models for regression can be characterized as regression models for which the prediction is a line for a single feature, a plane when using two features, or a hyperplane in higher dimensions (that is, when using more features).

If you compare the predictions made by the straight line with those made by the `KNeighborsRegressor` in Figure 2-10, using a straight line to make predictions seems very restrictive. It looks like all the fine details of the data are lost. In a sense, this is true. It is a strong (and somewhat unrealistic) assumption that our target y is a linear

combination of the features. But looking at one-dimensional data gives a somewhat skewed perspective. For datasets with many features, linear models can be very powerful. In particular, if you have more features than training data points, any target y can be perfectly modeled (on the training set) as a linear function.⁶

There are many different linear models for regression. The difference between these models lies in how the model parameters w and b are learned from the training data, and how model complexity can be controlled. We will now take a look at the most popular linear models for regression.

Linear regression (aka ordinary least squares)

Linear regression, or *ordinary least squares* (OLS), is the simplest and most classic linear method for regression. Linear regression finds the parameters w and b that minimize the *mean squared error* between predictions and the true regression targets, y , on the training set. The mean squared error is the sum of the squared differences between the predictions and the true values. Linear regression has no parameters, which is a benefit, but it also has no way to control model complexity.

Here is the code that produces the model you can see in [Figure 2-11](#):

In[26]:

```
from sklearn.linear_model import LinearRegression
X, y = mglearn.datasets.make_wave(n_samples=60)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

lr = LinearRegression().fit(X_train, y_train)
```

The “slope” parameters (w), also called weights or *coefficients*, are stored in the `coef_` attribute, while the offset or *intercept* (b) is stored in the `intercept_` attribute:

In[27]:

```
print("lr.coef_: {}".format(lr.coef_))
print("lr.intercept_: {}".format(lr.intercept_))
```

Out[27]:

```
lr.coef_: [ 0.394]
lr.intercept_: -0.031804343026759746
```

⁶ This is easy to see if you know some linear algebra.



You might notice the strange-looking trailing underscore at the end of `coef_` and `intercept_`. scikit-learn always stores anything that is derived from the training data in attributes that end with a trailing underscore. That is to separate them from parameters that are set by the user.

The `intercept_` attribute is always a single float number, while the `coef_` attribute is a NumPy array with one entry per input feature. As we only have a single input feature in the wave dataset, `lr.coef_` only has a single entry.

Let's look at the training set and test set performance:

In[28]:

```
print("Training set score: {:.2f}".format(lr.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lr.score(X_test, y_test)))
```

Out[28]:

```
Training set score: 0.67
Test set score: 0.66
```

An R^2 of around 0.66 is not very good, but we can see that the scores on the training and test sets are very close together. This means we are likely underfitting, not overfitting. For this one-dimensional dataset, there is little danger of overfitting, as the model is very simple (or restricted). However, with higher-dimensional datasets (meaning datasets with a large number of features), linear models become more powerful, and there is a higher chance of overfitting. Let's take a look at how LinearRegression performs on a more complex dataset, like the Boston Housing dataset. Remember that this dataset has 506 samples and 105 derived features. First, we load the dataset and split it into a training and a test set. Then we build the linear regression model as before:

In[29]:

```
X, y = mglearn.datasets.load_extended_boston()
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
lr = LinearRegression().fit(X_train, y_train)
```

When comparing training set and test set scores, we find that we predict very accurately on the training set, but the R^2 on the test set is much worse:

In[30]:

```
print("Training set score: {:.2f}".format(lr.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lr.score(X_test, y_test)))
```

Out[30]:

```
Training set score: 0.95
Test set score: 0.61
```

This discrepancy between performance on the training set and the test set is a clear sign of overfitting, and therefore we should try to find a model that allows us to control complexity. One of the most commonly used alternatives to standard linear regression is *ridge regression*, which we will look into next.

Ridge regression

Ridge regression is also a linear model for regression, so the formula it uses to make predictions is the same one used for ordinary least squares. In ridge regression, though, the coefficients (w) are chosen not only so that they predict well on the training data, but also to fit an additional constraint. We also want the magnitude of coefficients to be as small as possible; in other words, all entries of w should be close to zero. Intuitively, this means each feature should have as little effect on the outcome as possible (which translates to having a small slope), while still predicting well. This constraint is an example of what is called *regularization*. Regularization means explicitly restricting a model to avoid overfitting. The particular kind used by ridge regression is known as L2 regularization.⁷

Ridge regression is implemented in `linear_model.Ridge`. Let's see how well it does on the extended Boston Housing dataset:

In[31]:

```
from sklearn.linear_model import Ridge

ridge = Ridge().fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge.score(X_test, y_test)))
```

Out[31]:

```
Training set score: 0.89
Test set score: 0.75
```

As you can see, the training set score of `Ridge` is *lower* than for `LinearRegression`, while the test set score is *higher*. This is consistent with our expectation. With linear regression, we were overfitting our data. `Ridge` is a more restricted model, so we are less likely to overfit. A less complex model means worse performance on the training set, but better generalization. As we are only interested in generalization performance, we should choose the `Ridge` model over the `LinearRegression` model.

⁷ Mathematically, `Ridge` penalizes the L2 norm of the coefficients, or the Euclidean length of w .

The Ridge model makes a trade-off between the simplicity of the model (near-zero coefficients) and its performance on the training set. How much importance the model places on simplicity versus training set performance can be specified by the user, using the `alpha` parameter. In the previous example, we used the default parameter `alpha=1.0`. There is no reason why this will give us the best trade-off, though. The optimum setting of `alpha` depends on the particular dataset we are using. Increasing `alpha` forces coefficients to move more toward zero, which decreases training set performance but might help generalization. For example:

In[32]:

```
ridge10 = Ridge(alpha=10).fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge10.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge10.score(X_test, y_test)))
```

Out[32]:

```
Training set score: 0.79
Test set score: 0.64
```

Decreasing `alpha` allows the coefficients to be less restricted, meaning we move right in [Figure 2-1](#). For very small values of `alpha`, coefficients are barely restricted at all, and we end up with a model that resembles `LinearRegression`:

In[33]:

```
ridge01 = Ridge(alpha=0.1).fit(X_train, y_train)
print("Training set score: {:.2f}".format(ridge01.score(X_train, y_train)))
print("Test set score: {:.2f}".format(ridge01.score(X_test, y_test)))
```

Out[33]:

```
Training set score: 0.93
Test set score: 0.77
```

Here, `alpha=0.1` seems to be working well. We could try decreasing `alpha` even more to improve generalization. For now, notice how the parameter `alpha` corresponds to the model complexity as shown in [Figure 2-1](#). We will discuss methods to properly select parameters in [Chapter 5](#).

We can also get a more qualitative insight into how the `alpha` parameter changes the model by inspecting the `coef_` attribute of models with different values of `alpha`. A higher `alpha` means a more restricted model, so we expect the entries of `coef_` to have smaller magnitude for a high value of `alpha` than for a low value of `alpha`. This is confirmed in the plot in [Figure 2-12](#):

In[34]:

```
plt.plot(ridge.coef_, 's', label="Ridge alpha=1")
plt.plot(ridge10.coef_, '^', label="Ridge alpha=10")
plt.plot(ridge01.coef_, 'v', label="Ridge alpha=0.1")

plt.plot(lr.coef_, 'o', label="LinearRegression")
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
plt.hlines(0, 0, len(lr.coef_))
plt.ylim(-25, 25)
plt.legend()
```

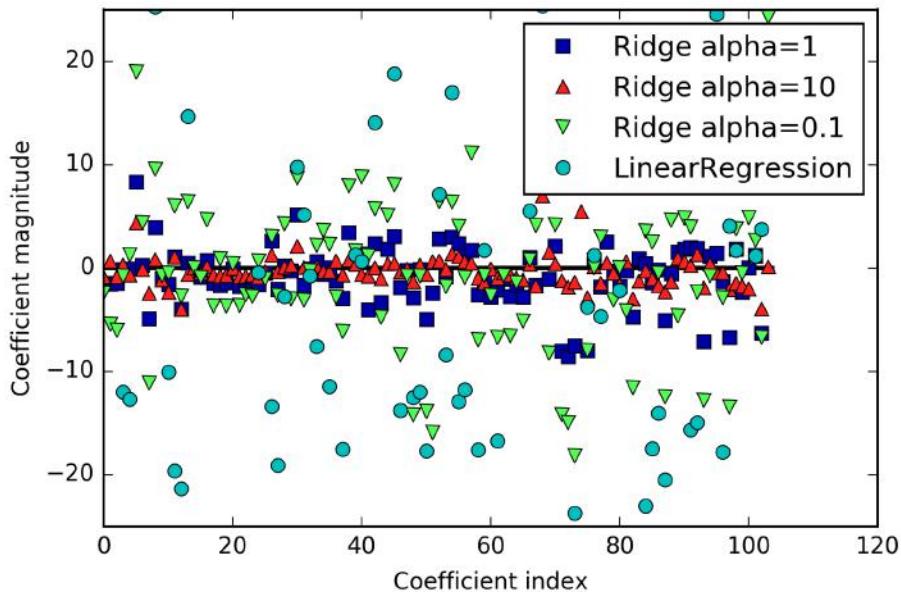


Figure 2-12. Comparing coefficient magnitudes for ridge regression with different values of alpha and linear regression

Here, the x-axis enumerates the entries of `coef_`: $x=0$ shows the coefficient associated with the first feature, $x=1$ the coefficient associated with the second feature, and so on up to $x=100$. The y-axis shows the numeric values of the corresponding values of the coefficients. The main takeaway here is that for $\alpha=10$, the coefficients are mostly between around -3 and 3. The coefficients for the Ridge model with $\alpha=1$ are somewhat larger. The dots corresponding to $\alpha=0.1$ have larger magnitude still, and many of the dots corresponding to linear regression without any regularization (which would be $\alpha=0$) are so large they are outside of the chart.

Another way to understand the influence of regularization is to fix a value of alpha but vary the amount of training data available. For Figure 2-13, we subsampled the Boston Housing dataset and evaluated LinearRegression and Ridge(alpha=1) on subsets of increasing size (plots that show model performance as a function of dataset size are called *learning curves*):

In[35]:

```
mlearn.plots.plot_ridge_n_samples()
```

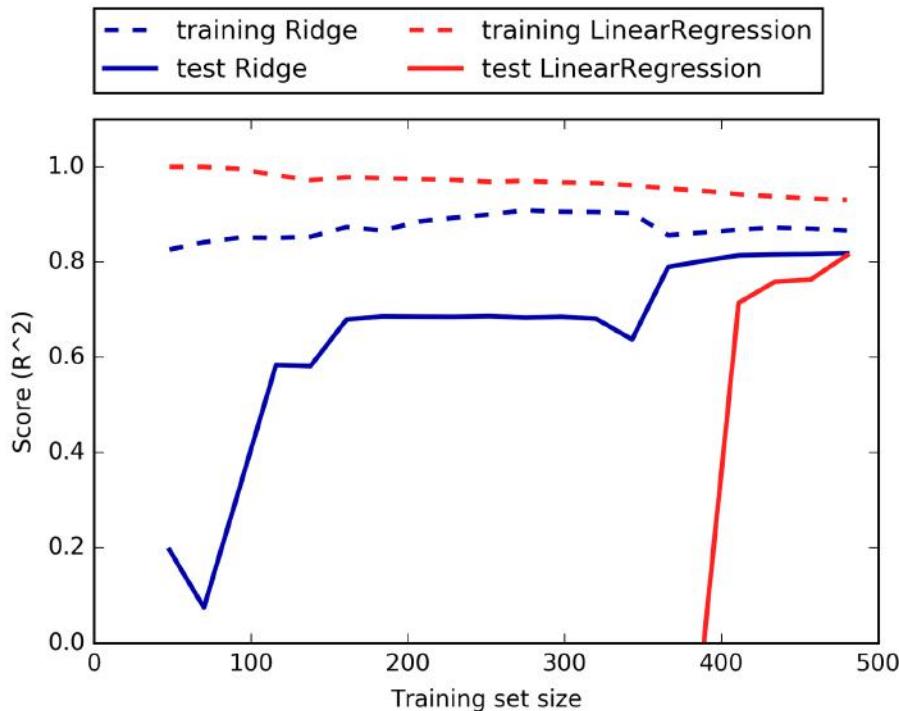


Figure 2-13. Learning curves for ridge regression and linear regression on the Boston Housing dataset

As one would expect, the training score is higher than the test score for all dataset sizes, for both ridge and linear regression. Because ridge is regularized, the training score of ridge is lower than the training score for linear regression across the board. However, the test score for ridge is better, particularly for small subsets of the data. For less than 400 data points, linear regression is not able to learn anything. As more and more data becomes available to the model, both models improve, and linear regression catches up with ridge in the end. The lesson here is that with enough training data, regularization becomes less important, and given enough data, ridge and

linear regression will have the same performance (the fact that this happens here when using the full dataset is just by chance). Another interesting aspect of [Figure 2-13](#) is the decrease in training performance for linear regression. If more data is added, it becomes harder for a model to overfit, or memorize the data.

Lasso

An alternative to Ridge for regularizing linear regression is Lasso. As with ridge regression, using the lasso also restricts coefficients to be close to zero, but in a slightly different way, called L1 regularization.⁸ The consequence of L1 regularization is that when using the lasso, some coefficients are *exactly zero*. This means some features are entirely ignored by the model. This can be seen as a form of automatic feature selection. Having some coefficients be exactly zero often makes a model easier to interpret, and can reveal the most important features of your model.

Let's apply the lasso to the extended Boston Housing dataset:

In[36]:

```
from sklearn.linear_model import Lasso

lasso = Lasso().fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso.coef_ != 0)))
```

Out[36]:

```
Training set score: 0.29
Test set score: 0.21
Number of features used: 4
```

As you can see, Lasso does quite badly, both on the training and the test set. This indicates that we are underfitting, and we find that it used only 4 of the 105 features. Similarly to Ridge, the Lasso also has a regularization parameter, alpha, that controls how strongly coefficients are pushed toward zero. In the previous example, we used the default of `alpha=1.0`. To reduce underfitting, let's try decreasing `alpha`. When we do this, we also need to increase the default setting of `max_iter` (the maximum number of iterations to run):

⁸ The lasso penalizes the L1 norm of the coefficient vector—or in other words, the sum of the absolute values of the coefficients.

In[37]:

```
# we increase the default setting of "max_iter",
# otherwise the model would warn us that we should increase max_iter.
lasso001 = Lasso(alpha=0.01, max_iter=100000).fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso001.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso001.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso001.coef_ != 0)))
```

Out[37]:

```
Training set score: 0.90
Test set score: 0.77
Number of features used: 33
```

A lower `alpha` allowed us to fit a more complex model, which worked better on the training and test data. The performance is slightly better than using `Ridge`, and we are using only 33 of the 105 features. This makes this model potentially easier to understand.

If we set `alpha` too low, however, we again remove the effect of regularization and end up overfitting, with a result similar to `LinearRegression`:

In[38]:

```
lasso00001 = Lasso(alpha=0.0001, max_iter=100000).fit(X_train, y_train)
print("Training set score: {:.2f}".format(lasso00001.score(X_train, y_train)))
print("Test set score: {:.2f}".format(lasso00001.score(X_test, y_test)))
print("Number of features used: {}".format(np.sum(lasso00001.coef_ != 0)))
```

Out[38]:

```
Training set score: 0.95
Test set score: 0.64
Number of features used: 94
```

Again, we can plot the coefficients of the different models, similarly to Figure 2-12. The result is shown in Figure 2-14:

In[39]:

```
plt.plot(lasso.coef_, 's', label="Lasso alpha=1")
plt.plot(lasso001.coef_, '^', label="Lasso alpha=0.01")
plt.plot(lasso00001.coef_, 'v', label="Lasso alpha=0.0001")

plt.plot(ridge01.coef_, 'o', label="Ridge alpha=0.1")
plt.legend(ncol=2, loc=(0, 1.05))
plt.ylim(-25, 25)
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
```

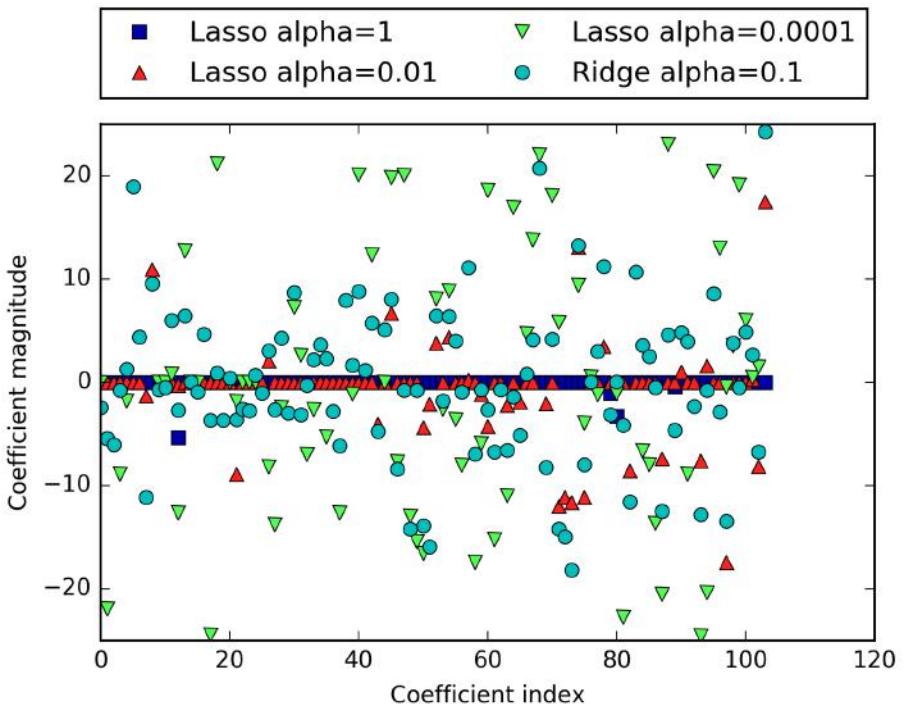


Figure 2-14. Comparing coefficient magnitudes for lasso regression with different values of α and ridge regression

For $\alpha=1$, we not only see that most of the coefficients are zero (which we already knew), but that the remaining coefficients are also small in magnitude. Decreasing α to 0.01, we obtain the solution shown as the green dots, which causes most features to be exactly zero. Using $\alpha=0.0001$, we get a model that is quite unregularized, with most coefficients nonzero and of large magnitude. For comparison, the best Ridge solution is shown in teal. The Ridge model with $\alpha=0.1$ has similar predictive performance as the lasso model with $\alpha=0.01$, but using Ridge, all coefficients are nonzero.

In practice, ridge regression is usually the first choice between these two models. However, if you have a large amount of features and expect only a few of them to be important, Lasso might be a better choice. Similarly, if you would like to have a model that is easy to interpret, Lasso will provide a model that is easier to understand, as it will select only a subset of the input features. scikit-learn also provides the `ElasticNet` class, which combines the penalties of Lasso and Ridge. In practice, this combination works best, though at the price of having two parameters to adjust: one for the L1 regularization, and one for the L2 regularization.

Linear models for classification

Linear models are also extensively used for classification. Let's look at binary classification first. In this case, a prediction is made using the following formula:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \dots + w[p] * x[p] + b > 0$$

The formula looks very similar to the one for linear regression, but instead of just returning the weighted sum of the features, we threshold the predicted value at zero. If the function is smaller than zero, we predict the class -1 ; if it is larger than zero, we predict the class $+1$. This prediction rule is common to all linear models for classification. Again, there are many different ways to find the coefficients (w) and the intercept (b).

For linear models for regression, the output, \hat{y} , is a linear function of the features: a line, plane, or hyperplane (in higher dimensions). For linear models for classification, the *decision boundary* is a linear function of the input. In other words, a (binary) linear classifier is a classifier that separates two classes using a line, a plane, or a hyperplane. We will see examples of that in this section.

There are many algorithms for learning linear models. These algorithms all differ in the following two ways:

- The way in which they measure how well a particular combination of coefficients and intercept fits the training data
- If and what kind of regularization they use

Different algorithms choose different ways to measure what “fitting the training set well” means. For technical mathematical reasons, it is not possible to adjust w and b to minimize the number of misclassifications the algorithms produce, as one might hope. For our purposes, and many applications, the different choices for item 1 in the preceding list (called *loss functions*) are of little significance.

The two most common linear classification algorithms are *logistic regression*, implemented in `linear_model.LogisticRegression`, and *linear support vector machines* (linear SVMs), implemented in `svm.LinearSVC` (SVC stands for support vector classifier). Despite its name, `LogisticRegression` is a classification algorithm and not a regression algorithm, and it should not be confused with `LinearRegression`.

We can apply the `LogisticRegression` and `LinearSVC` models to the `forge` dataset, and visualize the decision boundary as found by the linear models (Figure 2-15):

In[40]:

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC

X, y = mglearn.datasets.make_forge()

fig, axes = plt.subplots(1, 2, figsize=(10, 3))

for model, ax in zip([LinearSVC(), LogisticRegression()], axes):
    clf = model.fit(X, y)
    mglearn.plots.plot_2d_separator(clf, X, fill=False, eps=0.5,
                                    ax=ax, alpha=.7)
    mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
    ax.set_title("{}".format(clf.__class__.__name__))
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
    axes[0].legend()
```

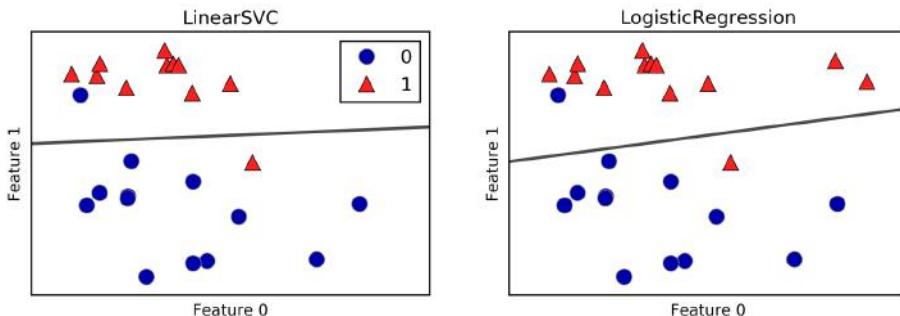


Figure 2-15. Decision boundaries of a linear SVM and logistic regression on the forge dataset with the default parameters

In this figure, we have the first feature of the forge dataset on the x-axis and the second feature on the y-axis, as before. We display the decision boundaries found by `LinearSVC` and `LogisticRegression` respectively as straight lines, separating the area classified as class 1 on the top from the area classified as class 0 on the bottom. In other words, any new data point that lies above the black line will be classified as class 1 by the respective classifier, while any point that lies below the black line will be classified as class 0.

The two models come up with similar decision boundaries. Note that both misclassify two of the points. By default, both models apply an L2 regularization, in the same way that `Ridge` does for regression.

For `LogisticRegression` and `LinearSVC` the trade-off parameter that determines the strength of the regularization is called `C`, and higher values of `C` correspond to *less*

regularization. In other words, when you use a high value for the parameter C , LogisticRegression and LinearSVC try to fit the training set as best as possible, while with low values of the parameter C , the models put more emphasis on finding a coefficient vector (w) that is close to zero.

There is another interesting aspect of how the parameter C acts. Using low values of C will cause the algorithms to try to adjust to the “majority” of data points, while using a higher value of C stresses the importance that each individual data point be classified correctly. Here is an illustration using LinearSVC (Figure 2-16):

In[41]:

```
mglearn.plots.plot_linear_svc_regularization()
```

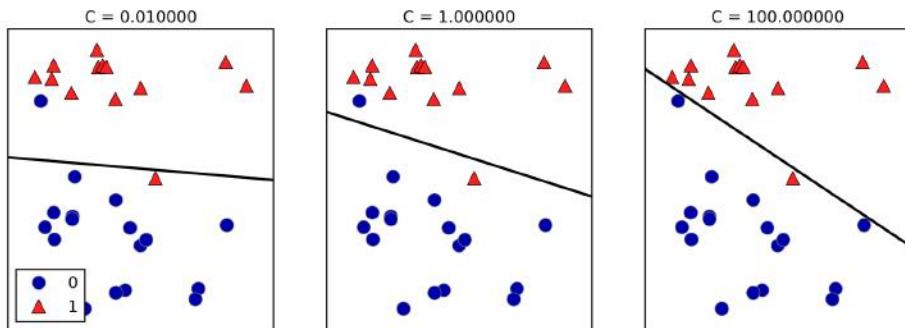


Figure 2-16. Decision boundaries of a linear SVM on the forge dataset for different values of C

On the lefthand side, we have a very small C corresponding to a lot of regularization. Most of the points in class 0 are at the top, and most of the points in class 1 are at the bottom. The strongly regularized model chooses a relatively horizontal line, misclassifying two points. In the center plot, C is slightly higher, and the model focuses more on the two misclassified samples, tilting the decision boundary. Finally, on the righthand side, the very high value of C in the model tilts the decision boundary a lot, now correctly classifying all points in class 0. One of the points in class 1 is still misclassified, as it is not possible to correctly classify all points in this dataset using a straight line. The model illustrated on the righthand side tries hard to correctly classify all points, but might not capture the overall layout of the classes well. In other words, this model is likely overfitting.

Similarly to the case of regression, linear models for classification might seem very restrictive in low-dimensional spaces, only allowing for decision boundaries that are straight lines or planes. Again, in high dimensions, linear models for classification

become very powerful, and guarding against overfitting becomes increasingly important when considering more features.

Let's analyze `LinearLogistic` in more detail on the Breast Cancer dataset:

In[42]:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
logreg = LogisticRegression().fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg.score(X_test, y_test)))
```

Out[42]:

```
Training set score: 0.953
Test set score: 0.958
```

The default value of $C=1$ provides quite good performance, with 95% accuracy on both the training and the test set. But as training and test set performance are very close, it is likely that we are underfitting. Let's try to increase C to fit a more flexible model:

In[43]:

```
logreg100 = LogisticRegression(C=100).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg100.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg100.score(X_test, y_test)))
```

Out[43]:

```
Training set score: 0.972
Test set score: 0.965
```

Using $C=100$ results in higher training set accuracy, and also a slightly increased test set accuracy, confirming our intuition that a more complex model should perform better.

We can also investigate what happens if we use an even more regularized model than the default of $C=1$, by setting $C=0.01$:

In[44]:

```
logreg001 = LogisticRegression(C=0.01).fit(X_train, y_train)
print("Training set score: {:.3f}".format(logreg001.score(X_train, y_train)))
print("Test set score: {:.3f}".format(logreg001.score(X_test, y_test)))
```

Out[44]:

```
Training set score: 0.934
Test set score: 0.930
```

As expected, when moving more to the left along the scale shown in [Figure 2-1](#) from an already underfit model, both training and test set accuracy decrease relative to the default parameters.

Finally, let's look at the coefficients learned by the models with the three different settings of the regularization parameter C ([Figure 2-17](#)):

In[45]:

```
plt.plot(logreg.coef_.T, 'o', label="C=1")
plt.plot(logreg100.coef_.T, '^', label="C=100")
plt.plot(logreg001.coef_.T, 'v', label="C=0.001")
plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.ylim(-5, 5)
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")
plt.legend()
```



As `LogisticRegression` applies an L2 regularization by default, the result looks similar to that produced by `Ridge` in [Figure 2-12](#). Stronger regularization pushes coefficients more and more toward zero, though coefficients never become exactly zero. Inspecting the plot more closely, we can also see an interesting effect in the third coefficient, for “mean perimeter.” For $C=100$ and $C=1$, the coefficient is negative, while for $C=0.001$, the coefficient is positive, with a magnitude that is even larger than for $C=1$. Interpreting a model like this, one might think the coefficient tells us which class a feature is associated with. For example, one might think that a high “texture error” feature is related to a sample being “malignant.” However, the change of sign in the coefficient for “mean perimeter” means that depending on which model we look at, a high “mean perimeter” could be taken as being either indicative of “benign” or indicative of “malignant.” This illustrates that interpretations of coefficients of linear models should always be taken with a grain of salt.

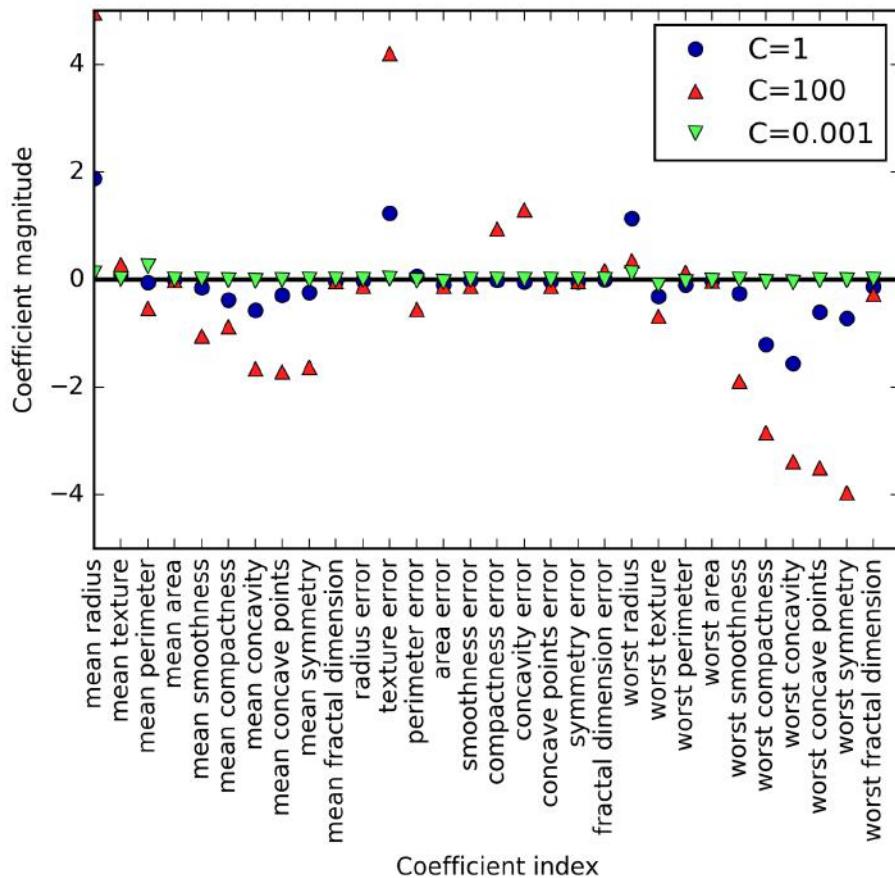


Figure 2-17. Coefficients learned by logistic regression on the Breast Cancer dataset for different values of C

If we desire a more interpretable model, using L1 regularization might help, as it limits the model to using only a few features. Here is the coefficient plot and classification accuracies for L1 regularization (Figure 2-18):

In[46]:

```
for C, marker in zip([0.001, 1, 100], ['o', '^', 'v']):
    lr_l1 = LogisticRegression(C=C, penalty="l1").fit(X_train, y_train)
    print("Training accuracy of l1 logreg with C={:.3f}: {:.2f}".format(
        C, lr_l1.score(X_train, y_train)))
    print("Test accuracy of l1 logreg with C={:.3f}: {:.2f}".format(
        C, lr_l1.score(X_test, y_test)))
    plt.plot(lr_l1.coef_.T, marker, label="C={:.3f}".format(C))

plt.xticks(range(cancer.data.shape[1]), cancer.feature_names, rotation=90)
plt.hlines(0, 0, cancer.data.shape[1])
plt.xlabel("Coefficient index")
plt.ylabel("Coefficient magnitude")

plt.ylim(-5, 5)
plt.legend(loc=3)
```

Out[46]:

```
Training accuracy of l1 logreg with C=0.001: 0.91
Test accuracy of l1 logreg with C=0.001: 0.92
Training accuracy of l1 logreg with C=1.000: 0.96
Test accuracy of l1 logreg with C=1.000: 0.96
Training accuracy of l1 logreg with C=100.000: 0.99
Test accuracy of l1 logreg with C=100.000: 0.98
```

As you can see, there are many parallels between linear models for binary classification and linear models for regression. As in regression, the main difference between the models is the `penalty` parameter, which influences the regularization and whether the model will use all available features or select only a subset.

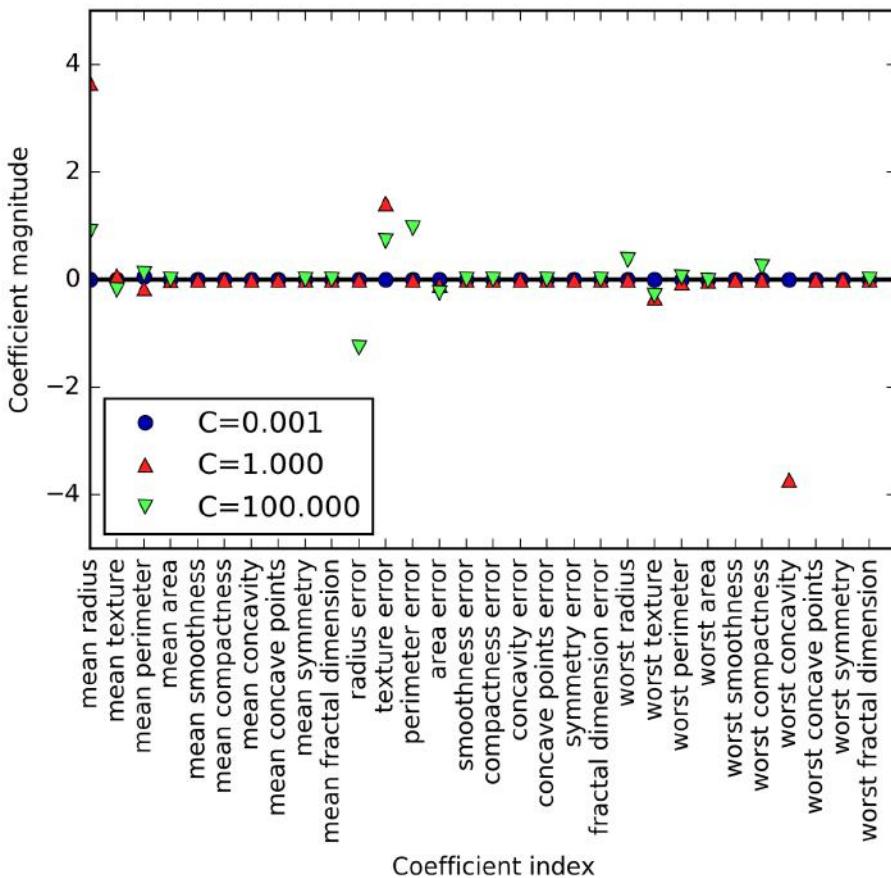


Figure 2-18. Coefficients learned by logistic regression with L1 penalty on the Breast Cancer dataset for different values of C

Linear models for multiclass classification

Many linear classification models are for binary classification only, and don't extend naturally to the multiclass case (with the exception of logistic regression). A common technique to extend a binary classification algorithm to a multiclass classification algorithm is the *one-vs.-rest* approach. In the one-vs.-rest approach, a binary model is learned for each class that tries to separate that class from all of the other classes, resulting in as many binary models as there are classes. To make a prediction, all binary classifiers are run on a test point. The classifier that has the highest score on its single class "wins," and this class label is returned as the prediction.

Having one binary classifier per class results in having one vector of coefficients (w) and one intercept (b) for each class. The class for which the result of the classification confidence formula given here is highest is the assigned class label:

$$w[0] * x[0] + w[1] * x[1] + \dots + w[p] * x[p] + b$$

The mathematics behind multiclass logistic regression differ somewhat from the one-vs.-rest approach, but they also result in one coefficient vector and intercept per class, and the same method of making a prediction is applied.

Let's apply the one-vs.-rest method to a simple three-class classification dataset. We use a two-dimensional dataset, where each class is given by data sampled from a Gaussian distribution (see [Figure 2-19](#)):

In[47]:

```
from sklearn.datasets import make_blobs

X, y = make_blobs(random_state=42)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(["Class 0", "Class 1", "Class 2"])
```

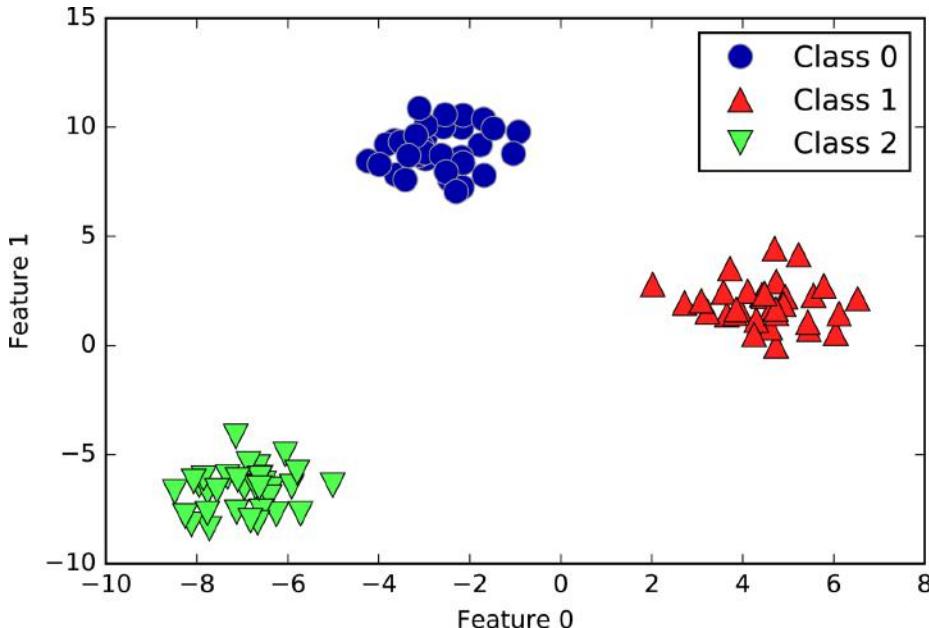


Figure 2-19. Two-dimensional toy dataset containing three classes

Now, we train a `LinearSVC` classifier on the dataset:

In[48]:

```
linear_svm = LinearSVC().fit(X, y)
print("Coefficient shape: ", linear_svm.coef_.shape)
print("Intercept shape: ", linear_svm.intercept_.shape)
```

Out[48]:

```
Coefficient shape: (3, 2)
Intercept shape: (3,)
```

We see that the shape of the `coef_` is $(3, 2)$, meaning that each row of `coef_` contains the coefficient vector for one of the three classes and each column holds the coefficient value for a specific feature (there are two in this dataset). The `intercept_` is now a one-dimensional array, storing the intercepts for each class.

Let's visualize the lines given by the three binary classifiers (Figure 2-20):

In[49]:

```
mlearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_,
['b', 'r', 'g']):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.ylim(-10, 15)
plt.xlim(-10, 8)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
plt.legend(['Class 0', 'Class 1', 'Class 2', 'Line class 0', 'Line class 1',
'Line class 2'], loc=(1.01, 0.3))
```

You can see that all the points belonging to class 0 in the training data are above the line corresponding to class 0, which means they are on the “class 0” side of this binary classifier. The points in class 0 are above the line corresponding to class 2, which means they are classified as “rest” by the binary classifier for class 2. The points belonging to class 0 are to the left of the line corresponding to class 1, which means the binary classifier for class 1 also classifies them as “rest.” Therefore, any point in this area will be classified as class 0 by the final classifier (the result of the classification confidence formula for classifier 0 is greater than zero, while it is smaller than zero for the other two classes).

But what about the triangle in the middle of the plot? All three binary classifiers classify points there as “rest.” Which class would a point there be assigned to? The answer is the one with the highest value for the classification formula: the class of the closest line.

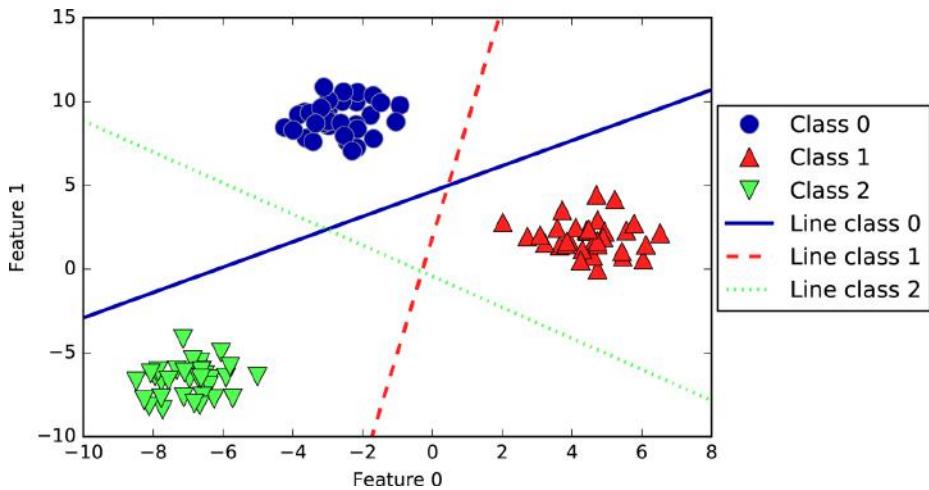


Figure 2-20. Decision boundaries learned by the three one-vs.-rest classifiers

The following example (Figure 2-21) shows the predictions for all regions of the 2D space:

In[50]:

```
mglearn.plots.plot_2d_classification(linear_svm, X, fill=True, alpha=.7)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
line = np.linspace(-15, 15)
for coef, intercept, color in zip(linear_svm.coef_, linear_svm.intercept_,
                                  ['b', 'r', 'g']):
    plt.plot(line, -(line * coef[0] + intercept) / coef[1], c=color)
plt.legend(['Class 0', 'Class 1', 'Class 2', 'Line class 0', 'Line class 1',
           'Line class 2'], loc=(1.01, 0.3))
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

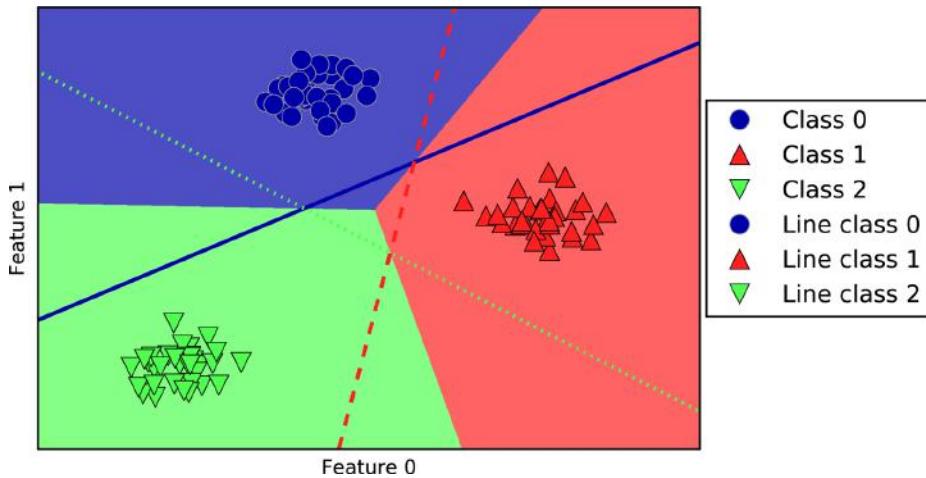


Figure 2-21. Multiclass decision boundaries derived from the three one-vs.-rest classifiers

Strengths, weaknesses, and parameters

The main parameter of linear models is the regularization parameter, called `alpha` in the regression models and `C` in `LinearSVC` and `LogisticRegression`. Large values for `alpha` or small values for `C` mean simple models. In particular for the regression models, tuning these parameters is quite important. Usually `C` and `alpha` are searched for on a logarithmic scale. The other decision you have to make is whether you want to use L1 regularization or L2 regularization. If you assume that only a few of your features are actually important, you should use L1. Otherwise, you should default to L2. L1 can also be useful if interpretability of the model is important. As L1 will use only a few features, it is easier to explain which features are important to the model, and what the effects of these features are.

Linear models are very fast to train, and also fast to predict. They scale to very large datasets and work well with sparse data. If your data consists of hundreds of thousands or millions of samples, you might want to investigate using the `solver='sag'` option in `LogisticRegression` and `Ridge`, which can be faster than the default on large datasets. Other options are the `SGDClassifier` class and the `SGDRegressor` class, which implement even more scalable versions of the linear models described here.

Another strength of linear models is that they make it relatively easy to understand how a prediction is made, using the formulas we saw earlier for regression and classification. Unfortunately, it is often not entirely clear why coefficients are the way they are. This is particularly true if your dataset has highly correlated features; in these cases, the coefficients might be hard to interpret.

Linear models often perform well when the number of features is large compared to the number of samples. They are also often used on very large datasets, simply because it's not feasible to train other models. However, in lower-dimensional spaces, other models might yield better generalization performance. We will look at some examples in which linear models fail in “[Kernelized Support Vector Machines](#)” on [page 92](#).

Method Chaining

The `fit` method of all `scikit-learn` models returns `self`. This allows you to write code like the following, which we've already used extensively in this chapter:

In[51]:

```
# instantiate model and fit it in one line
logreg = LogisticRegression().fit(X_train, y_train)
```

Here, we used the return value of `fit` (which is `self`) to assign the trained model to the variable `logreg`. This concatenation of method calls (here `__init__` and then `fit`) is known as *method chaining*. Another common application of method chaining in `scikit-learn` is to `fit` and `predict` in one line:

In[52]:

```
logreg = LogisticRegression()
y_pred = logreg.fit(X_train, y_train).predict(X_test)
```

Finally, you can even do model instantiation, fitting, and predicting in one line:

In[53]:

```
y_pred = LogisticRegression().fit(X_train, y_train).predict(X_test)
```

This very short variant is not ideal, though. A lot is happening in a single line, which might make the code hard to read. Additionally, the fitted logistic regression model isn't stored in any variable, so we can't inspect it or use it to predict on any other data.

Naive Bayes Classifiers

Naive Bayes classifiers are a family of classifiers that are quite similar to the linear models discussed in the previous section. However, they tend to be even faster in training. The price paid for this efficiency is that naive Bayes models often provide generalization performance that is slightly worse than that of linear classifiers like `LogisticRegression` and `LinearSVC`.

The reason that naive Bayes models are so efficient is that they learn parameters by looking at each feature individually and collect simple per-class statistics from each feature. There are three kinds of naive Bayes classifiers implemented in `scikit-`

`learn`: `GaussianNB`, `BernoulliNB`, and `MultinomialNB`. `GaussianNB` can be applied to any continuous data, while `BernoulliNB` assumes binary data and `MultinomialNB` assumes count data (that is, that each feature represents an integer count of something, like how often a word appears in a sentence). `BernoulliNB` and `MultinomialNB` are mostly used in text data classification.

The `BernoulliNB` classifier counts how often every feature of each class is not zero. This is most easily understood with an example:

In[54]:

```
X = np.array([[0, 1, 0, 1],
              [1, 0, 1, 1],
              [0, 0, 0, 1],
              [1, 0, 1, 0]])
y = np.array([0, 1, 0, 1])
```

Here, we have four data points, with four binary features each. There are two classes, 0 and 1. For class 0 (the first and third data points), the first feature is zero two times and nonzero zero times, the second feature is zero one time and nonzero one time, and so on. These same counts are then calculated for the data points in the second class. Counting the nonzero entries per class in essence looks like this:

In[55]:

```
counts = {}
for label in np.unique(y):
    # iterate over each class
    # count (sum) entries of 1 per feature
    counts[label] = X[y == label].sum(axis=0)
print("Feature counts:\n{}".format(counts))
```

Out[55]:

```
Feature counts:
{0: array([0, 1, 0, 2]), 1: array([2, 0, 2, 1])}
```

The other two naive Bayes models, `MultinomialNB` and `GaussianNB`, are slightly different in what kinds of statistics they compute. `MultinomialNB` takes into account the average value of each feature for each class, while `GaussianNB` stores the average value as well as the standard deviation of each feature for each class.

To make a prediction, a data point is compared to the statistics for each of the classes, and the best matching class is predicted. Interestingly, for both `MultinomialNB` and `BernoulliNB`, this leads to a prediction formula that is of the same form as in the linear models (see “[Linear models for classification](#)” on page 56). Unfortunately, `coef_` for the naive Bayes models has a somewhat different meaning than in the linear models, in that `coef_` is not the same as w .

Strengths, weaknesses, and parameters

`MultinomialNB` and `BernoulliNB` have a single parameter, `alpha`, which controls model complexity. The way `alpha` works is that the algorithm adds to the data `alpha` many virtual data points that have positive values for all the features. This results in a “smoothing” of the statistics. A large `alpha` means more smoothing, resulting in less complex models. The algorithm’s performance is relatively robust to the setting of `alpha`, meaning that setting `alpha` is not critical for good performance. However, tuning it usually improves accuracy somewhat.

`GaussianNB` is mostly used on very high-dimensional data, while the other two variants of naive Bayes are widely used for sparse count data such as text. `MultinomialNB` usually performs better than `BinaryNB`, particularly on datasets with a relatively large number of nonzero features (i.e., large documents).

The naive Bayes models share many of the strengths and weaknesses of the linear models. They are very fast to train and to predict, and the training procedure is easy to understand. The models work very well with high-dimensional sparse data and are relatively robust to the parameters. Naive Bayes models are great baseline models and are often used on very large datasets, where training even a linear model might take too long.

Decision Trees

Decision trees are widely used models for classification and regression tasks. Essentially, they learn a hierarchy of if/else questions, leading to a decision.

These questions are similar to the questions you might ask in a game of 20 Questions. Imagine you want to distinguish between the following four animals: bears, hawks, penguins, and dolphins. Your goal is to get to the right answer by asking as few if/else questions as possible. You might start off by asking whether the animal has feathers, a question that narrows down your possible animals to just two. If the answer is “yes,” you can ask another question that could help you distinguish between hawks and penguins. For example, you could ask whether the animal can fly. If the animal doesn’t have feathers, your possible animal choices are dolphins and bears, and you will need to ask a question to distinguish between these two animals—for example, asking whether the animal has fins.

This series of questions can be expressed as a decision tree, as shown in [Figure 2-22](#).

In[56]:

```
mglearn.plots.plot_animal_tree()
```

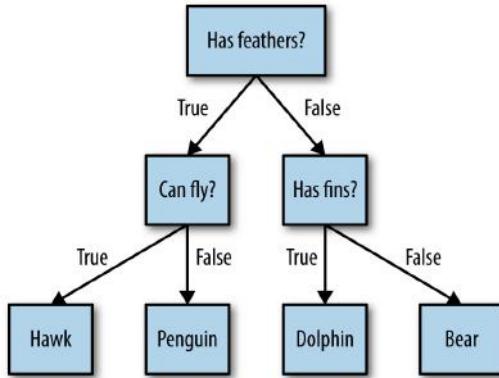


Figure 2-22. A decision tree to distinguish among several animals

In this illustration, each node in the tree either represents a question or a terminal node (also called a *leaf*) that contains the answer. The edges connect the answers to a question with the next question you would ask.

In machine learning parlance, we built a model to distinguish between four classes of animals (hawks, penguins, dolphins, and bears) using the three features “has feathers,” “can fly,” and “has fins.” Instead of building these models by hand, we can learn them from data using supervised learning.

Building decision trees

Let’s go through the process of building a decision tree for the 2D classification dataset shown in [Figure 2-23](#). The dataset consists of two half-moon shapes, with each class consisting of 75 data points. We will refer to this dataset as `two_moons`.

Learning a decision tree means learning the sequence of if/else questions that gets us to the true answer most quickly. In the machine learning setting, these questions are called *tests* (not to be confused with the test set, which is the data we use to test to see how generalizable our model is). Usually data does not come in the form of binary yes/no features as in the animal example, but is instead represented as continuous features such as in the 2D dataset shown in [Figure 2-23](#). The tests that are used on continuous data are of the form “Is feature i larger than value a ?”

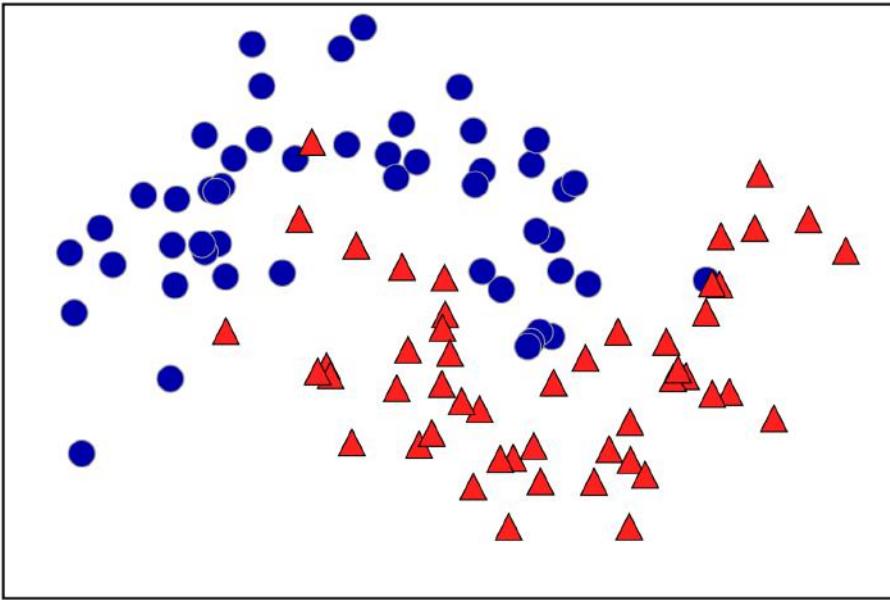


Figure 2-23. Two-moons dataset on which the decision tree will be built

To build a tree, the algorithm searches over all possible tests and finds the one that is most informative about the target variable. [Figure 2-24](#) shows the first test that is picked. Splitting the dataset vertically at $x[1]=0.0596$ yields the most information; it best separates the points in class 1 from the points in class 2. The top node, also called the *root*, represents the whole dataset, consisting of 75 points belonging to class 0 and 75 points belonging to class 1. The split is done by testing whether $x[1] \leq 0.0596$, indicated by a black line. If the test is true, a point is assigned to the left node, which contains 2 points belonging to class 0 and 32 points belonging to class 1. Otherwise the point is assigned to the right node, which contains 48 points belonging to class 0 and 18 points belonging to class 1. These two nodes correspond to the top and bottom regions shown in [Figure 2-24](#). Even though the first split did a good job of separating the two classes, the bottom region still contains points belonging to class 0, and the top region still contains points belonging to class 1. We can build a more accurate model by repeating the process of looking for the best test in both regions. [Figure 2-25](#) shows that the most informative next split for the left and the right region is based on $x[0]$.

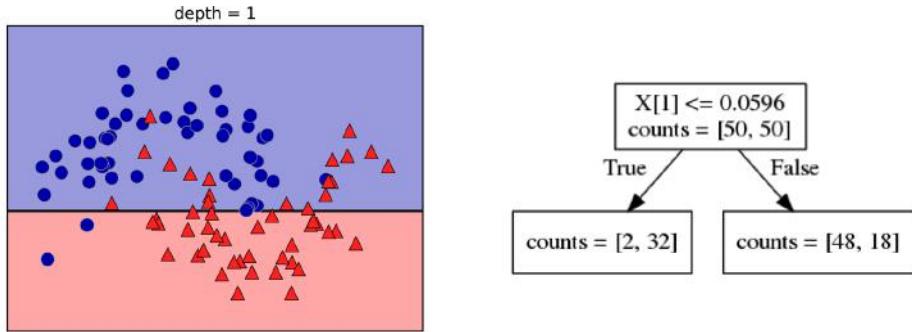


Figure 2-24. Decision boundary of tree with depth 1 (left) and corresponding tree (right)

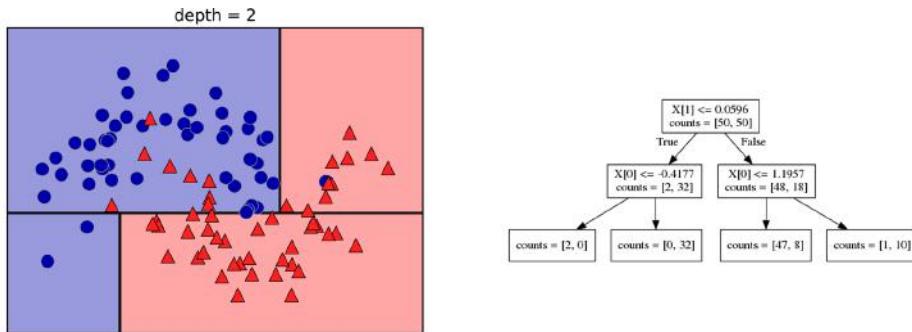


Figure 2-25. Decision boundary of tree with depth 2 (left) and corresponding decision tree (right)

This recursive process yields a binary tree of decisions, with each node containing a test. Alternatively, you can think of each test as splitting the part of the data that is currently being considered along one axis. This yields a view of the algorithm as building a hierarchical partition. As each test concerns only a single feature, the regions in the resulting partition always have axis-parallel boundaries.

The recursive partitioning of the data is repeated until each region in the partition (each leaf in the decision tree) only contains a single target value (a single class or a single regression value). A leaf of the tree that contains data points that all share the same target value is called *pure*. The final partitioning for this dataset is shown in Figure 2-26.

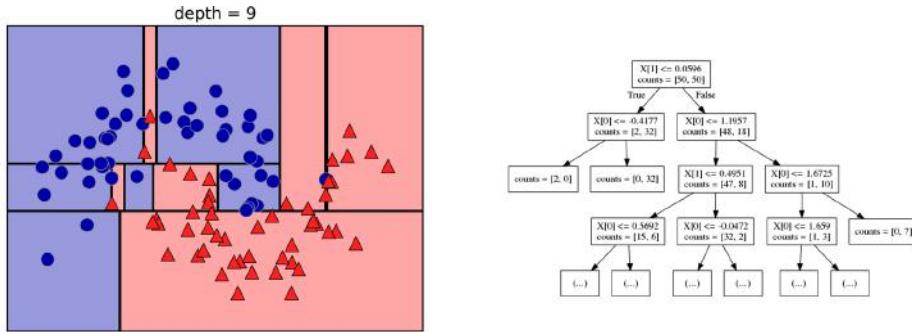


Figure 2-26. Decision boundary of tree with depth 9 (left) and part of the corresponding tree (right); the full tree is quite large and hard to visualize

A prediction on a new data point is made by checking which region of the partition of the feature space the point lies in, and then predicting the majority target (or the single target in the case of pure leaves) in that region. The region can be found by traversing the tree from the root and going left or right, depending on whether the test is fulfilled or not.

It is also possible to use trees for regression tasks, using exactly the same technique. To make a prediction, we traverse the tree based on the tests in each node and find the leaf the new data point falls into. The output for this data point is the mean target of the training points in this leaf.

Controlling complexity of decision trees

Typically, building a tree as described here and continuing until all leaves are pure leads to models that are very complex and highly overfit to the training data. The presence of pure leaves mean that a tree is 100% accurate on the training set; each data point in the training set is in a leaf that has the correct majority class. The overfitting can be seen on the left of Figure 2-26. You can see the regions determined to belong to class 1 in the middle of all the points belonging to class 0. On the other hand, there is a small strip predicted as class 0 around the point belonging to class 0 to the very right. This is not how one would imagine the decision boundary to look, and the decision boundary focuses a lot on single outlier points that are far away from the other points in that class.

There are two common strategies to prevent overfitting: stopping the creation of the tree early (also called *pre-pruning*), or building the tree but then removing or collapsing nodes that contain little information (also called *post-pruning* or just *pruning*). Possible criteria for pre-pruning include limiting the maximum depth of the tree, limiting the maximum number of leaves, or requiring a minimum number of points in a node to keep splitting it.

Decision trees in scikit-learn are implemented in the `DecisionTreeRegressor` and `DecisionTreeClassifier` classes. scikit-learn only implements pre-pruning, not post-pruning.

Let's look at the effect of pre-pruning in more detail on the Breast Cancer dataset. As always, we import the dataset and split it into a training and a test part. Then we build a model using the default setting of fully developing the tree (growing the tree until all leaves are pure). We fix the `random_state` in the tree, which is used for tie-breaking internally:

In[58]:

```
from sklearn.tree import DecisionTreeClassifier

cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(X_train, y_train)
print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

Out[58]:

```
Accuracy on training set: 1.000
Accuracy on test set: 0.937
```

As expected, the accuracy on the training set is 100%—because the leaves are pure, the tree was grown deep enough that it could perfectly memorize all the labels on the training data. The test set accuracy is slightly worse than for the linear models we looked at previously, which had around 95% accuracy.

If we don't restrict the depth of a decision tree, the tree can become arbitrarily deep and complex. Unpruned trees are therefore prone to overfitting and not generalizing well to new data. Now let's apply pre-pruning to the tree, which will stop developing the tree before we perfectly fit to the training data. One option is to stop building the tree after a certain depth has been reached. Here we set `max_depth=4`, meaning only four consecutive questions can be asked (cf. Figures 2-24 and 2-26). Limiting the depth of the tree decreases overfitting. This leads to a lower accuracy on the training set, but an improvement on the test set:

In[59]:

```
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(tree.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(X_test, y_test)))
```

Out[59]:

```
Accuracy on training set: 0.988
Accuracy on test set: 0.951
```

Analyzing decision trees

We can visualize the tree using the `export_graphviz` function from the `tree` module. This writes a file in the `.dot` file format, which is a text file format for storing graphs. We set an option to color the nodes to reflect the majority class in each node and pass the class and features names so the tree can be properly labeled:

In[61]:

```
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file="tree.dot", class_names=["malignant", "benign"],
                feature_names=cancer.feature_names, impurity=False, filled=True)
```

We can read this file and visualize it, as seen in [Figure 2-27](#), using the `graphviz` module (or you can use any program that can read `.dot` files):

In[61]:

```
import graphviz

with open("tree.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```

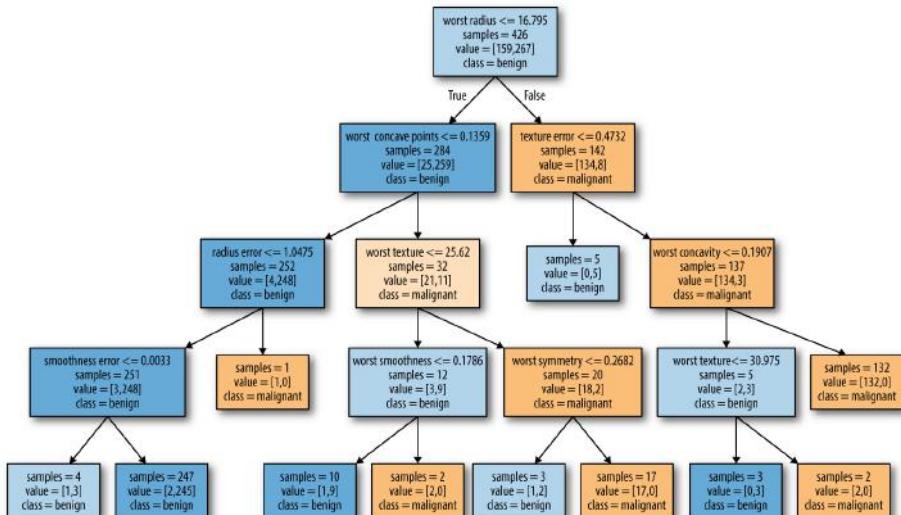


Figure 2-27. Visualization of the decision tree built on the Breast Cancer dataset

The visualization of the tree provides a great in-depth view of how the algorithm makes predictions, and is a good example of a machine learning algorithm that is easily explained to nonexperts. However, even with a tree of depth four, as seen here, the tree can become a bit overwhelming. Deeper trees (a depth of 10 is not uncommon) are even harder to grasp. One method of inspecting the tree that may be helpful is to find out which path most of the data actually takes. The `n_samples` shown in each node in [Figure 2-27](#) gives the number of samples in that node, while `value` provides the number of samples per class. Following the branches to the right, we see that `worst radius <= 16.795` creates a node that contains only 8 benign but 134 malignant samples. The rest of this side of the tree then uses some finer distinctions to split off these 8 remaining benign samples. Of the 142 samples that went to the right in the initial split, nearly all of them (132) end up in the leaf to the very right.

Taking a left at the root, for `worst radius > 16.795` we end up with 25 malignant and 259 benign samples. Nearly all of the benign samples end up in the second leaf from the right, with most of the other leaves containing very few samples.

Feature importance in trees

Instead of looking at the whole tree, which can be taxing, there are some useful properties that we can derive to summarize the workings of the tree. The most commonly used summary is *feature importance*, which rates how important each feature is for the decision a tree makes. It is a number between 0 and 1 for each feature, where 0 means “not used at all” and 1 means “perfectly predicts the target.” The feature importances always sum to 1:

In[62]:

```
print("Feature importances:\n{}".format(tree.feature_importances_))
```

Out[62]:

```
Feature importances:  
[ 0.        0.        0.        0.        0.        0.        0.        0.        0.        0.        0.01  
 0.048     0.        0.        0.002    0.        0.        0.        0.        0.        0.727    0.046  
 0.        0.        0.014    0.        0.018    0.122    0.012    0.        ]
```

We can visualize the feature importances in a way that is similar to the way we visualize the coefficients in the linear model ([Figure 2-28](#)):

In[63]:

```
def plot_feature_importances_cancer(model):  
    n_features = cancer.data.shape[1]  
    plt.barh(range(n_features), model.feature_importances_, align='center')  
    plt.yticks(np.arange(n_features), cancer.feature_names)  
    plt.xlabel("Feature importance")  
    plt.ylabel("Feature")  
  
plot_feature_importances_cancer(tree)
```

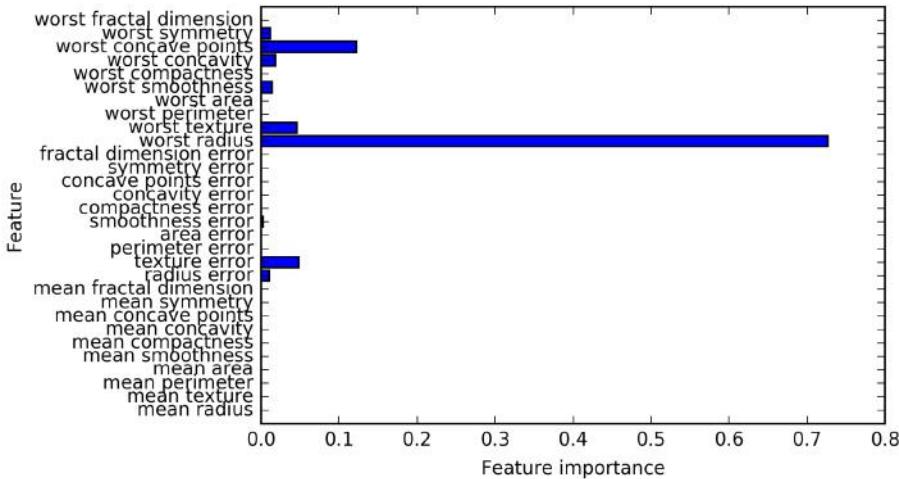


Figure 2-28. Feature importances computed from a decision tree learned on the Breast Cancer dataset

Here we see that the feature used in the top split (“worst radius”) is by far the most important feature. This confirms our observation in analyzing the tree that the first level already separates the two classes fairly well.

However, if a feature has a low `feature_importance`, it doesn’t mean that this feature is uninformative. It only means that the feature was not picked by the tree, likely because another feature encodes the same information.

In contrast to the coefficients in linear models, feature importances are always positive, and don’t encode which class a feature is indicative of. The feature importances tell us that “worst radius” is important, but not whether a high radius is indicative of a sample being benign or malignant. In fact, there might not be such a simple relationship between features and class, as you can see in the following example (Figures 2-29 and 2-30):

In[64]:

```
tree = mglearn.plots.plot_tree_not_monotone()
display(tree)
```

Out[64]:

```
Feature importances: [ 0.  1.]
```

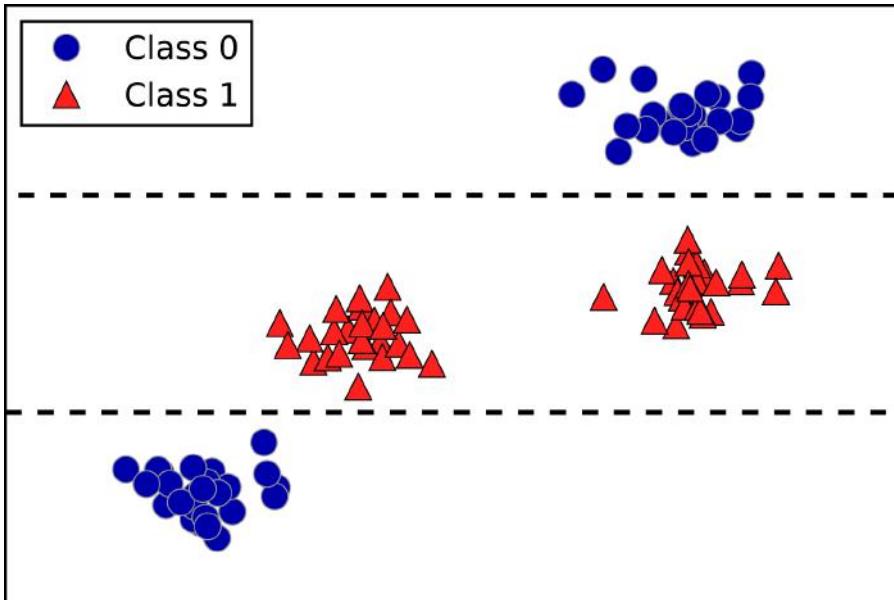


Figure 2-29. A two-dimensional dataset in which the feature on the y-axis has a nonmonotonic relationship with the class label, and the decision boundaries found by a decision tree

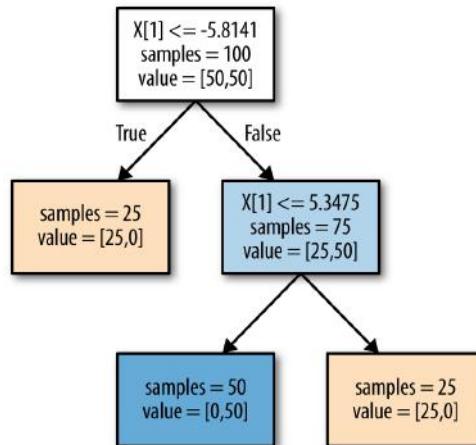


Figure 2-30. Decision tree learned on the data shown in Figure 2-29

The plot shows a dataset with two features and two classes. Here, all the information is contained in $X[1]$, and $X[0]$ is not used at all. But the relation between $X[1]$ and

the output class is not monotonous, meaning we cannot say “a high value of $X[0]$ means class 0, and a low value means class 1” (or vice versa).

While we focused our discussion here on decision trees for classification, all that was said is similarly true for decision trees for regression, as implemented in `DecisionTreeRegressor`. The usage and analysis of regression trees is very similar to that of classification trees. There is one particular property of using tree-based models for regression that we want to point out, though. The `DecisionTreeRegressor` (and all other tree-based regression models) is not able to *extrapolate*, or make predictions outside of the range of the training data.

Let’s look into this in more detail, using a dataset of historical computer memory (RAM) prices. Figure 2-31 shows the dataset, with the date on the x-axis and the price of one megabyte of RAM in that year on the y-axis:

In[65]:

```
import pandas as pd
ram_prices = pd.read_csv("data/ram_price.csv")

plt.semilogy(ram_prices.date, ram_prices.price)
plt.xlabel("Year")
plt.ylabel("Price in $/Mbyte")
```

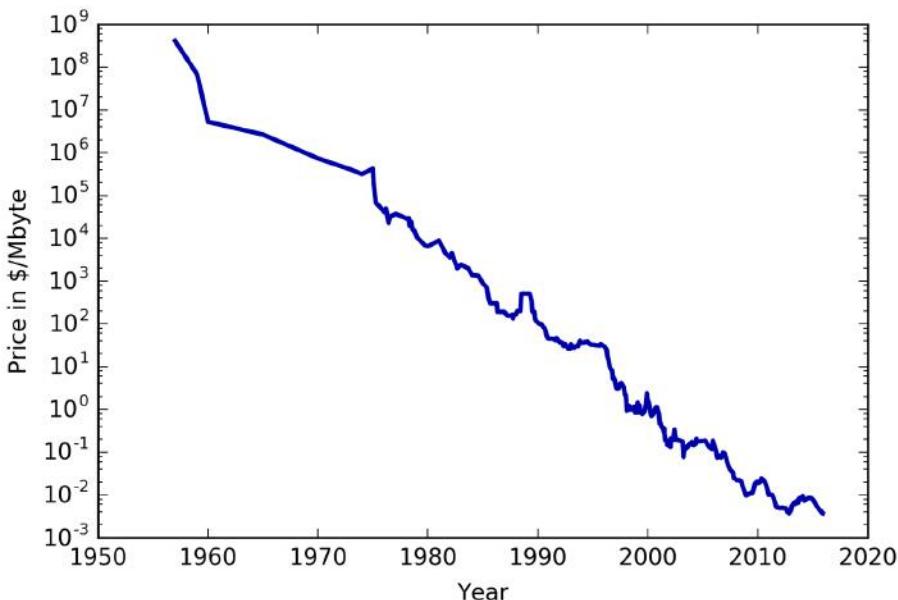


Figure 2-31. Historical development of the price of RAM, plotted on a log scale

Note the logarithmic scale of the y-axis. When plotting logarithmically, the relation seems to be quite linear and so should be relatively easy to predict, apart from some bumps.

We will make a forecast for the years after 2000 using the historical data up to that point, with the date as our only feature. We will compare two simple models: a `DecisionTreeRegressor` and `LinearRegression`. We rescale the prices using a logarithm, so that the relationship is relatively linear. This doesn't make a difference for the `DecisionTreeRegressor`, but it makes a big difference for `LinearRegression` (we will discuss this in more depth in [Chapter 4](#)). After training the models and making predictions, we apply the exponential map to undo the logarithm transform. We make predictions on the whole dataset for visualization purposes here, but for a quantitative evaluation we would only consider the test dataset:

In[66]:

```
from sklearn.tree import DecisionTreeRegressor
# use historical data to forecast prices after the year 2000
data_train = ram_prices[ram_prices.date < 2000]
data_test = ram_prices[ram_prices.date >= 2000]

# predict prices based on date
X_train = data_train.date[:, np.newaxis]
# we use a log-transform to get a simpler relationship of data to target
y_train = np.log(data_train.price)

tree = DecisionTreeRegressor().fit(X_train, y_train)
linear_reg = LinearRegression().fit(X_train, y_train)

# predict on all data
X_all = ram_prices.date[:, np.newaxis]

pred_tree = tree.predict(X_all)
pred_lr = linear_reg.predict(X_all)

# undo log-transform
price_tree = np.exp(pred_tree)
price_lr = np.exp(pred_lr)
```

[Figure 2-32](#), created here, compares the predictions of the decision tree and the linear regression model with the ground truth:

In[67]:

```
plt.semilogy(data_train.date, data_train.price, label="Training data")
plt.semilogy(data_test.date, data_test.price, label="Test data")
plt.semilogy(ram_prices.date, price_tree, label="Tree prediction")
plt.semilogy(ram_prices.date, price_lr, label="Linear prediction")
plt.legend()
```

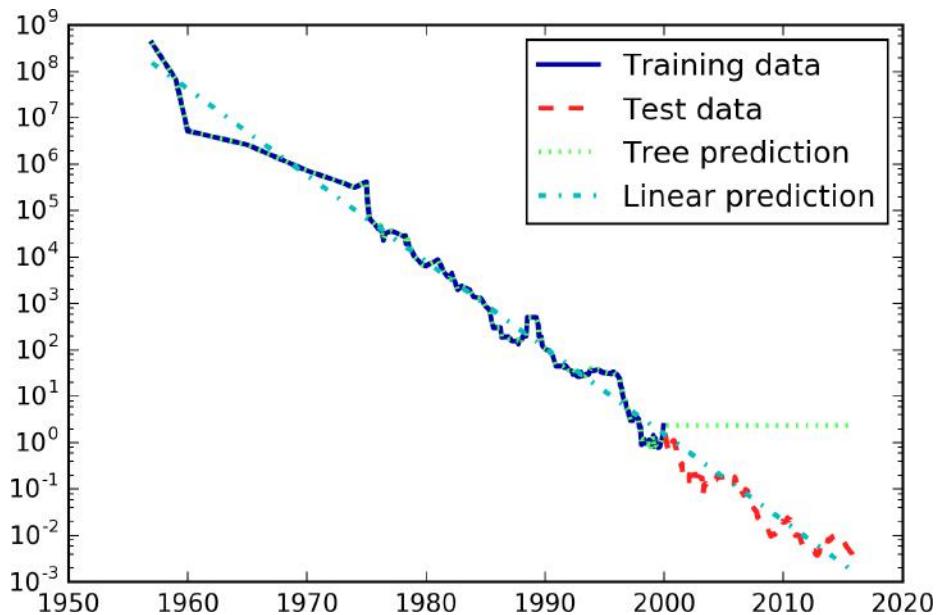


Figure 2-32. Comparison of predictions made by a linear model and predictions made by a regression tree on the RAM price data

The difference between the models is quite striking. The linear model approximates the data with a line, as we knew it would. This line provides quite a good forecast for the test data (the years after 2000), while glossing over some of the finer variations in both the training and the test data. The tree model, on the other hand, makes perfect predictions on the training data; we did not restrict the complexity of the tree, so it learned the whole dataset by heart. However, once we leave the data range for which the model has data, the model simply keeps predicting the last known point. The tree has no ability to generate “new” responses, outside of what was seen in the training data. This shortcoming applies to all models based on trees.⁹

Strengths, weaknesses, and parameters

As discussed earlier, the parameters that control model complexity in decision trees are the pre-pruning parameters that stop the building of the tree before it is fully developed. Usually, picking one of the pre-pruning strategies—setting either

⁹ It is actually possible to make very good forecasts with tree-based models (for example, when trying to predict whether a price will go up or down). The point of this example was not to show that trees are a bad model for time series, but to illustrate a particular property of how trees make predictions.

`max_depth`, `max_leaf_nodes`, or `min_samples_leaf`—is sufficient to prevent overfitting.

Decision trees have two advantages over many of the algorithms we've discussed so far: the resulting model can easily be visualized and understood by nonexperts (at least for smaller trees), and the algorithms are completely invariant to scaling of the data. As each feature is processed separately, and the possible splits of the data don't depend on scaling, no preprocessing like normalization or standardization of features is needed for decision tree algorithms. In particular, decision trees work well when you have features that are on completely different scales, or a mix of binary and continuous features.

The main downside of decision trees is that even with the use of pre-pruning, they tend to overfit and provide poor generalization performance. Therefore, in most applications, the ensemble methods we discuss next are usually used in place of a single decision tree.

Ensembles of Decision Trees

Ensembles are methods that combine multiple machine learning models to create more powerful models. There are many models in the machine learning literature that belong to this category, but there are two ensemble models that have proven to be effective on a wide range of datasets for classification and regression, both of which use decision trees as their building blocks: random forests and gradient boosted decision trees.

Random forests

As we just observed, a main drawback of decision trees is that they tend to overfit the training data. Random forests are one way to address this problem. A random forest is essentially a collection of decision trees, where each tree is slightly different from the others. The idea behind random forests is that each tree might do a relatively good job of predicting, but will likely overfit on part of the data. If we build many trees, all of which work well and overfit in different ways, we can reduce the amount of overfitting by averaging their results. This reduction in overfitting, while retaining the predictive power of the trees, can be shown using rigorous mathematics.

To implement this strategy, we need to build many decision trees. Each tree should do an acceptable job of predicting the target, and should also be different from the other trees. Random forests get their name from injecting randomness into the tree building to ensure each tree is different. There are two ways in which the trees in a random forest are randomized: by selecting the data points used to build a tree and by selecting the features in each split test. Let's go into this process in more detail.

Building random forests. To build a random forest model, you need to decide on the number of trees to build (the `n_estimators` parameter of `RandomForestRegressor` or `RandomForestClassifier`). Let's say we want to build 10 trees. These trees will be built completely independently from each other, and the algorithm will make different random choices for each tree to make sure the trees are distinct. To build a tree, we first take what is called a *bootstrap sample* of our data. That is, from our `n_samples` data points, we repeatedly draw an example randomly with replacement (meaning the same sample can be picked multiple times), `n_samples` times. This will create a dataset that is as big as the original dataset, but some data points will be missing from it (approximately one third), and some will be repeated.

To illustrate, let's say we want to create a bootstrap sample of the list `['a', 'b', 'c', 'd']`. A possible bootstrap sample would be `['b', 'd', 'd', 'c']`. Another possible sample would be `['d', 'a', 'd', 'a']`.

Next, a decision tree is built based on this newly created dataset. However, the algorithm we described for the decision tree is slightly modified. Instead of looking for the best test for each node, in each node the algorithm randomly selects a subset of the features, and it looks for the best possible test involving one of these features. The number of features that are selected is controlled by the `max_features` parameter. This selection of a subset of features is repeated separately in each node, so that each node in a tree can make a decision using a different subset of the features.

The bootstrap sampling leads to each decision tree in the random forest being built on a slightly different dataset. Because of the selection of features in each node, each split in each tree operates on a different subset of features. Together, these two mechanisms ensure that all the trees in the random forest are different.

A critical parameter in this process is `max_features`. If we set `max_features` to `n_features`, that means that each split can look at all features in the dataset, and no randomness will be injected in the feature selection (the randomness due to the bootstrapping remains, though). If we set `max_features` to 1, that means that the splits have no choice at all on which feature to test, and can only search over different thresholds for the feature that was selected randomly. Therefore, a high `max_features` means that the trees in the random forest will be quite similar, and they will be able to fit the data easily, using the most distinctive features. A low `max_features` means that the trees in the random forest will be quite different, and that each tree might need to be very deep in order to fit the data well.

To make a prediction using the random forest, the algorithm first makes a prediction for every tree in the forest. For regression, we can average these results to get our final prediction. For classification, a “soft voting” strategy is used. This means each algorithm makes a “soft” prediction, providing a probability for each possible output

label. The probabilities predicted by all the trees are averaged, and the class with the highest probability is predicted.

Analyzing random forests. Let's apply a random forest consisting of five trees to the `two_moons` dataset we studied earlier:

In[68]:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    random_state=42)

forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(X_train, y_train)
```

The trees that are built as part of the random forest are stored in the `estimator_` attribute. Let's visualize the decision boundaries learned by each tree, together with their aggregate prediction as made by the forest (Figure 2-33):

In[69]:

```
fig, axes = plt.subplots(2, 3, figsize=(20, 10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
    ax.set_title("Tree {}".format(i))
    mglearn.plots.plot_tree_partition(X_train, y_train, tree, ax=ax)

mglearn.plots.plot_2d_separator(forest, X_train, fill=True, ax=axes[-1, -1],
                                alpha=.4)
axes[-1, -1].set_title("Random Forest")
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
```

You can clearly see that the decision boundaries learned by the five trees are quite different. Each of them makes some mistakes, as some of the training points that are plotted here were not actually included in the training sets of the trees, due to the bootstrap sampling.

The random forest overfits less than any of the trees individually, and provides a much more intuitive decision boundary. In any real application, we would use many more trees (often hundreds or thousands), leading to even smoother boundaries.

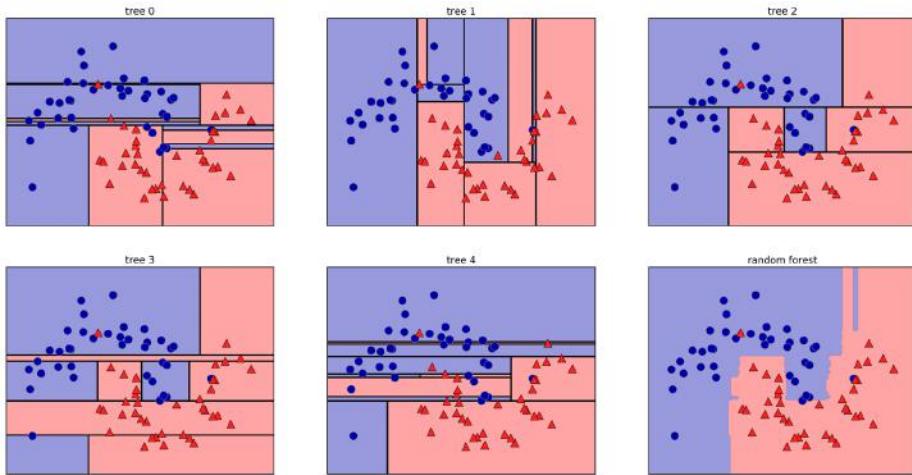


Figure 2-33. Decision boundaries found by five randomized decision trees and the decision boundary obtained by averaging their predicted probabilities

As another example, let's apply a random forest consisting of 100 trees on the Breast Cancer dataset:

In[70]:

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(forest.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(forest.score(X_test, y_test)))
```

Out[70]:

```
Accuracy on training set: 1.000
Accuracy on test set: 0.972
```

The random forest gives us an accuracy of 97%, better than the linear models or a single decision tree, without tuning any parameters. We could adjust the `max_features` setting, or apply pre-pruning as we did for the single decision tree. However, often the default parameters of the random forest already work quite well.

Similarly to the decision tree, the random forest provides feature importances, which are computed by aggregating the feature importances over the trees in the forest. Typically, the feature importances provided by the random forest are more reliable than the ones provided by a single tree. Take a look at [Figure 2-34](#).

In[71]:

```
plot_feature_importances_cancer(forest)
```

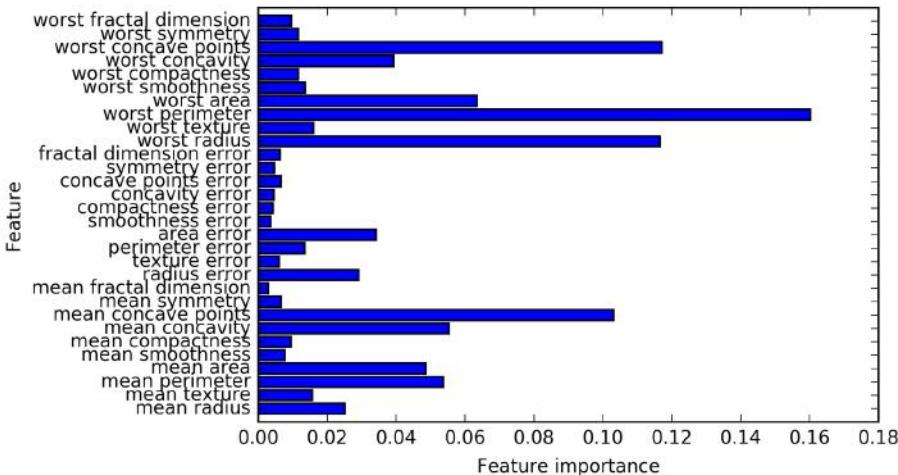


Figure 2-34. Feature importances computed from a random forest that was fit to the Breast Cancer dataset

As you can see, the random forest gives nonzero importance to many more features than the single tree. Similarly to the single decision tree, the random forest also gives a lot of importance to the “worst radius” feature, but it actually chooses “worst perimeter” to be the most informative feature overall. The randomness in building the random forest forces the algorithm to consider many possible explanations, the result being that the random forest captures a much broader picture of the data than a single tree.

Strengths, weaknesses, and parameters. Random forests for regression and classification are currently among the most widely used machine learning methods. They are very powerful, often work well without heavy tuning of the parameters, and don’t require scaling of the data.

Essentially, random forests share all of the benefits of decision trees, while making up for some of their deficiencies. One reason to still use decision trees is if you need a compact representation of the decision-making process. It is basically impossible to interpret tens or hundreds of trees in detail, and trees in random forests tend to be deeper than decision trees (because of the use of feature subsets). Therefore, if you need to summarize the prediction making in a visual way to nonexperts, a single decision tree might be a better choice. While building random forests on large datasets might be somewhat time consuming, it can be parallelized across multiple CPU

cores within a computer easily. If you are using a multi-core processor (as nearly all modern computers do), you can use the `n_jobs` parameter to adjust the number of cores to use. Using more CPU cores will result in linear speed-ups (using two cores, the training of the random forest will be twice as fast), but specifying `n_jobs` larger than the number of cores will not help. You can set `n_jobs=-1` to use all the cores in your computer.

You should keep in mind that random forests, by their nature, are random, and setting different random states (or not setting the `random_state` at all) can drastically change the model that is built. The more trees there are in the forest, the more robust it will be against the choice of random state. If you want to have reproducible results, it is important to fix the `random_state`.

Random forests don't tend to perform well on very high dimensional, sparse data, such as text data. For this kind of data, linear models might be more appropriate. Random forests usually work well even on very large datasets, and training can easily be parallelized over many CPU cores within a powerful computer. However, random forests require more memory and are slower to train and to predict than linear models. If time and memory are important in an application, it might make sense to use a linear model instead.

The important parameters to adjust are `n_estimators`, `max_features`, and possibly pre-pruning options like `max_depth`. For `n_estimators`, larger is always better. Averaging more trees will yield a more robust ensemble by reducing overfitting. However, there are diminishing returns, and more trees need more memory and more time to train. A common rule of thumb is to build “as many as you have time/memory for.”

As described earlier, `max_features` determines how random each tree is, and a smaller `max_features` reduces overfitting. In general, it's a good rule of thumb to use the default values: `max_features=sqrt(n_features)` for classification and `max_features=log2(n_features)` for regression. Adding `max_features` or `max_leaf_nodes` might sometimes improve performance. It can also drastically reduce space and time requirements for training and prediction.

Gradient boosted regression trees (gradient boosting machines)

The gradient boosted regression tree is another ensemble method that combines multiple decision trees to create a more powerful model. Despite the “regression” in the name, these models can be used for regression and classification. In contrast to the random forest approach, gradient boosting works by building trees in a serial manner, where each tree tries to correct the mistakes of the previous one. By default, there is no randomization in gradient boosted regression trees; instead, strong pre-pruning is used. Gradient boosted trees often use very shallow trees, of depth one to five, which makes the model smaller in terms of memory and makes predictions faster.

The main idea behind gradient boosting is to combine many simple models (in this context known as *weak learners*), like shallow trees. Each tree can only provide good predictions on part of the data, and so more and more trees are added to iteratively improve performance.

Gradient boosted trees are frequently the winning entries in machine learning competitions, and are widely used in industry. They are generally a bit more sensitive to parameter settings than random forests, but can provide better accuracy if the parameters are set correctly.

Apart from the pre-pruning and the number of trees in the ensemble, another important parameter of gradient boosting is the `learning_rate`, which controls how strongly each tree tries to correct the mistakes of the previous trees. A higher learning rate means each tree can make stronger corrections, allowing for more complex models. Adding more trees to the ensemble, which can be accomplished by increasing `n_estimators`, also increases the model complexity, as the model has more chances to correct mistakes on the training set.

Here is an example of using `GradientBoostingClassifier` on the Breast Cancer dataset. By default, 100 trees of maximum depth 3 and a learning rate of 0.1 are used:

In[72]:

```
from sklearn.ensemble import GradientBoostingClassifier

X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

gbdt = GradientBoostingClassifier(random_state=0)
gbdt.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(gbdt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbdt.score(X_test, y_test)))
```

Out[72]:

```
Accuracy on training set: 1.000
Accuracy on test set: 0.958
```

As the training set accuracy is 100%, we are likely to be overfitting. To reduce overfitting, we could either apply stronger pre-pruning by limiting the maximum depth or lower the learning rate:

In[73]:

```
gbrt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbrt.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(gbdt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbdt.score(X_test, y_test)))
```

Out[73]:

```
Accuracy on training set: 0.991
Accuracy on test set: 0.972
```

In[74]:

```
gbdt = GradientBoostingClassifier(random_state=0, learning_rate=0.01)
gbdt.fit(X_train, y_train)

print("Accuracy on training set: {:.3f}".format(gbdt.score(X_train, y_train)))
print("Accuracy on test set: {:.3f}".format(gbdt.score(X_test, y_test)))
```

Out[74]:

```
Accuracy on training set: 0.988
Accuracy on test set: 0.965
```

Both methods of decreasing the model complexity reduced the training set accuracy, as expected. In this case, lowering the maximum depth of the trees provided a significant improvement of the model, while lowering the learning rate only increased the generalization performance slightly.

As for the other decision tree-based models, we can again visualize the feature importances to get more insight into our model (Figure 2-35). As we used 100 trees, it is impractical to inspect them all, even if they are all of depth 1:

In[75]:

```
gbdt = GradientBoostingClassifier(random_state=0, max_depth=1)
gbdt.fit(X_train, y_train)

plot_feature_importances_cancer(gbdt)
```

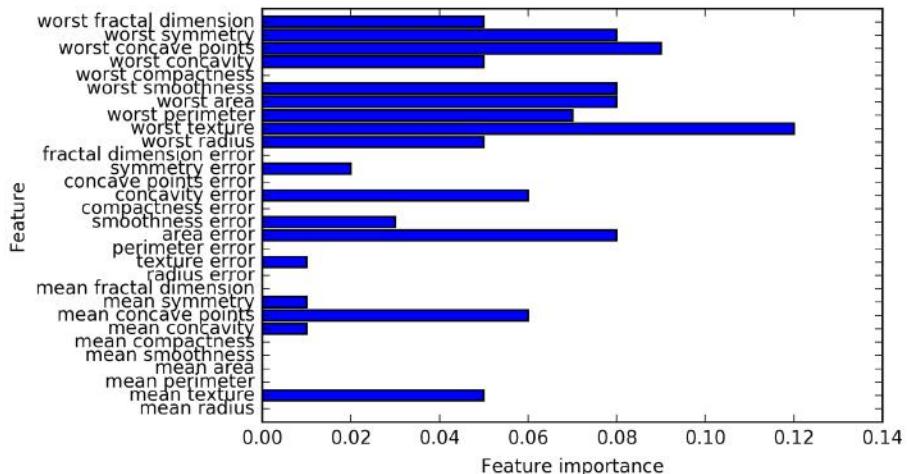


Figure 2-35. Feature importances computed from a gradient boosting classifier that was fit to the Breast Cancer dataset

We can see that the feature importances of the gradient boosted trees are somewhat similar to the feature importances of the random forests, though the gradient boosting completely ignored some of the features.

As both gradient boosting and random forests perform well on similar kinds of data, a common approach is to first try random forests, which work quite robustly. If random forests work well but prediction time is at a premium, or it is important to squeeze out the last percentage of accuracy from the machine learning model, moving to gradient boosting often helps.

If you want to apply gradient boosting to a large-scale problem, it might be worth looking into the `xgboost` package and its Python interface, which at the time of writing is faster (and sometimes easier to tune) than the `scikit-learn` implementation of gradient boosting on many datasets.

Strengths, weaknesses, and parameters. Gradient boosted decision trees are among the most powerful and widely used models for supervised learning. Their main drawback is that they require careful tuning of the parameters and may take a long time to train. Similarly to other tree-based models, the algorithm works well without scaling and on a mixture of binary and continuous features. As with other tree-based models, it also often does not work well on high-dimensional sparse data.

The main parameters of gradient boosted tree models are the number of trees, `n_estimators`, and the `learning_rate`, which controls the degree to which each tree is allowed to correct the mistakes of the previous trees. These two parameters are highly

interconnected, as a lower `learning_rate` means that more trees are needed to build a model of similar complexity. In contrast to random forests, where a higher `n_estimators` value is always better, increasing `n_estimators` in gradient boosting leads to a more complex model, which may lead to overfitting. A common practice is to fit `n_estimators` depending on the time and memory budget, and then search over different `learning_rates`.

Another important parameter is `max_depth` (or alternatively `max_leaf_nodes`), to reduce the complexity of each tree. Usually `max_depth` is set very low for gradient boosted models, often not deeper than five splits.

Kernelized Support Vector Machines

The next type of supervised model we will discuss is kernelized support vector machines. We explored the use of linear support vector machines for classification in “[Linear models for classification](#)” on page 56. Kernelized support vector machines (often just referred to as SVMs) are an extension that allows for more complex models that are not defined simply by hyperplanes in the input space. While there are support vector machines for classification and regression, we will restrict ourselves to the classification case, as implemented in `SVC`. Similar concepts apply to support vector regression, as implemented in `SVR`.

The math behind kernelized support vector machines is a bit involved, and is beyond the scope of this book. You can find the details in Chapter 1 of Hastie, Tibshirani, and Friedman’s *The Elements of Statistical Learning*. However, we will try to give you some sense of the idea behind the method.

Linear models and nonlinear features

As you saw in [Figure 2-15](#), linear models can be quite limiting in low-dimensional spaces, as lines and hyperplanes have limited flexibility. One way to make a linear model more flexible is by adding more features—for example, by adding interactions or polynomials of the input features.

Let’s look at the synthetic dataset we used in “[Feature importance in trees](#)” on page 77 (see [Figure 2-29](#)):

In[76]:

```
X, y = make_blobs(centers=4, random_state=8)
y = y % 2

mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

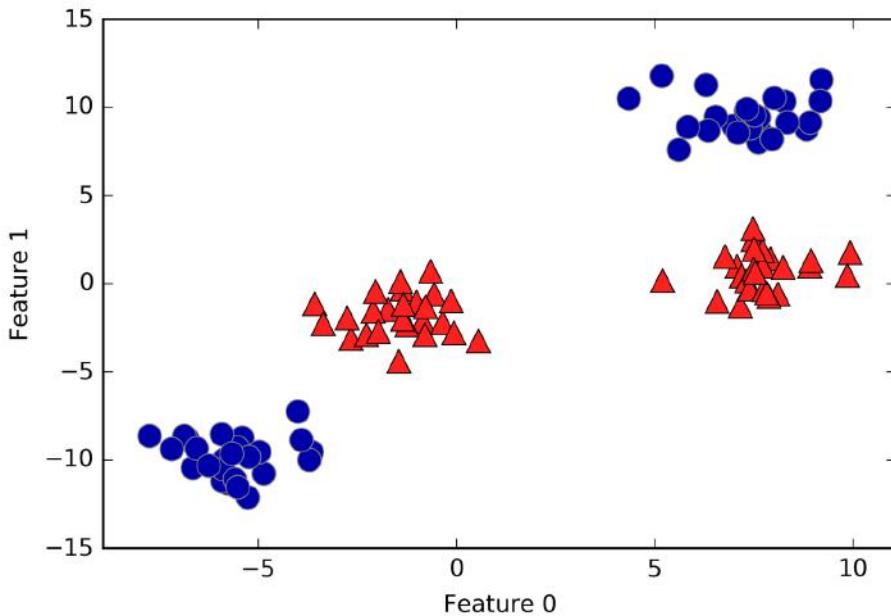


Figure 2-36. Two-class classification dataset in which classes are not linearly separable

A linear model for classification can only separate points using a line, and will not be able to do a very good job on this dataset (see [Figure 2-37](#)):

In[77]:

```
from sklearn.svm import LinearSVC
linear_svm = LinearSVC().fit(X, y)

mglearn.plots.plot_2d_separator(linear_svm, X)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

Now let's expand the set of input features, say by also adding `feature1 ** 2`, the square of the second feature, as a new feature. Instead of representing each data point as a two-dimensional point, (`feature0, feature1`), we now represent it as a three-dimensional point, (`feature0, feature1, feature1 ** 2`).¹⁰ This new representation is illustrated in [Figure 2-38](#) in a three-dimensional scatter plot:

¹⁰ We picked this particular feature to add for illustration purposes. The choice is not particularly important.

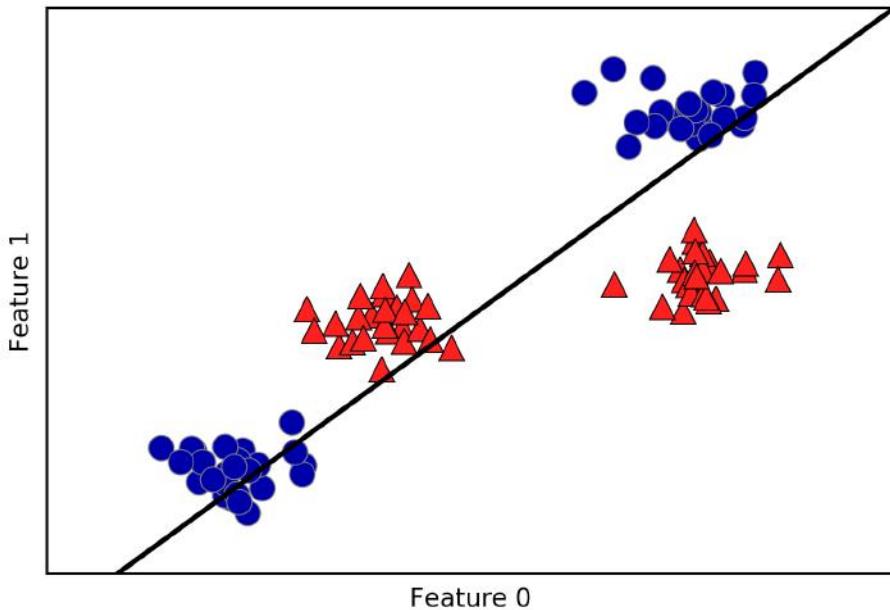


Figure 2-37. Decision boundary found by a linear SVM

In[78]:

```
# add the squared first feature
X_new = np.hstack([X, X[:, 1:] ** 2])

from mpl_toolkits.mplot3d import Axes3D, axes3d
figure = plt.figure()
# visualize in 3D
ax = Axes3D(figure, elev=-152, azim=-26)
# plot first all the points with y == 0, then all with y == 1
mask = y == 0
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b',
           cmap=mglearn.cm2, s=60)
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^',
           cmap=mglearn.cm2, s=60)
ax.set_xlabel("feature0")
ax.set_ylabel("feature1")
ax.set_zlabel("feature1 ** 2")
```

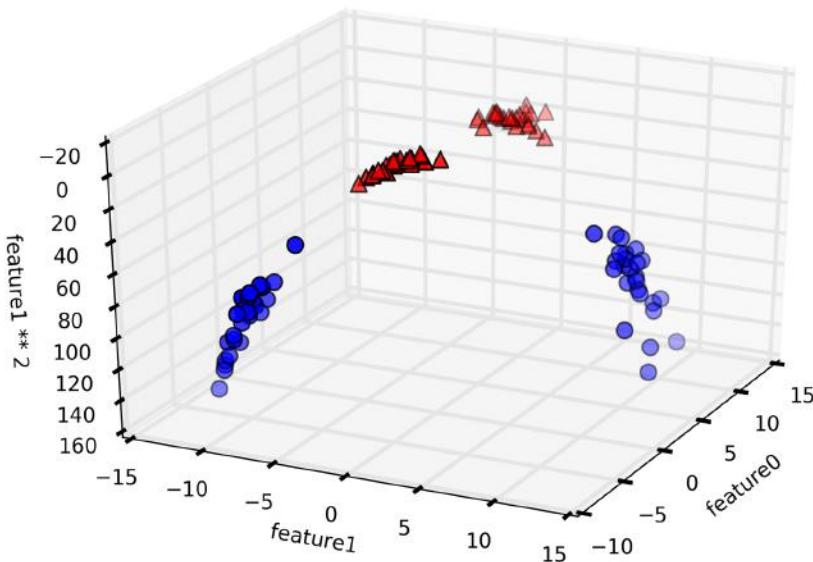


Figure 2-38. Expansion of the dataset shown in Figure 2-37, created by adding a third feature derived from feature1

In the new representation of the data, it is now indeed possible to separate the two classes using a linear model, a plane in three dimensions. We can confirm this by fitting a linear model to the augmented data (see Figure 2-39):

In[79]:

```
linear_svm_3d = LinearSVC().fit(X_new, y)
coef, intercept = linear_svm_3d.coef_.ravel(), linear_svm_3d.intercept_

# show linear decision boundary
figure = plt.figure()
ax = Axes3D(figure, elev=-152, azim=-26)
xx = np.linspace(X_new[:, 0].min() - 2, X_new[:, 0].max() + 2, 50)
yy = np.linspace(X_new[:, 1].min() - 2, X_new[:, 1].max() + 2, 50)

XX, YY = np.meshgrid(xx, yy)
ZZ = (coef[0] * XX + coef[1] * YY + intercept) / -coef[2]
ax.plot_surface(XX, YY, ZZ, rstride=8, cstride=8, alpha=0.3)
ax.scatter(X_new[mask, 0], X_new[mask, 1], X_new[mask, 2], c='b',
           cmap=mpl.cm.cm2, s=60)
ax.scatter(X_new[~mask, 0], X_new[~mask, 1], X_new[~mask, 2], c='r', marker='^',
           cmap=mpl.cm.cm2, s=60)

ax.set_xlabel("feature0")
ax.set_ylabel("feature1")
ax.set_zlabel("feature0 ** 2")
```

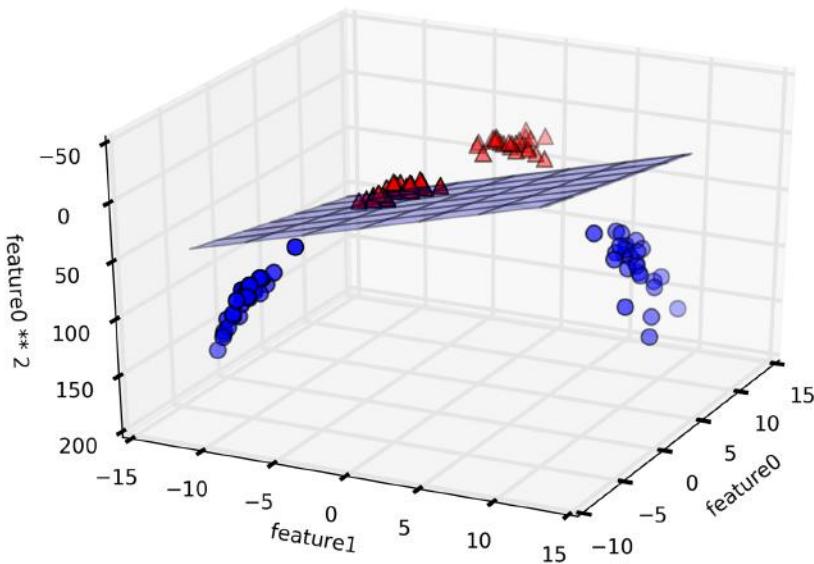


Figure 2-39. Decision boundary found by a linear SVM on the expanded three-dimensional dataset

As a function of the original features, the linear SVM model is not actually linear anymore. It is not a line, but more of an ellipse, as you can see from the plot created here (Figure 2-40):

In[80]:

```
ZZ = YY ** 2
dec = linear_svm_3d.decision_function(np.c_[XX.ravel(), YY.ravel(), ZZ.ravel()])
plt.contourf(XX, YY, dec.reshape(XX.shape), levels=[dec.min(), 0, dec.max()],
             cmap=mglearn.cm2, alpha=0.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

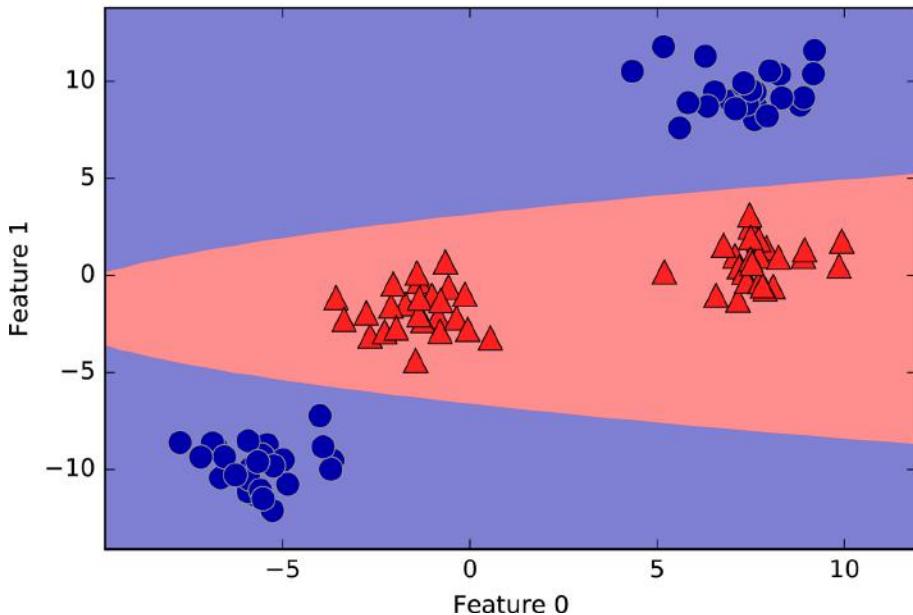


Figure 2-40. The decision boundary from Figure 2-39 as a function of the original two features

The kernel trick

The lesson here is that adding nonlinear features to the representation of our data can make linear models much more powerful. However, often we don't know which features to add, and adding many features (like all possible interactions in a 100-dimensional feature space) might make computation very expensive. Luckily, there is a clever mathematical trick that allows us to learn a classifier in a higher-dimensional space without actually computing the new, possibly very large representation. This is known as the *kernel trick*, and it works by directly computing the distance (more precisely, the scalar products) of the data points for the expanded feature representation, without ever actually computing the expansion.

There are two ways to map your data into a higher-dimensional space that are commonly used with support vector machines: the polynomial kernel, which computes all possible polynomials up to a certain degree of the original features (like `feature1 ** 2 * feature2 ** 5`); and the radial basis function (RBF) kernel, also known as the Gaussian kernel. The Gaussian kernel is a bit harder to explain, as it corresponds to an infinite-dimensional feature space. One way to explain the Gaussian kernel is that

it considers all possible polynomials of all degrees, but the importance of the features decreases for higher degrees.¹¹

In practice, the mathematical details behind the kernel SVM are not that important, though, and how an SVM with an RBF kernel makes a decision can be summarized quite easily—we'll do so in the next section.

Understanding SVMs

During training, the SVM learns how important each of the training data points is to represent the decision boundary between the two classes. Typically only a subset of the training points matter for defining the decision boundary: the ones that lie on the border between the classes. These are called *support vectors* and give the support vector machine its name.

To make a prediction for a new point, the distance to each of the support vectors is measured. A classification decision is made based on the distances to the support vector, and the importance of the support vectors that was learned during training (stored in the `dual_coef_` attribute of SVC).

The distance between data points is measured by the Gaussian kernel:

$$k_{\text{rbf}}(x_1, x_2) = \exp(-\gamma \|x_1 - x_2\|^2)$$

Here, x_1 and x_2 are data points, $\|x_1 - x_2\|$ denotes Euclidean distance, and γ (gamma) is a parameter that controls the width of the Gaussian kernel.

Figure 2-41 shows the result of training a support vector machine on a two-dimensional two-class dataset. The decision boundary is shown in black, and the support vectors are larger points with the wide outline. The following code creates this plot by training an SVM on the forge dataset:

In[81]:

```
from sklearn.svm import SVC
X, y = mglearn.tools.make_handcrafted_dataset()
svm = SVC(kernel='rbf', C=10, gamma=0.1).fit(X, y)
mglearn.plots.plot_2d_separator(svm, X, eps=.5)
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
# plot support vectors
sv = svm.support_vectors_
# class labels of support vectors are given by the sign of the dual coefficients
sv_labels = svm.dual_coef_.ravel() > 0
mglearn.discrete_scatter(sv[:, 0], sv[:, 1], sv_labels, s=15, markeredgewidth=3)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

¹¹ This follows from the Taylor expansion of the exponential map.

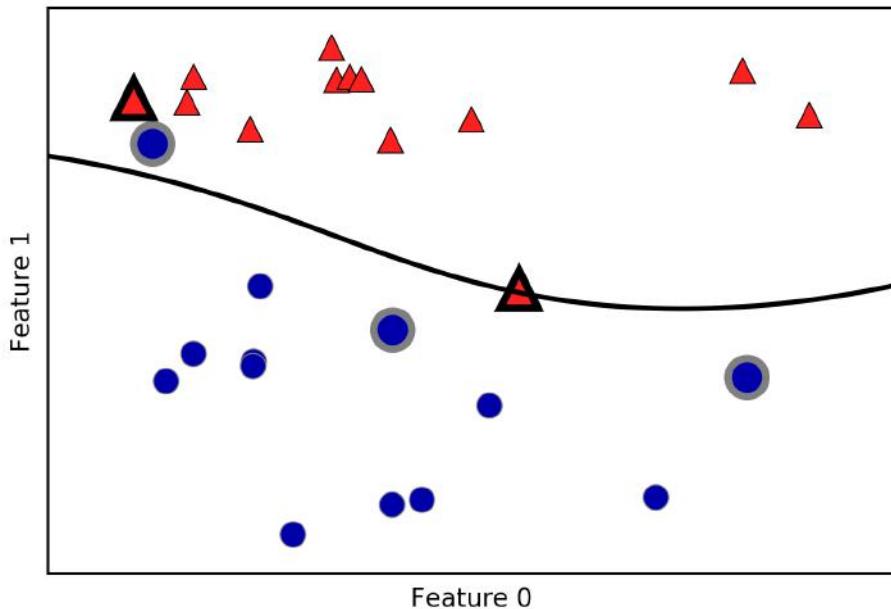


Figure 2-41. Decision boundary and support vectors found by an SVM with RBF kernel

In this case, the SVM yields a very smooth and nonlinear (not a straight line) boundary. We adjusted two parameters here: the `C` parameter and the `gamma` parameter, which we will now discuss in detail.

Tuning SVM parameters

The `gamma` parameter is the one shown in the formula given in the previous section, which controls the width of the Gaussian kernel. It determines the scale of what it means for points to be close together. The `C` parameter is a regularization parameter, similar to that used in the linear models. It limits the importance of each point (or more precisely, their `dual_coef_`).

Let's have a look at what happens when we vary these parameters (Figure 2-42):

In[82]:

```
fig, axes = plt.subplots(3, 3, figsize=(15, 10))

for ax, C in zip(axes, [-1, 0, 3]):
    for a, gamma in zip(ax, range(-1, 2)):
        mlearn.plots.plot_svm(log_C=C, log_gamma=gamma, ax=a)

axes[0, 0].legend(["class 0", "class 1", "sv class 0", "sv class 1"],
                  ncol=4, loc=(.9, 1.2))
```

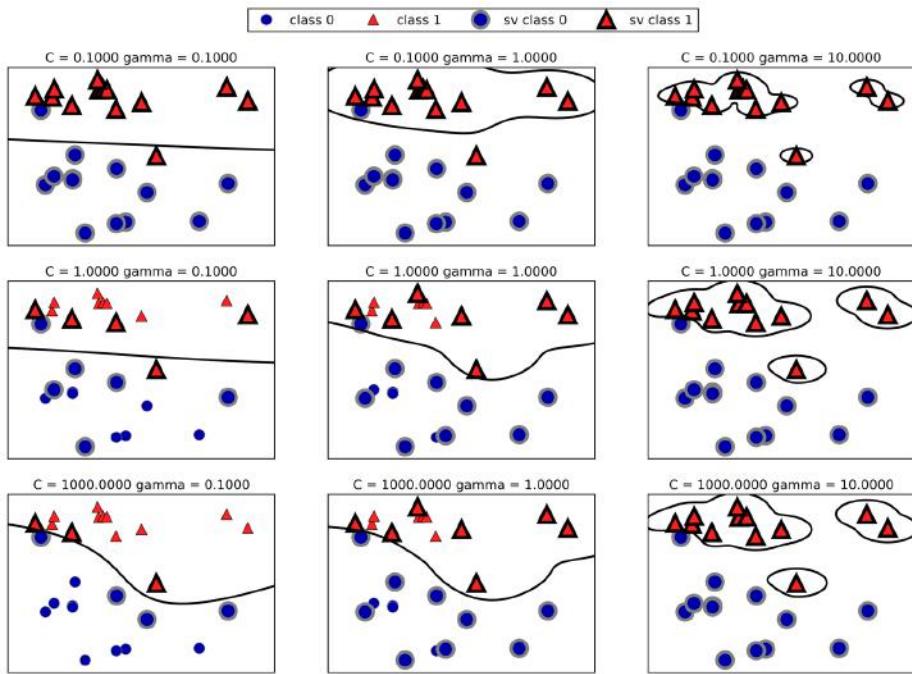


Figure 2-42. Decision boundaries and support vectors for different settings of the parameters C and γ

Going from left to right, we increase the value of the parameter `gamma` from `0.1` to `10`. A small `gamma` means a large radius for the Gaussian kernel, which means that many points are considered close by. This is reflected in very smooth decision boundaries on the left, and boundaries that focus more on single points further to the right. A low value of `gamma` means that the decision boundary will vary slowly, which yields a model of low complexity, while a high value of `gamma` yields a more complex model.

Going from top to bottom, we increase the `C` parameter from `0.1` to `1000`. As with the linear models, a small `C` means a very restricted model, where each data point can only have very limited influence. You can see that at the top left the decision boundary looks nearly linear, with the misclassified points barely having any influence on the line. Increasing `C`, as shown on the bottom right, allows these points to have a stronger influence on the model and makes the decision boundary bend to correctly classify them.

Let's apply the RBF kernel SVM to the Breast Cancer dataset. By default, $C=1$ and $\gamma=1/n_features$:

In[83]:

```
X_train, X_test, y_train, y_test = train_test_split(  
    cancer.data, cancer.target, random_state=0)  
  
svc = SVC()  
svc.fit(X_train, y_train)  
  
print("Accuracy on training set: {:.2f}".format(svc.score(X_train, y_train)))  
print("Accuracy on test set: {:.2f}".format(svc.score(X_test, y_test)))
```

Out[83]:

```
Accuracy on training set: 1.00  
Accuracy on test set: 0.63
```

The model overfits quite substantially, with a perfect score on the training set and only 63% accuracy on the test set. While SVMs often perform quite well, they are very sensitive to the settings of the parameters and to the scaling of the data. In particular, they require all the features to vary on a similar scale. Let's look at the minimum and maximum values for each feature, plotted in log-space (Figure 2-43):

In[84]:

```
plt.plot(X_train.min(axis=0), 'o', label="min")  
plt.plot(X_train.max(axis=0), '^', label="max")  
plt.legend(loc=4)  
plt.xlabel("Feature index")  
plt.ylabel("Feature magnitude")  
plt.yscale("log")
```

From this plot we can determine that features in the Breast Cancer dataset are of completely different orders of magnitude. This can be somewhat of a problem for other models (like linear models), but it has devastating effects for the kernel SVM. Let's examine some ways to deal with this issue.

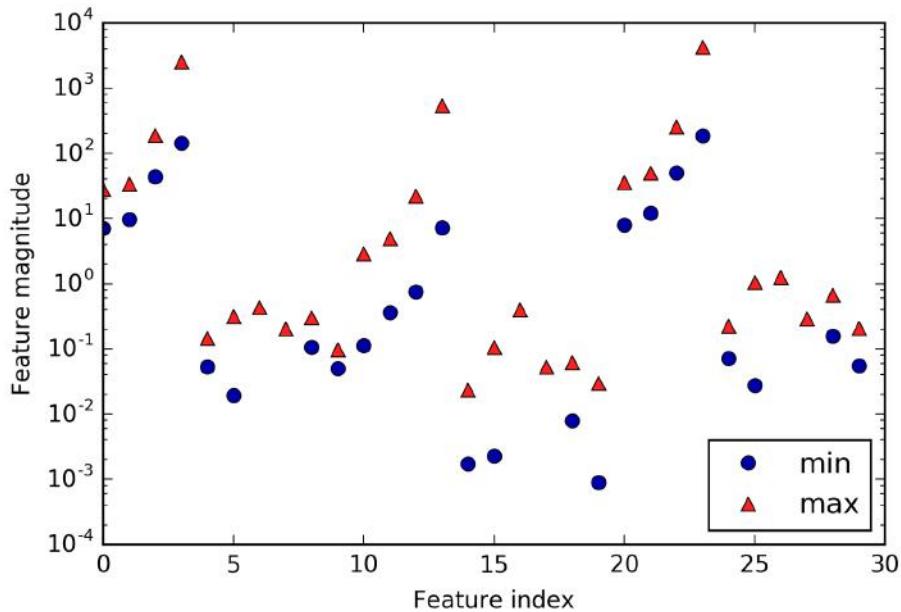


Figure 2-43. Feature ranges for the Breast Cancer dataset (note that the y axis has a logarithmic scale)

Preprocessing data for SVMs

One way to resolve this problem is by rescaling each feature so that they are all approximately on the same scale. A common rescaling method for kernel SVMs is to scale the data such that all features are between 0 and 1. We will see how to do this using the `MinMaxScaler` preprocessing method in [Chapter 3](#), where we'll give more details. For now, let's do this “by hand”:

In[85]:

```
# compute the minimum value per feature on the training set
min_on_training = X_train.min(axis=0)
# compute the range of each feature (max - min) on the training set
range_on_training = (X_train - min_on_training).max(axis=0)

# subtract the min, and divide by range
# afterward, min=0 and max=1 for each feature
X_train_scaled = (X_train - min_on_training) / range_on_training
print("Minimum for each feature\n{}".format(X_train_scaled.min(axis=0)))
print("Maximum for each feature\n {}".format(X_train_scaled.max(axis=0)))
```

Out[85]:

```
Minimum for each feature  
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  
 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]  
Maximum for each feature  
[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  
 1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

In[86]:

```
# use THE SAME transformation on the test set,  
# using min and range of the training set (see Chapter 3 for details)  
X_test_scaled = (X_test - min_on_training) / range_on_training
```

In[87]:

```
svc = SVC()  
svc.fit(X_train_scaled, y_train)  
  
print("Accuracy on training set: {:.3f}".format(  
    svc.score(X_train_scaled, y_train)))  
print("Accuracy on test set: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```

Out[87]:

```
Accuracy on training set: 0.948  
Accuracy on test set: 0.951
```

Scaling the data made a huge difference! Now we are actually in an underfitting regime, where training and test set performance are quite similar but less close to 100% accuracy. From here, we can try increasing either C or gamma to fit a more complex model. For example:

In[88]:

```
svc = SVC(C=1000)  
svc.fit(X_train_scaled, y_train)  
  
print("Accuracy on training set: {:.3f}".format(  
    svc.score(X_train_scaled, y_train)))  
print("Accuracy on test set: {:.3f}".format(svc.score(X_test_scaled, y_test)))
```

Out[88]:

```
Accuracy on training set: 0.988  
Accuracy on test set: 0.972
```

Here, increasing C allows us to improve the model significantly, resulting in 97.2% accuracy.

Strengths, weaknesses, and parameters

Kernelized support vector machines are powerful models and perform well on a variety of datasets. SVMs allow for complex decision boundaries, even if the data has only a few features. They work well on low-dimensional and high-dimensional data (i.e., few and many features), but don't scale very well with the number of samples. Running an SVM on data with up to 10,000 samples might work well, but working with datasets of size 100,000 or more can become challenging in terms of runtime and memory usage.

Another downside of SVMs is that they require careful preprocessing of the data and tuning of the parameters. This is why, these days, most people instead use tree-based models such as random forests or gradient boosting (which require little or no preprocessing) in many applications. Furthermore, SVM models are hard to inspect; it can be difficult to understand why a particular prediction was made, and it might be tricky to explain the model to a nonexpert.

Still, it might be worth trying SVMs, particularly if all of your features represent measurements in similar units (e.g., all are pixel intensities) and they are on similar scales.

The important parameters in kernel SVMs are the regularization parameter `C`, the choice of the kernel, and the kernel-specific parameters. Although we primarily focused on the RBF kernel, other choices are available in `scikit-learn`. The RBF kernel has only one parameter, `gamma`, which is the inverse of the width of the Gaussian kernel. `gamma` and `C` both control the complexity of the model, with large values in either resulting in a more complex model. Therefore, good settings for the two parameters are usually strongly correlated, and `C` and `gamma` should be adjusted together.

Neural Networks (Deep Learning)

A family of algorithms known as neural networks has recently seen a revival under the name “deep learning.” While deep learning shows great promise in many machine learning applications, deep learning algorithms are often tailored very carefully to a specific use case. Here, we will only discuss some relatively simple methods, namely *multilayer perceptrons* for classification and regression, that can serve as a starting point for more involved deep learning methods. Multilayer perceptrons (MLPs) are also known as (vanilla) feed-forward neural networks, or sometimes just neural networks.

The neural network model

MLPs can be viewed as generalizations of linear models that perform multiple stages of processing to come to a decision.

Remember that the prediction by a linear regressor is given as:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \dots + w[p] * x[p] + b$$

In plain English, \hat{y} is a weighted sum of the input features $x[0]$ to $x[p]$, weighted by the learned coefficients $w[0]$ to $w[p]$. We could visualize this graphically as shown in Figure 2-44:

In[89]:

```
display(mglearn.plots.plot_logistic_regression_graph())
```

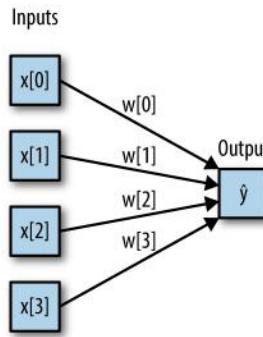


Figure 2-44. Visualization of logistic regression, where input features and predictions are shown as nodes, and the coefficients are connections between the nodes

Here, each node on the left represents an input feature, the connecting lines represent the learned coefficients, and the node on the right represents the output, which is a weighted sum of the inputs.

In an MLP this process of computing weighted sums is repeated multiple times, first computing *hidden units* that represent an intermediate processing step, which are again combined using weighted sums to yield the final result (Figure 2-45):

In[90]:

```
display(mglearn.plots.plot_single_hidden_layer_graph())
```

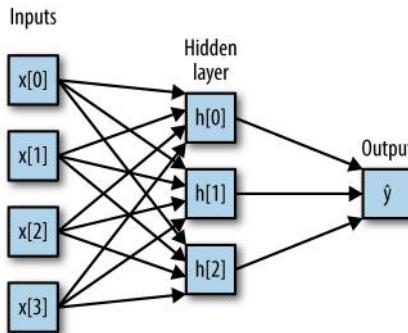


Figure 2-45. Illustration of a multilayer perceptron with a single hidden layer

This model has a lot more coefficients (also called weights) to learn: there is one between every input and every hidden unit (which make up the *hidden layer*), and one between every unit in the hidden layer and the output.

Computing a series of weighted sums is mathematically the same as computing just one weighted sum, so to make this model truly more powerful than a linear model, we need one extra trick. After computing a weighted sum for each hidden unit, a nonlinear function is applied to the result—usually the *rectifying nonlinearity* (also known as rectified linear unit or *relu*) or the *tangens hyperbolicus* (*tanh*). The result of this function is then used in the weighted sum that computes the output, \hat{y} . The two functions are visualized in Figure 2-46. The *relu* cuts off values below zero, while *tanh* saturates to -1 for low input values and $+1$ for high input values. Either nonlinear function allows the neural network to learn much more complicated functions than a linear model could:

In[91]:

```
line = np.linspace(-3, 3, 100)
plt.plot(line, np.tanh(line), label="tanh")
plt.plot(line, np.maximum(line, 0), label="relu")
plt.legend(loc="best")
plt.xlabel("x")
plt.ylabel("relu(x), tanh(x)")
```

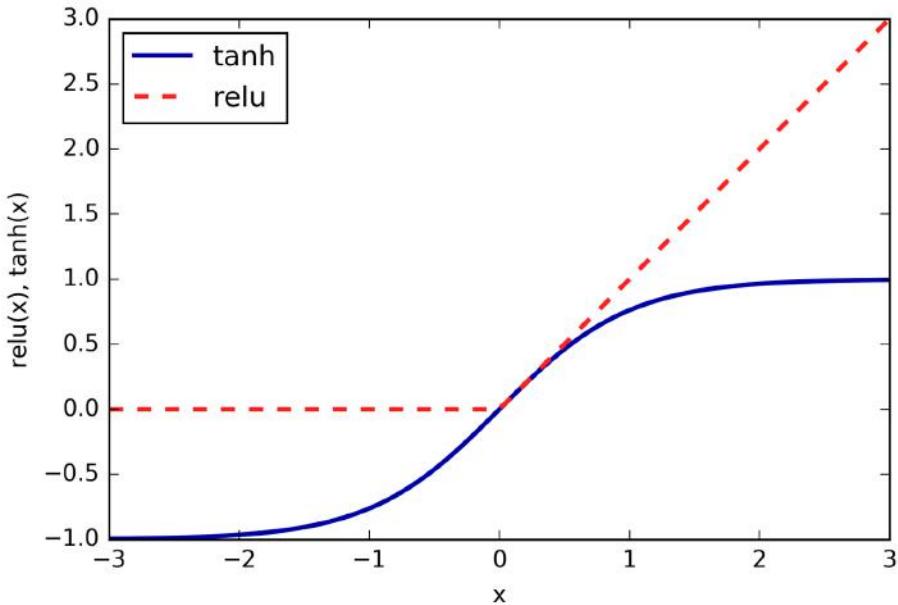


Figure 2-46. The hyperbolic tangent activation function and the rectified linear activation function

For the small neural network pictured in [Figure 2-45](#), the full formula for computing \hat{y} in the case of regression would be (when using a tanh nonlinearity):

$$\begin{aligned} h[0] &= \tanh(w[0, 0] * x[0] + w[1, 0] * x[1] + w[2, 0] * x[2] + w[3, 0] * x[3]) \\ h[1] &= \tanh(w[0, 0] * x[0] + w[1, 0] * x[1] + w[2, 0] * x[2] + w[3, 0] * x[3]) \\ h[2] &= \tanh(w[0, 0] * x[0] + w[1, 0] * x[1] + w[2, 0] * x[2] + w[3, 0] * x[3]) \\ \hat{y} &= v[0] * h[0] + v[1] * h[1] + v[2] * h[2] \end{aligned}$$

Here, w are the weights between the input x and the hidden layer h , and v are the weights between the hidden layer h and the output \hat{y} . The weights v and w are learned from data, x are the input features, \hat{y} is the computed output, and h are intermediate computations. An important parameter that needs to be set by the user is the number of nodes in the hidden layer. This can be as small as 10 for very small or simple data-sets and as big as 10,000 for very complex data. It is also possible to add additional hidden layers, as shown in [Figure 2-47](#):

In[92]:

```
mglearn.plots.plot_two_hidden_layer_graph()
```

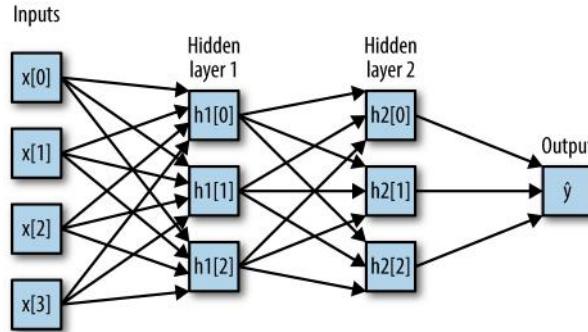


Figure 2-47. A multilayer perceptron with two hidden layers

Having large neural networks made up of many of these layers of computation is what inspired the term “deep learning.”

Tuning neural networks

Let’s look into the workings of the MLP by applying the `MLPClassifier` to the `two_moons` dataset we used earlier in this chapter. The results are shown in Figure 2-48:

In[93]:

```
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=100, noise=0.25, random_state=3)

X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    random_state=42)

mlp = MLPClassifier(algorithm='l-bfgs', random_state=0).fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

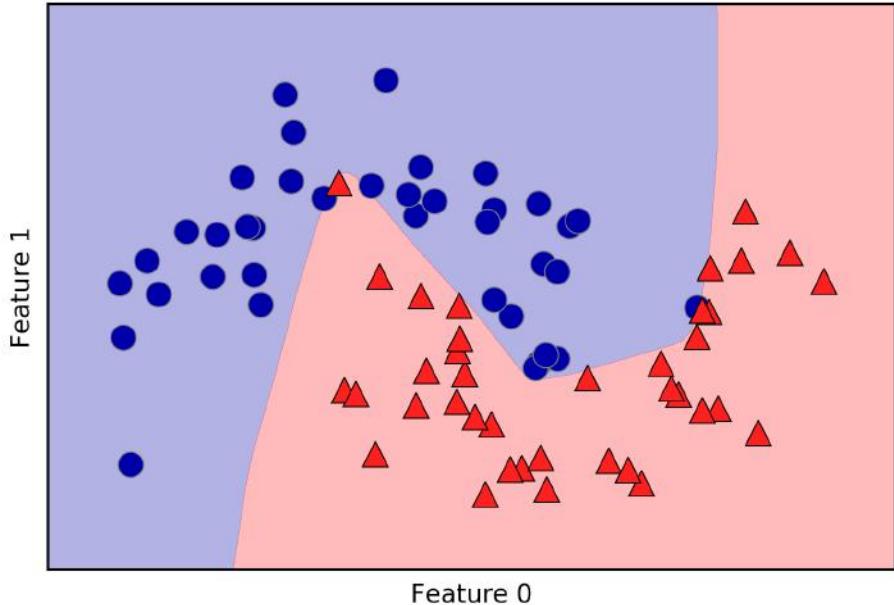


Figure 2-48. Decision boundary learned by a neural network with 100 hidden units on the two_moons dataset

As you can see, the neural network learned a very nonlinear but relatively smooth decision boundary. We used `algorithm='l-bfgs'`, which we will discuss later.

By default, the MLP uses 100 hidden nodes, which is quite a lot for this small dataset. We can reduce the number (which reduces the complexity of the model) and still get a good result ([Figure 2-49](#)):

In[94]:

```
mlp = MLPClassifier(algorithm='l-bfgs', random_state=0, hidden_layer_sizes=[10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

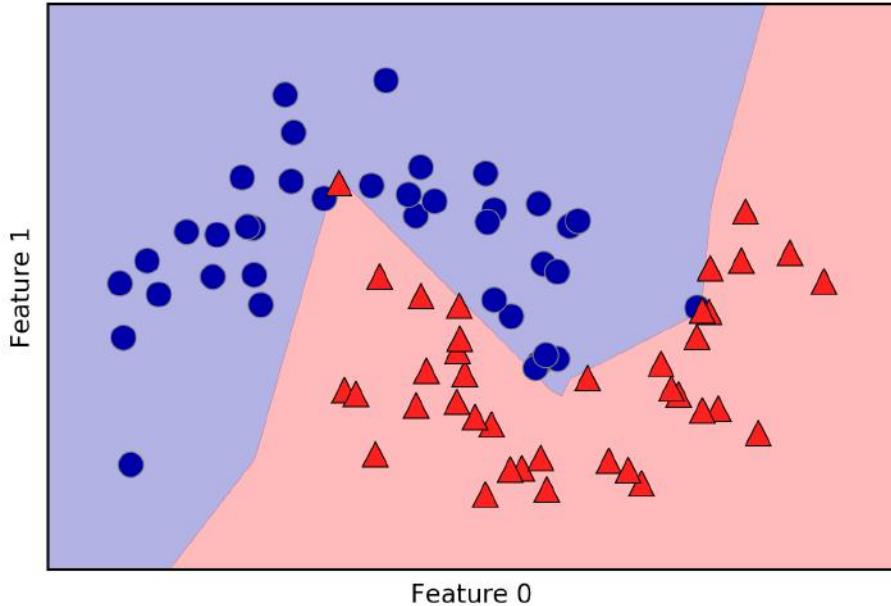


Figure 2-49. Decision boundary learned by a neural network with 10 hidden units on the two_moons dataset

With only 10 hidden units, the decision boundary looks somewhat more ragged. The default nonlinearity is `relu`, shown in [Figure 2-46](#). With a single hidden layer, this means the decision function will be made up of 10 straight line segments. If we want a smoother decision boundary, we could add more hidden units (as in [Figure 2-49](#)), add a second hidden layer ([Figure 2-50](#)), or use the `tanh` nonlinearity ([Figure 2-51](#)):

In[95]:

```
# using two hidden layers, with 10 units each
mlp = MLPClassifier(algorithm='l-bfgs', random_state=0,
                     hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

In[96]:

```
# using two hidden layers, with 10 units each, now with tanh nonlinearity
mlp = MLPClassifier(algorithm='l-bfgs', activation='tanh',
                     random_state=0, hidden_layer_sizes=[10, 10])
mlp.fit(X_train, y_train)
mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3)
mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

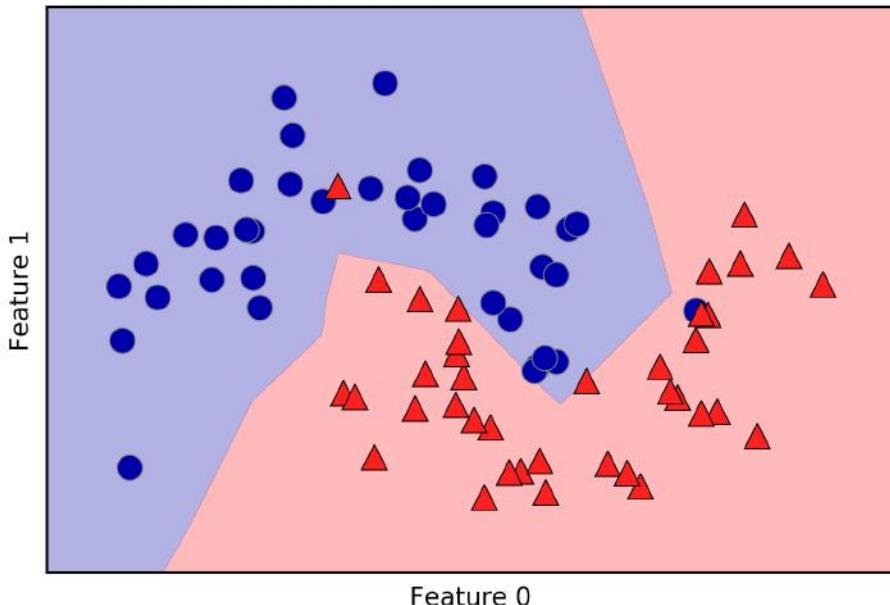


Figure 2-50. Decision boundary learned using 2 hidden layers with 10 hidden units each, with rect activation function

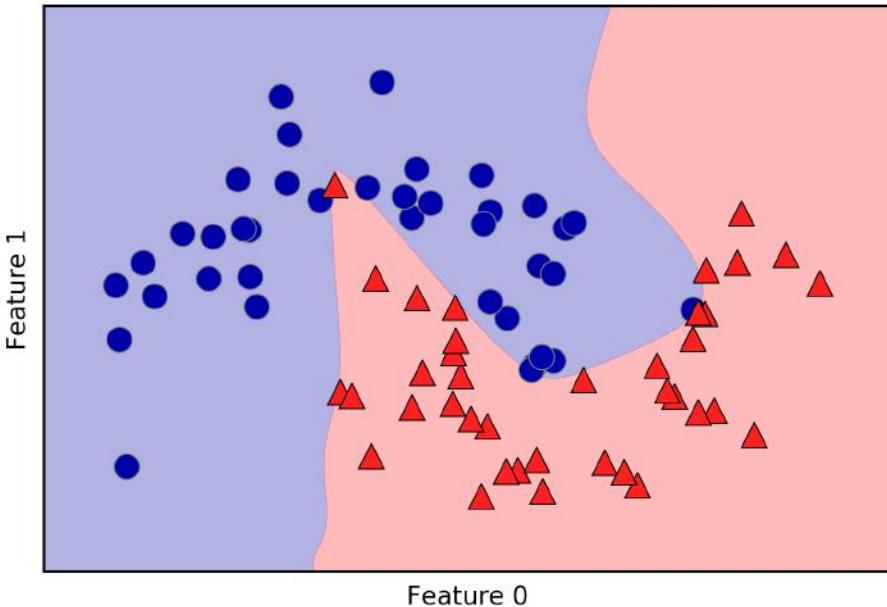


Figure 2-51. Decision boundary learned using 2 hidden layers with 10 hidden units each, with tanh activation function

Finally, we can also control the complexity of a neural network by using an ℓ_2 penalty to shrink the weights toward zero, as we did in ridge regression and the linear classifiers. The parameter for this in the `MLPClassifier` is `alpha` (as in the linear regression models), and it's set to a very low value (little regularization) by default. [Figure 2-52](#) shows the effect of different values of `alpha` on the `two_moons` dataset, using two hidden layers of 10 or 100 units each:

In[97]:

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for axx, n_hidden_nodes in zip(axes, [10, 100]):
    for ax, alpha in zip(axx, [0.0001, 0.01, 0.1, 1]):
        mlp = MLPClassifier(algorithm='l-bfgs', random_state=0,
                            hidden_layer_sizes=[n_hidden_nodes, n_hidden_nodes],
                            alpha=alpha)
        mlp.fit(X_train, y_train)
        mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
        mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, ax=ax)
        ax.set_title("n_hidden=[{}, {}]\nalpha={:.4f}".format(
            n_hidden_nodes, n_hidden_nodes, alpha))
```

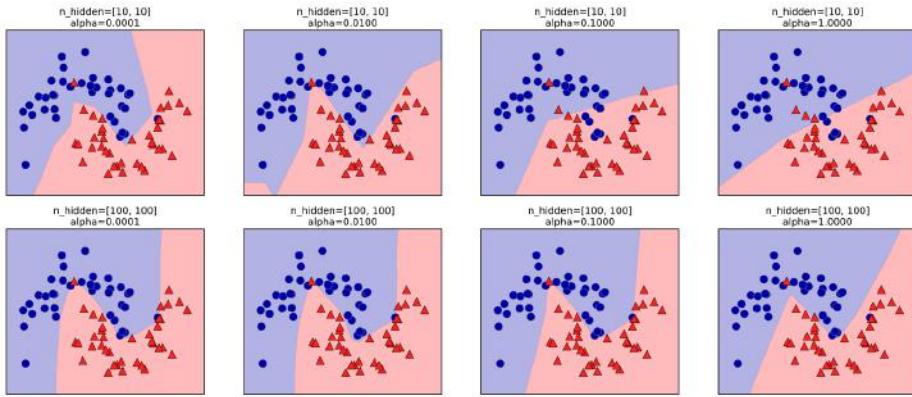


Figure 2-52. Decision functions for different numbers of hidden units and different settings of the alpha parameter

As you probably have realized by now, there are many ways to control the complexity of a neural network: the number of hidden layers, the number of units in each hidden layer, and the regularization (`alpha`). There are actually even more, which we won't go into here.

An important property of neural networks is that their weights are set randomly before learning is started, and this random initialization affects the model that is learned. That means that even when using exactly the same parameters, we can obtain very different models when using different random seeds. If the networks are large, and their complexity is chosen properly, this should not affect accuracy too much, but it is worth keeping in mind (particularly for smaller networks). [Figure 2-53](#) shows plots of several models, all learned with the same settings of the parameters:

In[98]:

```
fig, axes = plt.subplots(2, 4, figsize=(20, 8))
for i, ax in enumerate(axes.ravel()):
    mlp = MLPClassifier(algorithm='l-bfgs', random_state=i,
                         hidden_layer_sizes=[100, 100])
    mlp.fit(X_train, y_train)
    mglearn.plots.plot_2d_separator(mlp, X_train, fill=True, alpha=.3, ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train, ax=ax)
```

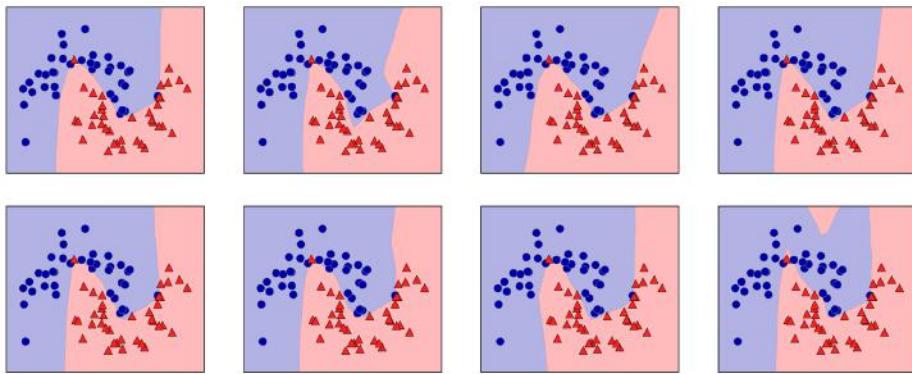


Figure 2-53. Decision functions learned with the same parameters but different random initializations

To get a better understanding of neural networks on real-world data, let's apply the `MLPClassifier` to the Breast Cancer dataset. We start with the default parameters:

In[99]:

```
print("Cancer data per-feature maxima:\n{}".format(cancer.data.max(axis=0)))
```

Out[99]:

```
Cancer data per-feature maxima:
 [ 28.110    39.280   188.500  2501.000     0.163     0.345     0.427
   0.201     0.304     0.097     2.873     4.885    21.980   542.200
   0.031     0.135     0.396     0.053     0.079     0.030   36.040
  49.540   251.200  4254.000     0.223     1.058     1.252     0.291
   0.664     0.207]
```

In[100]:

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

mlp = MLPClassifier(random_state=42)
mlp.fit(X_train, y_train)

print("Accuracy on training set: {:.2f}".format(mlp.score(X_train, y_train)))
print("Accuracy on test set: {:.2f}".format(mlp.score(X_test, y_test)))
```

Out[100]:

```
Accuracy on training set: 0.92
Accuracy on test set: 0.90
```

The accuracy of the MLP is quite good, but not as good as the other models. As in the earlier SVC example, this is likely due to scaling of the data. Neural networks also expect all input features to vary in a similar way, and ideally to have a mean of 0, and

a variance of 1. We must rescale our data so that it fulfills these requirements. Again, we will do this by hand here, but we'll introduce the `StandardScaler` to do this automatically in [Chapter 3](#):

In[101]:

```
# compute the mean value per feature on the training set
mean_on_train = X_train.mean(axis=0)
# compute the standard deviation of each feature on the training set
std_on_train = X_train.std(axis=0)

# subtract the mean, and scale by inverse standard deviation
# afterward, mean=0 and std=1
X_train_scaled = (X_train - mean_on_train) / std_on_train
# use THE SAME transformation (using training mean and std) on the test set
X_test_scaled = (X_test - mean_on_train) / std_on_train

mlp = MLPClassifier(random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

Out[101]:

```
Accuracy on training set: 0.991
Accuracy on test set: 0.965

ConvergenceWarning:
  Stochastic Optimizer: Maximum iterations reached and the optimization
  hasn't converged yet.
```

The results are much better after scaling, and already quite competitive. We got a warning from the model, though, that tells us that the maximum number of iterations has been reached. This is part of the `adam` algorithm for learning the model, and tells us that we should increase the number of iterations:

In[102]:

```
mlp = MLPClassifier(max_iter=1000, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

Out[102]:

```
Accuracy on training set: 0.995
Accuracy on test set: 0.965
```

Increasing the number of iterations only increased the training set performance, not the generalization performance. Still, the model is performing quite well. As there is some gap between the training and the test performance, we might try to decrease the model's complexity to get better generalization performance. Here, we choose to increase the alpha parameter (quite aggressively, from 0.0001 to 1) to add stronger regularization of the weights:

In[103]:

```
mlp = MLPClassifier(max_iter=1000, alpha=1, random_state=0)
mlp.fit(X_train_scaled, y_train)

print("Accuracy on training set: {:.3f}".format(
    mlp.score(X_train_scaled, y_train)))
print("Accuracy on test set: {:.3f}".format(mlp.score(X_test_scaled, y_test)))
```

Out[103]:

```
Accuracy on training set: 0.988
Accuracy on test set: 0.972
```

This leads to a performance on par with the best models so far.¹²

While it is possible to analyze what a neural network has learned, this is usually much trickier than analyzing a linear model or a tree-based model. One way to introspect what was learned is to look at the weights in the model. You can see an example of this in the [scikit-learn example gallery](#). For the Breast Cancer dataset, this might be a bit hard to understand. The following plot (Figure 2-54) shows the weights that were learned connecting the input to the first hidden layer. The rows in this plot correspond to the 30 input features, while the columns correspond to the 100 hidden units. Light colors represent large positive values, while dark colors represent negative values:

In[104]:

```
plt.figure(figsize=(20, 5))
plt.imshow(mlp.coefs_[0], interpolation='none', cmap='viridis')
plt.yticks(range(30), cancer.feature_names)
plt.xlabel("Columns in weight matrix")
plt.ylabel("Input feature")
plt.colorbar()
```

¹² You might have noticed at this point that many of the well-performing models achieved exactly the same accuracy of 0.972. This means that all of the models make exactly the same number of mistakes, which is four. If you compare the actual predictions, you can even see that they make exactly the same mistakes! This might be a consequence of the dataset being very small, or it may be because these points are really different from the rest.

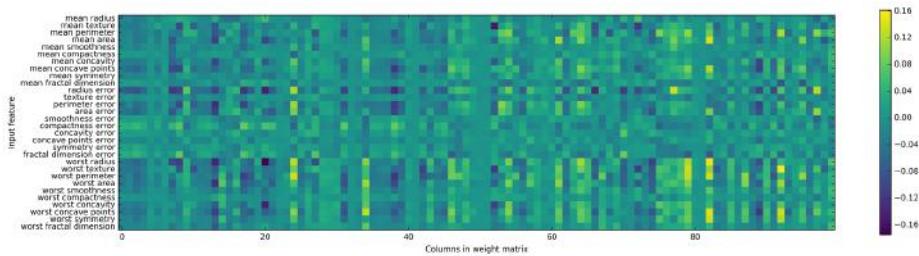


Figure 2-54. Heat map of the first layer weights in a neural network learned on the Breast Cancer dataset

One possible inference we can make is that features that have very small weights for all of the hidden units are “less important” to the model. We can see that “mean smoothness” and “mean compactness,” in addition to the features found between “smoothness error” and “fractal dimension error,” have relatively low weights compared to other features. This could mean that these are less important features or possibly that we didn’t represent them in a way that the neural network could use.

We could also visualize the weights connecting the hidden layer to the output layer, but those are even harder to interpret.

While the `MLPClassifier` and `MLPRegressor` provide easy-to-use interfaces for the most common neural network architectures, they only capture a small subset of what is possible with neural networks. If you are interested in working with more flexible or larger models, we encourage you to look beyond `scikit-learn` into the fantastic deep learning libraries that are out there. For Python users, the most well-established are `keras`, `lasagna`, and `tensor-flow`. `lasagna` builds on the `theano` library, while `keras` can use either `tensor-flow` or `theano`. These libraries provide a much more flexible interface to build neural networks and track the rapid progress in deep learning research. All of the popular deep learning libraries also allow the use of high-performance graphics processing units (GPUs), which `scikit-learn` does not support. Using GPUs allows us to accelerate computations by factors of 10x to 100x, and they are essential for applying deep learning methods to large-scale datasets.

Strengths, weaknesses, and parameters

Neural networks have reemerged as state-of-the-art models in many applications of machine learning. One of their main advantages is that they are able to capture information contained in large amounts of data and build incredibly complex models. Given enough computation time, data, and careful tuning of the parameters, neural networks often beat other machine learning algorithms (for classification and regression tasks).

This brings us to the downsides. Neural networks—particularly the large and powerful ones—often take a long time to train. They also require careful preprocessing of the data, as we saw here. Similarly to SVMs, they work best with “homogeneous” data, where all the features have similar meanings. For data that has very different kinds of features, tree-based models might work better. Tuning neural network parameters is also an art unto itself. In our experiments, we barely scratched the surface of possible ways to adjust neural network models and how to train them.

Estimating complexity in neural networks. The most important parameters are the number of layers and the number of hidden units per layer. You should start with one or two hidden layers, and possibly expand from there. The number of nodes per hidden layer is often similar to the number of input features, but rarely higher than in the low to mid-thousands.

A helpful measure when thinking about the model complexity of a neural network is the number of weights or coefficients that are learned. If you have a binary classification dataset with 100 features, and you have 100 hidden units, then there are $100 * 100 = 10,000$ weights between the input and the first hidden layer. There are also $100 * 1 = 100$ weights between the hidden layer and the output layer, for a total of around 10,100 weights. If you add a second hidden layer with 100 hidden units, there will be another $100 * 100 = 10,000$ weights from the first hidden layer to the second hidden layer, resulting in a total of 20,100 weights. If instead you use one layer with 1,000 hidden units, you are learning $100 * 1,000 = 100,000$ weights from the input to the hidden layer and $1,000 * 1$ weights from the hidden layer to the output layer, for a total of 101,000. If you add a second hidden layer you add $1,000 * 1,000 = 1,000,000$ weights, for a whopping total of 1,101,000—50 times larger than the model with two hidden layers of size 100.

A common way to adjust parameters in a neural network is to first create a network that is large enough to overfit, making sure that the task can actually be learned by the network. Then, once you know the training data can be learned, either shrink the network or increase `alpha` to add regularization, which will improve generalization performance.

In our experiments, we focused mostly on the definition of the model: the number of layers and nodes per layer, the regularization, and the nonlinearity. These define the model we want to learn. There is also the question of *how* to learn the model, or the algorithm that is used for learning the parameters, which is set using the `algorithm` parameter. There are two easy-to-use choices for `algorithm`. The default is '`adam`', which works well in most situations but is quite sensitive to the scaling of the data (so it is important to always scale your data to 0 mean and unit variance). The other one is '`l-bfgs`', which is quite robust but might take a long time on larger models or larger datasets. There is also the more advanced '`sgd`' option, which is what many deep learning researchers use. The '`sgd`' option comes with many additional param-

eters that need to be tuned for best results. You can find all of these parameters and their definitions in the user guide. When starting to work with MLPs, we recommend sticking to 'adam' and 'l-bfgs'.



fit Resets a Model

An important property of scikit-learn models is that calling `fit` will always reset everything a model previously learned. So if you build a model on one dataset, and then call `fit` again on a different dataset, the model will “forget” everything it learned from the first dataset. You can call `fit` as often as you like on a model, and the outcome will be the same as calling `fit` on a “new” model.

Uncertainty Estimates from Classifiers

Another useful part of the scikit-learn interface that we haven’t talked about yet is the ability of classifiers to provide uncertainty estimates of predictions. Often, you are not only interested in which class a classifier predicts for a certain test point, but also how certain it is that this is the right class. In practice, different kinds of mistakes lead to very different outcomes in real-world applications. Imagine a medical application testing for cancer. Making a false positive prediction might lead to a patient undergoing additional tests, while a false negative prediction might lead to a serious disease not being treated. We will go into this topic in more detail in [Chapter 6](#).

There are two different functions in scikit-learn that can be used to obtain uncertainty estimates from classifiers: `decision_function` and `predict_proba`. Most (but not all) classifiers have at least one of them, and many classifiers have both. Let’s look at what these two functions do on a synthetic two-dimensional dataset, when building a `GradientBoostingClassifier` classifier, which has both a `decision_function` and a `predict_proba` method:

In[105]:

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import make_blobs, make_circles
X, y = make_circles(noise=0.25, factor=0.5, random_state=1)

# we rename the classes "blue" and "red" for illustration purposes
y_named = np.array(["blue", "red"])[y]

# we can call train_test_split with arbitrarily many arrays;
# all will be split in a consistent manner
X_train, X_test, y_train_named, y_test_named, y_train, y_test = \
    train_test_split(X, y_named, y, random_state=0)

# build the gradient boosting model
gbdt = GradientBoostingClassifier(random_state=0)
gbdt.fit(X_train, y_train_named)
```

The Decision Function

In the binary classification case, the return value of `decision_function` is of shape `(n_samples,)`, and it returns one floating-point number for each sample:

In[106]:

```
print("X_test.shape: {}".format(X_test.shape))
print("Decision function shape: {}".format(
    gbrt.decision_function(X_test).shape))
```

Out[106]:

```
X_test.shape: (25, 2)
Decision function shape: (25,)
```

This value encodes how strongly the model believes a data point to belong to the “positive” class, in this case class 1. Positive values indicate a preference for the positive class, and negative values indicate a preference for the “negative” (other) class:

In[107]:

```
# show the first few entries of decision_function
print("Decision function:\n{}".format(gbrt.decision_function(X_test)[:6]))
```

Out[107]:

```
Decision function:
[ 4.136 -1.683 -3.951 -3.626  4.29   3.662]
```

We can recover the prediction by looking only at the sign of the decision function:

In[108]:

```
print("Thresholded decision function:\n{}".format(
    gbrt.decision_function(X_test) > 0))
print("Predictions:\n{}".format(gbrt.predict(X_test)))
```

Out[108]:

```
Thresholded decision function:
[ True False False False  True  True False  True  True  True False  True
  True False  True False False False  True  True  True  True  True False
  False]
Predictions:
['red' 'blue' 'blue' 'blue' 'red' 'red' 'blue' 'red' 'red' 'red' 'blue'
 'red' 'red' 'blue' 'red' 'blue' 'blue' 'blue' 'red' 'red' 'red' 'red'
 'red' 'blue' 'blue']
```

For binary classification, the “negative” class is always the first entry of the `classes_` attribute, and the “positive” class is the second entry of `classes_`. So if you want to fully recover the output of `predict`, you need to make use of the `classes_` attribute:

In[109]:

```
# make the boolean True/False into 0 and 1
greater_zero = (gbrt.decision_function(X_test) > 0).astype(int)
# use 0 and 1 as indices into classes_
pred = gbrt.classes_[greater_zero]
# pred is the same as the output of gbrt.predict
print("pred is equal to predictions: {}".format(
    np.all(pred == gbrt.predict(X_test))))
```

Out[109]:

```
pred is equal to predictions: True
```

The range of `decision_function` can be arbitrary, and depends on the data and the model parameters:

In[110]:

```
decision_function = gbrt.decision_function(X_test)
print("Decision function minimum: {:.2f} maximum: {:.2f}".format(
    np.min(decision_function), np.max(decision_function)))
```

Out[110]:

```
Decision function minimum: -7.69 maximum: 4.29
```

This arbitrary scaling makes the output of `decision_function` often hard to interpret.

In the following example we plot the `decision_function` for all points in the 2D plane using a color coding, next to a visualization of the decision boundary, as we saw earlier. We show training points as circles and test data as triangles (Figure 2-55):

In[111]:

```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))
mglearn.tools.plot_2d_separator(gbrt, X, ax=axes[0], alpha=.4,
                                fill=True, cm=mglearn.cm2)
scores_image = mglearn.tools.plot_2d_scores(gbrt, X, ax=axes[1],
                                            alpha=.4, cm=mglearn.ReBl)

for ax in axes:
    # plot training and test points
    mglearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test,
                             markers='^', ax=ax)
    mglearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train,
                             markers='o', ax=ax)
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")
cbar = plt.colorbar(scores_image, ax=axes.tolist())
axes[0].legend(["Test class 0", "Test class 1", "Train class 0",
                "Train class 1"], ncol=4, loc=(.1, 1.1))
```

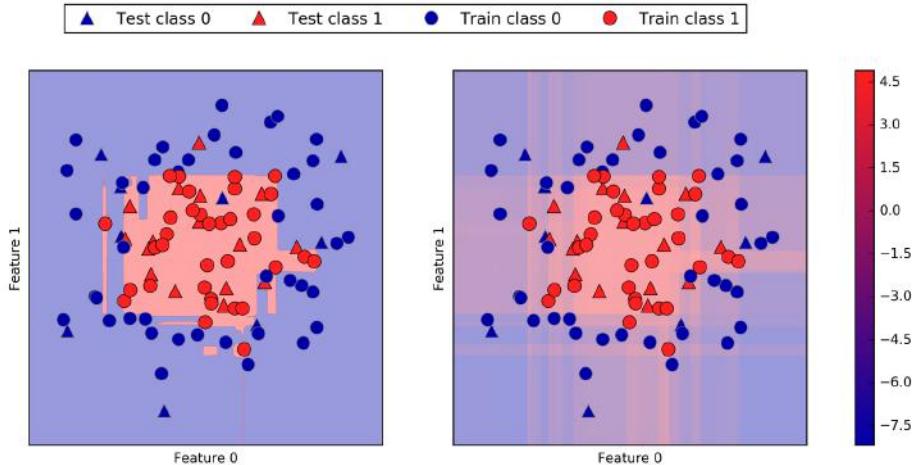


Figure 2-55. Decision boundary (left) and decision function (right) for a gradient boosting model on a two-dimensional toy dataset

Encoding not only the predicted outcome but also how certain the classifier is provides additional information. However, in this visualization, it is hard to make out the boundary between the two classes.

Predicting Probabilities

The output of `predict_proba` is a probability for each class, and is often more easily understood than the output of `decision_function`. It is always of shape `(n_samples, 2)` for binary classification:

In[112]:

```
print("Shape of probabilities: {}".format(gbrt.predict_proba(X_test).shape))
```

Out[112]:

```
Shape of probabilities: (25, 2)
```

The first entry in each row is the estimated probability of the first class, and the second entry is the estimated probability of the second class. Because it is a probability, the output of `predict_proba` is always between 0 and 1, and the sum of the entries for both classes is always 1:

In[113]:

```
# show the first few entries of predict_proba
print("Predicted probabilities:\n{}".format(
    gbrt.predict_proba(X_test[:6])))
```

Out[113]:

```
Predicted probabilities:  
[[ 0.016 0.984]  
 [ 0.843 0.157]  
 [ 0.981 0.019]  
 [ 0.974 0.026]  
 [ 0.014 0.986]  
 [ 0.025 0.975]]
```

Because the probabilities for the two classes sum to 1, exactly one of the classes will be above 50% certainty. That class is the one that is predicted.¹³

You can see in the previous output that the classifier is relatively certain for most points. How well the uncertainty actually reflects uncertainty in the data depends on the model and the parameters. A model that is more overfitted tends to make more certain predictions, even if they might be wrong. A model with less complexity usually has more uncertainty in its predictions. A model is called *calibrated* if the reported uncertainty actually matches how correct it is—in a calibrated model, a prediction made with 70% certainty would be correct 70% of the time.

In the following example (Figure 2-56) we again show the decision boundary on the dataset, next to the class probabilities for the class 1:

In[114]:

```
fig, axes = plt.subplots(1, 2, figsize=(13, 5))  
  
mlearn.tools.plot_2d_separator(  
    gbrt, X, ax=axes[0], alpha=.4, fill=True, cm=mlearn.cm2)  
scores_image = mlearn.tools.plot_2d_scores(  
    gbrt, X, ax=axes[1], alpha=.5, cm=mlearn.ReBl, function='predict_proba')  
  
for ax in axes:  
    # plot training and test points  
    mlearn.discrete_scatter(X_test[:, 0], X_test[:, 1], y_test,  
                            markers='^', ax=ax)  
    mlearn.discrete_scatter(X_train[:, 0], X_train[:, 1], y_train,  
                            markers='o', ax=ax)  
    ax.set_xlabel("Feature 0")  
    ax.set_ylabel("Feature 1")  
cbar = plt.colorbar(scores_image, ax=axes.tolist())  
axes[0].legend(["Test class 0", "Test class 1", "Train class 0",  
                "Train class 1"], ncol=4, loc=(.1, 1.1))
```

¹³ Because the probabilities are floating-point numbers, it is unlikely that they will both be exactly 0.500. However, if that happens, the prediction is made at random.

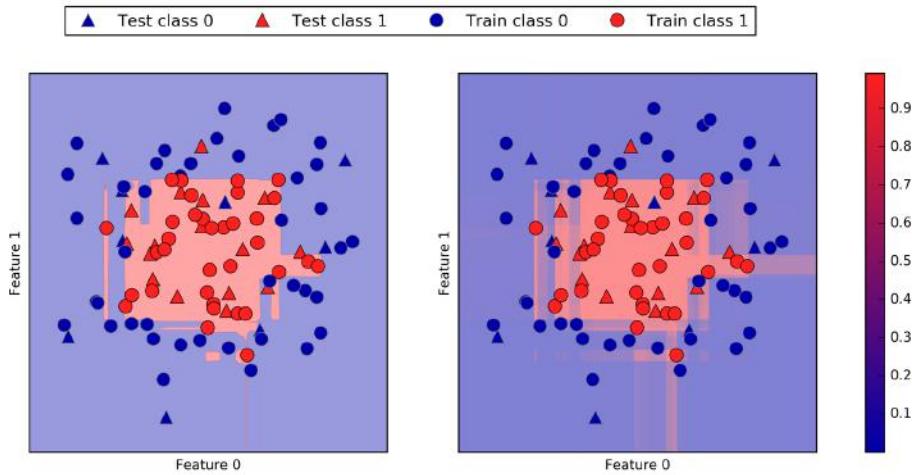


Figure 2-56. Decision boundary (left) and predicted probabilities for the gradient boosting model shown in Figure 2-55

The boundaries in this plot are much more well-defined, and the small areas of uncertainty are clearly visible.

The [scikit-learn website](#) has a great comparison of many models and what their uncertainty estimates look like. We've reproduced this in [Figure 2-57](#), and we encourage you to go though the example there.

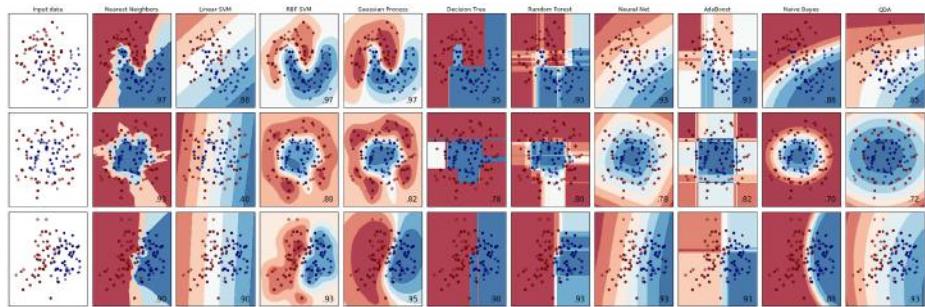


Figure 2-57. Comparison of several classifiers in scikit-learn on synthetic datasets (image courtesy <http://scikit-learn.org>)

Uncertainty in Multiclass Classification

So far, we've only talked about uncertainty estimates in binary classification. But the `decision_function` and `predict_proba` methods also work in the multiclass setting. Let's apply them on the Iris dataset, which is a three-class classification dataset:

In[115]:

```
from sklearn.datasets import load_iris

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=42)

gbrt = GradientBoostingClassifier(learning_rate=0.01, random_state=0)
gbrt.fit(X_train, y_train)
```

In[116]:

```
print("Decision function shape: {}".format(gbdt.decision_function(X_test).shape))
# plot the first few entries of the decision function
print("Decision function:\n{}".format(gbdt.decision_function(X_test)[:6, :]))
```

Out[116]:

```
Decision function shape: (38, 3)
Decision function:
[[-0.529  1.466 -0.504]
 [ 1.512 -0.496 -0.503]
 [-0.524 -0.468  1.52 ]
 [-0.529  1.466 -0.504]
 [-0.531  1.282  0.215]
 [ 1.512 -0.496 -0.503]]
```

In the multiclass case, the `decision_function` has the shape `(n_samples, n_classes)` and each column provides a “certainty score” for each class, where a large score means that a class is more likely and a small score means the class is less likely. You can recover the predictions from these scores by finding the maximum entry for each data point:

In[117]:

```
print("Argmax of decision function:\n{}".format(
    np.argmax(gbdt.decision_function(X_test), axis=1)))
print("Predictions:\n{}".format(gbdt.predict(X_test)))
```

Out[117]:

```
Argmax of decision function:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
Predictions:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
```

The output of `predict_proba` has the same shape, `(n_samples, n_classes)`. Again, the probabilities for the possible classes for each data point sum to 1:

In[118]:

```
# show the first few entries of predict_proba
print("Predicted probabilities:\n{}".format(gbrt.predict_proba(X_test)[:6]))
# show that sums across rows are one
print("Sums: {}".format(gbrt.predict_proba(X_test)[:6].sum(axis=1)))
```

Out[118]:

```
Predicted probabilities:
[[ 0.107  0.784  0.109]
 [ 0.789  0.106  0.105]
 [ 0.102  0.108  0.789]
 [ 0.107  0.784  0.109]
 [ 0.108  0.663  0.228]
 [ 0.789  0.106  0.105]]
Sums: [ 1.  1.  1.  1.  1.  1.]
```

We can again recover the predictions by computing the `argmax` of `predict_proba`:

In[119]:

```
print("Argmax of predicted probabilities:\n{}".format(
    np.argmax(gbrt.predict_proba(X_test), axis=1)))
print("Predictions:\n{}".format(gbrt.predict(X_test)))
```

Out[119]:

```
Argmax of predicted probabilities:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
Predictions:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 0 0 0 0 1 0 0 2 1 0]
```

To summarize, `predict_proba` and `decision_function` always have shape `(n_samples, n_classes)`—apart from `decision_function` in the special binary case. In the binary case, `decision_function` only has one column, corresponding to the “positive” class `classes_[1]`. This is mostly for historical reasons.

You can recover the prediction when there are `n_classes` many columns by computing the `argmax` across columns. Be careful, though, if your classes are strings, or you use integers but they are not consecutive and starting from 0. If you want to compare results obtained with `predict` to results obtained via `decision_function` or `predict_proba`, make sure to use the `classes_` attribute of the classifier to get the actual class names:

In[120]:

```
logreg = LogisticRegression()

# represent each target by its class name in the iris dataset
named_target = iris.target_names[y_train]
logreg.fit(X_train, named_target)
print("unique classes in training data: {}".format(logreg.classes_))
print("predictions: {}".format(logreg.predict(X_test)[:10]))
argmax_dec_func = np.argmax(logreg.decision_function(X_test), axis=1)
print("argmax of decision function: {}".format(argmax_dec_func[:10]))
print("argmax combined with classes_: {}".format(
    logreg.classes_[argmax_dec_func][:10]))
```

Out[120]:

```
unique classes in training data: ['setosa' 'versicolor' 'virginica']
predictions: ['versicolor' 'setosa' 'virginica' 'versicolor' 'versicolor'
 'setosa' 'versicolor' 'virginica' 'versicolor' 'versicolor']
argmax of decision function: [1 0 2 1 1 0 1 2 1 1]
argmax combined with classes_: ['versicolor' 'setosa' 'virginica' 'versicolor'
 'versicolor' 'setosa' 'versicolor' 'virginica' 'versicolor' 'versicolor']
```

Summary and Outlook

We started this chapter with a discussion of model complexity, then discussed *generalization*, or learning a model that is able to perform well on new, previously unseen data. This led us to the concepts of underfitting, which describes a model that cannot capture the variations present in the training data, and overfitting, which describes a model that focuses too much on the training data and is not able to generalize to new data very well.

We then discussed a wide array of machine learning models for classification and regression, what their advantages and disadvantages are, and how to control model complexity for each of them. We saw that for many of the algorithms, setting the right parameters is important for good performance. Some of the algorithms are also sensitive to how we represent the input data, and in particular to how the features are scaled. Therefore, blindly applying an algorithm to a dataset without understanding the assumptions the model makes and the meanings of the parameter settings will rarely lead to an accurate model.

This chapter contains a lot of information about the algorithms, and it is not necessary for you to remember all of these details for the following chapters. However, some knowledge of the models described here—and which to use in a specific situation—is important for successfully applying machine learning in practice. Here is a quick summary of when to use each model:

Nearest neighbors

For small datasets, good as a baseline, easy to explain.

Linear models

Go-to as a first algorithm to try, good for very large datasets, good for very high-dimensional data.

Naive Bayes

Only for classification. Even faster than linear models, good for very large datasets and high-dimensional data. Often less accurate than linear models.

Decision trees

Very fast, don't need scaling of the data, can be visualized and easily explained.

Random forests

Nearly always perform better than a single decision tree, very robust and powerful. Don't need scaling of data. Not good for very high-dimensional sparse data.

Gradient boosted decision trees

Often slightly more accurate than random forests. Slower to train but faster to predict than random forests, and smaller in memory. Need more parameter tuning than random forests.

Support vector machines

Powerful for medium-sized datasets of features with similar meaning. Require scaling of data, sensitive to parameters.

Neural networks

Can build very complex models, particularly for large datasets. Sensitive to scaling of the data and to the choice of parameters. Large models need a long time to train.

When working with a new dataset, it is in general a good idea to start with a simple model, such as a linear model or a naive Bayes or nearest neighbors classifier, and see how far you can get. After understanding more about the data, you can consider moving to an algorithm that can build more complex models, such as random forests, gradient boosted decision trees, SVMs, or neural networks.

You should now be in a position where you have some idea of how to apply, tune, and analyze the models we discussed here. In this chapter, we focused on the binary classification case, as this is usually easiest to understand. Most of the algorithms presented have classification and regression variants, however, and all of the classification algorithms support both binary and multiclass classification. Try applying any of these algorithms to the built-in datasets in `scikit-learn`, like the `boston_housing` or `diabetes` datasets for regression, or the `digits` dataset for multiclass classification. Playing around with the algorithms on different datasets will give you a better feel for

how long they need to train, how easy it is to analyze the models, and how sensitive they are to the representation of the data.

While we analyzed the consequences of different parameter settings for the algorithms we investigated, building a model that actually generalizes well to new data in production is a bit trickier than that. We will see how to properly adjust parameters and how to find good parameters automatically in [Chapter 6](#).

First, though, we will dive in more detail into unsupervised learning and preprocessing in the next chapter.

Unsupervised Learning and Preprocessing

The second family of machine learning algorithms that we will discuss is unsupervised learning algorithms. Unsupervised learning subsumes all kinds of machine learning where there is no known output, no teacher to instruct the learning algorithm. In unsupervised learning, the learning algorithm is just shown the input data and asked to extract knowledge from this data.

Types of Unsupervised Learning

We will look into two kinds of unsupervised learning in this chapter: transformations of the dataset and clustering.

Unsupervised transformations of a dataset are algorithms that create a new representation of the data which might be easier for humans or other machine learning algorithms to understand compared to the original representation of the data. A common application of unsupervised transformations is dimensionality reduction, which takes a high-dimensional representation of the data, consisting of many features, and finds a new way to represent this data that summarizes the essential characteristics with fewer features. A common application for dimensionality reduction is reduction to two dimensions for visualization purposes.

Another application for unsupervised transformations is finding the parts or components that “make up” the data. An example of this is topic extraction on collections of text documents. Here, the task is to find the unknown topics that are talked about in each document, and to learn what topics appear in each document. This can be useful for tracking the discussion of themes like elections, gun control, or pop stars on social media.

Clustering algorithms, on the other hand, partition data into distinct groups of similar items. Consider the example of uploading photos to a social media site. To allow you

to organize your pictures, the site might want to group together pictures that show the same person. However, the site doesn't know which pictures show whom, and it doesn't know how many different people appear in your photo collection. A sensible approach would be to extract all the faces and divide them into groups of faces that look similar. Hopefully, these correspond to the same person, and the images can be grouped together for you.

Challenges in Unsupervised Learning

A major challenge in unsupervised learning is evaluating whether the algorithm learned something useful. Unsupervised learning algorithms are usually applied to data that does not contain any label information, so we don't know what the right output should be. Therefore, it is very hard to say whether a model "did well." For example, our hypothetical clustering algorithm could have grouped together all the pictures that show faces in profile and all the full-face pictures. This would certainly be a possible way to divide a collection of pictures of people's faces, but it's not the one we were looking for. However, there is no way for us to "tell" the algorithm what we are looking for, and often the only way to evaluate the result of an unsupervised algorithm is to inspect it manually.

As a consequence, unsupervised algorithms are used often in an exploratory setting, when a data scientist wants to understand the data better, rather than as part of a larger automatic system. Another common application for unsupervised algorithms is as a preprocessing step for supervised algorithms. Learning a new representation of the data can sometimes improve the accuracy of supervised algorithms, or can lead to reduced memory and time consumption.

Before we start with "real" unsupervised algorithms, we will briefly discuss some simple preprocessing methods that often come in handy. Even though preprocessing and scaling are often used in tandem with supervised learning algorithms, scaling methods don't make use of the supervised information, making them unsupervised.

Preprocessing and Scaling

In the previous chapter we saw that some algorithms, like neural networks and SVMs, are very sensitive to the scaling of the data. Therefore, a common practice is to adjust the features so that the data representation is more suitable for these algorithms. Often, this is a simple per-feature rescaling and shift of the data. The following code (Figure 3-1) shows a simple example:

In[2]:

```
mlearn.plots.plot_scaling()
```

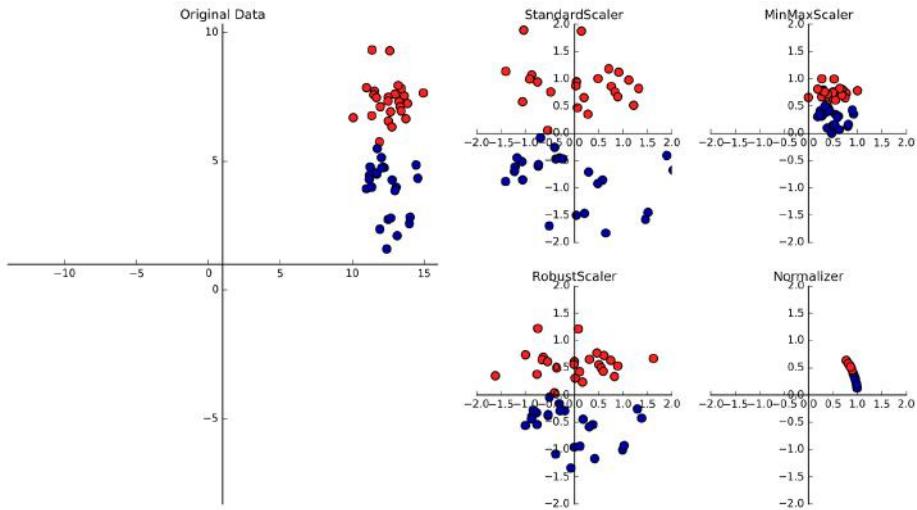


Figure 3-1. Different ways to rescale and preprocess a dataset

Different Kinds of Preprocessing

The first plot in Figure 3-1 shows a synthetic two-class classification dataset with two features. The first feature (the x-axis value) is between 10 and 15. The second feature (the y-axis value) is between around 1 and 9.

The following four plots show four different ways to transform the data that yield more standard ranges. The `StandardScaler` in `scikit-learn` ensures that for each feature the mean is 0 and the variance is 1, bringing all features to the same magnitude. However, this scaling does not ensure any particular minimum and maximum values for the features. The `RobustScaler` works similarly to the `StandardScaler` in that it ensures statistical properties for each feature that guarantee that they are on the same scale. However, the `RobustScaler` uses the median and quartiles,¹ instead of mean and variance. This makes the `RobustScaler` ignore data points that are very different from the rest (like measurement errors). These odd data points are also called *outliers*, and can lead to trouble for other scaling techniques.

The `MinMaxScaler`, on the other hand, shifts the data such that all features are exactly between 0 and 1. For the two-dimensional dataset this means all of the data is con-

¹ The median of a set of numbers is the number x such that half of the numbers are smaller than x and half of the numbers are larger than x . The lower quartile is the number x such that one-fourth of the numbers are smaller than x , and the upper quartile is the number x such that one-fourth of the numbers are larger than x .

tained within the rectangle created by the x-axis between 0 and 1 and the y-axis between 0 and 1.

Finally, the `Normalizer` does a very different kind of rescaling. It scales each data point such that the feature vector has a Euclidean length of 1. In other words, it projects a data point on the circle (or sphere, in the case of higher dimensions) with a radius of 1. This means every data point is scaled by a different number (by the inverse of its length). This normalization is often used when only the direction (or angle) of the data matters, not the length of the feature vector.

Applying Data Transformations

Now that we've seen what the different kinds of transformations do, let's apply them using `scikit-learn`. We will use the `cancer` dataset that we saw in [Chapter 2](#). Pre-processing methods like the scalers are usually applied before applying a supervised machine learning algorithm. As an example, say we want to apply the kernel SVM (`SVC`) to the `cancer` dataset, and use `MinMaxScaler` for preprocessing the data. We start by loading our dataset and splitting it into a training set and a test set (we need separate training and test sets to evaluate the supervised model we will build after the preprocessing):

In[3]:

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=1)
print(X_train.shape)
print(X_test.shape)
```

Out[3]:

```
(426, 30)
(143, 30)
```

As a reminder, the dataset contains 569 data points, each represented by 30 measurements. We split the dataset into 426 samples for the training set and 143 samples for the test set.

As with the supervised models we built earlier, we first import the class that implements the preprocessing, and then instantiate it:

In[4]:

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
```

We then fit the scaler using the `fit` method, applied to the training data. For the `MinMaxScaler`, the `fit` method computes the minimum and maximum value of each feature on the training set. In contrast to the classifiers and regressors of [Chapter 2](#), the scaler is only provided with the data (`X_train`) when `fit` is called, and `y_train` is not used:

In[5]:

```
scaler.fit(X_train)
```

Out[5]:

```
MinMaxScaler(copy=True, feature_range=(0, 1))
```

To apply the transformation that we just learned—that is, to actually *scale* the training data—we use the `transform` method of the scaler. The `transform` method is used in `scikit-learn` whenever a model returns a new representation of the data:

In[6]:

```
# transform data
X_train_scaled = scaler.transform(X_train)
# print dataset properties before and after scaling
print("transformed shape: {}".format(X_train_scaled.shape))
print("per-feature minimum before scaling:\n {}".format(X_train.min(axis=0)))
print("per-feature maximum before scaling:\n {}".format(X_train.max(axis=0)))
print("per-feature minimum after scaling:\n {}".format(
    X_train_scaled.min(axis=0)))
print("per-feature maximum after scaling:\n {}".format(
    X_train_scaled.max(axis=0)))
```

Out[6]:

```
transformed shape: (426, 30)
per-feature minimum before scaling:
 [ 6.98    9.71   43.79  143.50     0.05     0.02     0.      0.      0.11
   0.05    0.12    0.36    0.76    6.80     0.      0.      0.      0.
   0.01    0.      7.93   12.02   50.41   185.20    0.07    0.03    0.
   0.      0.16    0.06]
per-feature maximum before scaling:
 [ 28.11   39.28   188.5   2501.0     0.16     0.29     0.43     0.2
   0.300   0.100   2.87    4.88    21.98   542.20    0.03    0.14
   0.400   0.050   0.06    0.03    36.04   49.54   251.20   4254.00
   0.220   0.940   1.17    0.29    0.58    0.15]
per-feature minimum after scaling:
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
   0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
per-feature maximum after scaling:
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.
   1.  1.  1.  1.  1.  1.  1.  1.  1.]
```

The transformed data has the same shape as the original data—the features are simply shifted and scaled. You can see that all of the features are now between 0 and 1, as desired.

To apply the SVM to the scaled data, we also need to transform the test set. This is again done by calling the `transform` method, this time on `X_test`:

In[7]:

```
# transform test data
X_test_scaled = scaler.transform(X_test)
# print test data properties after scaling
print("per-feature minimum after scaling:\n{}".format(X_test_scaled.min(axis=0)))
print("per-feature maximum after scaling:\n{}".format(X_test_scaled.max(axis=0)))
```

Out[7]:

```
per-feature minimum after scaling:
[ 0.034  0.023  0.031  0.011  0.141  0.044  0.       0.       0.154 -0.006
 -0.001  0.006  0.004  0.001  0.039  0.011  0.       0.       -0.032  0.007
  0.027  0.058  0.02   0.009  0.109  0.026  0.       0.       -0.       -0.002]
per-feature maximum after scaling:
[ 0.958  0.815  0.956  0.894  0.811  1.22   0.88   0.933  0.932  1.037
  0.427  0.498  0.441  0.284  0.487  0.739  0.767  0.629  1.337  0.391
  0.896  0.793  0.849  0.745  0.915  1.132  1.07   0.924  1.205  1.631]
```

Maybe somewhat surprisingly, you can see that for the test set, after scaling, the minimum and maximum are not 0 and 1. Some of the features are even outside the 0–1 range! The explanation is that the `MinMaxScaler` (and all the other scalers) always applies exactly the same transformation to the training and the test set. This means the `transform` method always subtracts the training set minimum and divides by the training set range, which might be different from the minimum and range for the test set.

Scaling Training and Test Data the Same Way

It is important to apply exactly the same transformation to the training set and the test set for the supervised model to work on the test set. The following example (Figure 3-2) illustrates what would happen if we were to use the minimum and range of the test set instead:

In[8]:

```
from sklearn.datasets import make_blobs
# make synthetic data
X, _ = make_blobs(n_samples=50, centers=5, random_state=4, cluster_std=2)
# split it into training and test sets
X_train, X_test = train_test_split(X, random_state=5, test_size=.1)

# plot the training and test sets
fig, axes = plt.subplots(1, 3, figsize=(13, 4))
```

```

axes[0].scatter(X_train[:, 0], X_train[:, 1],
                c=mlearn.cm2(0), label="Training set", s=60)
axes[0].scatter(X_test[:, 0], X_test[:, 1], marker='^',
                c=mlearn.cm2(1), label="Test set", s=60)
axes[0].legend(loc='upper left')
axes[0].set_title("Original Data")

# scale the data using MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# visualize the properly scaled data
axes[1].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1],
                c=mlearn.cm2(0), label="Training set", s=60)
axes[1].scatter(X_test_scaled[:, 0], X_test_scaled[:, 1], marker='^',
                c=mlearn.cm2(1), label="Test set", s=60)
axes[1].set_title("Scaled Data")

# rescale the test set separately
# so test set min is 0 and test set max is 1
# DO NOT DO THIS! For illustration purposes only.
test_scaler = MinMaxScaler()
test_scaler.fit(X_test)
X_test_scaled_badly = test_scaler.transform(X_test)

# visualize wrongly scaled data
axes[2].scatter(X_train_scaled[:, 0], X_train_scaled[:, 1],
                c=mlearn.cm2(0), label="training set", s=60)
axes[2].scatter(X_test_scaled_badly[:, 0], X_test_scaled_badly[:, 1],
                marker='^', c=mlearn.cm2(1), label="test set", s=60)
axes[2].set_title("Improperly Scaled Data")

for ax in axes:
    ax.set_xlabel("Feature 0")
    ax.set_ylabel("Feature 1")

```

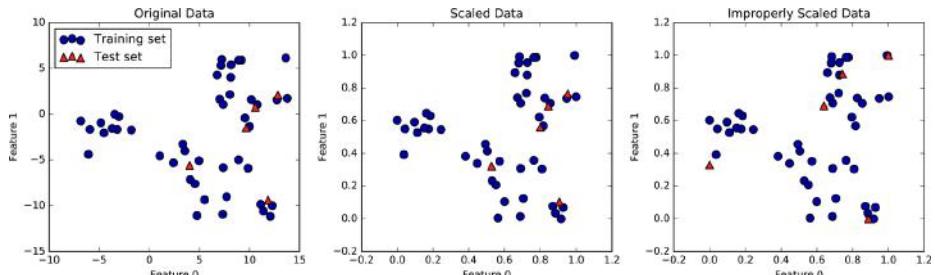


Figure 3-2. Effect of scaling training and test data shown on the left together (center) and separately (right)

The first panel is an unscaled two-dimensional dataset, with the training set shown as circles and the test set shown as triangles. The second panel is the same data, but scaled using the `MinMaxScaler`. Here, we called `fit` on the training set, and then called `transform` on the training and test sets. You can see that the dataset in the second panel looks identical to the first; only the ticks on the axes have changed. Now all the features are between 0 and 1. You can also see that the minimum and maximum feature values for the test data (the triangles) are not 0 and 1.

The third panel shows what would happen if we scaled the training set and test set separately. In this case, the minimum and maximum feature values for both the training and the test set are 0 and 1. But now the dataset looks different. The test points moved incongruously to the training set, as they were scaled differently. We changed the arrangement of the data in an arbitrary way. Clearly this is not what we want to do.

As another way to think about this, imagine your test set is a single point. There is no way to scale a single point correctly, to fulfill the minimum and maximum requirements of the `MinMaxScaler`. But the size of your test set should not change your processing.

Shortcuts and Efficient Alternatives

Often, you want to `fit` a model on some dataset, and then `transform` it. This is a very common task, which can often be computed more efficiently than by simply calling `fit` and then `transform`. For this use case, all models that have a `transform` method also have a `fit_transform` method. Here is an example using `StandardScaler`:

In[9]:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# calling fit and transform in sequence (using method chaining)
X_scaled = scaler.fit(X).transform(X)
# same result, but more efficient computation
X_scaled_d = scaler.fit_transform(X)
```

While `fit_transform` is not necessarily more efficient for all models, it is still good practice to use this method when trying to transform the training set.

The Effect of Preprocessing on Supervised Learning

Now let's go back to the cancer dataset and see the effect of using the `MinMaxScaler` on learning the SVC (this is a different way of doing the same scaling we did in [Chapter 2](#)). First, let's fit the SVC on the original data again for comparison:

In[10]:

```
from sklearn.svm import SVC

X_train, X_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
                                                    random_state=0)

svm = SVC(C=100)
svm.fit(X_train, y_train)
print("Test set accuracy: {:.2f}".format(svm.score(X_test, y_test)))
```

Out[10]:

```
Test set accuracy: 0.63
```

Now, let's scale the data using `MinMaxScaler` before fitting the SVC:

In[11]:

```
# preprocessing using 0-1 scaling
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# learning an SVM on the scaled training data
svm.fit(X_train_scaled, y_train)

# scoring on the scaled test set
print("Scaled test set accuracy: {:.2f}".format(
    svm.score(X_test_scaled, y_test)))
```

Out[11]:

```
Scaled test set accuracy: 0.97
```

As we saw before, the effect of scaling the data is quite significant. Even though scaling the data doesn't involve any complicated math, it is good practice to use the scaling mechanisms provided by `scikit-learn` instead of reimplementing them yourself, as it's easy to make mistakes even in these simple computations.

You can also easily replace one preprocessing algorithm with another by changing the class you use, as all of the preprocessing classes have the same interface, consisting of the `fit` and `transform` methods:

In[12]:

```
# preprocessing using zero mean and unit variance scaling
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
# learning an SVM on the scaled training data
svm.fit(X_train_scaled, y_train)

# scoring on the scaled test set
print("SVM test accuracy: {:.2f}".format(svm.score(X_test_scaled, y_test)))
```

Out[12]:

```
SVM test accuracy: 0.96
```

Now that we've seen how simple data transformations for preprocessing work, let's move on to more interesting transformations using unsupervised learning.

Dimensionality Reduction, Feature Extraction, and Manifold Learning

As we discussed earlier, transforming data using unsupervised learning can have many motivations. The most common motivations are visualization, compressing the data, and finding a representation that is more informative for further processing.

One of the simplest and most widely used algorithms for all of these is principal component analysis. We'll also look at two other algorithms: non-negative matrix factorization (NMF), which is commonly used for feature extraction, and t-SNE, which is commonly used for visualization using two-dimensional scatter plots.

Principal Component Analysis (PCA)

Principal component analysis is a method that rotates the dataset in a way such that the rotated features are statistically uncorrelated. This rotation is often followed by selecting only a subset of the new features, according to how important they are for explaining the data. The following example (Figure 3-3) illustrates the effect of PCA on a synthetic two-dimensional dataset:

In[13]:

```
mlearn.plots.plot_pca_illustration()
```

The first plot (top left) shows the original data points, colored to distinguish among them. The algorithm proceeds by first finding the direction of maximum variance, labeled "Component 1." This is the direction (or vector) in the data that contains most of the information, or in other words, the direction along which the features are most correlated with each other. Then, the algorithm finds the direction that contains the most information while being orthogonal (at a right angle) to the first direction. In two dimensions, there is only one possible orientation that is at a right angle, but in higher-dimensional spaces there would be (infinitely) many orthogonal directions. Although the two components are drawn as arrows, it doesn't really matter where the head and the tail are; we could have drawn the first component from the center up to

the top left instead of down to the bottom right. The directions found using this process are called *principal components*, as they are the main directions of variance in the data. In general, there are as many principal components as original features.

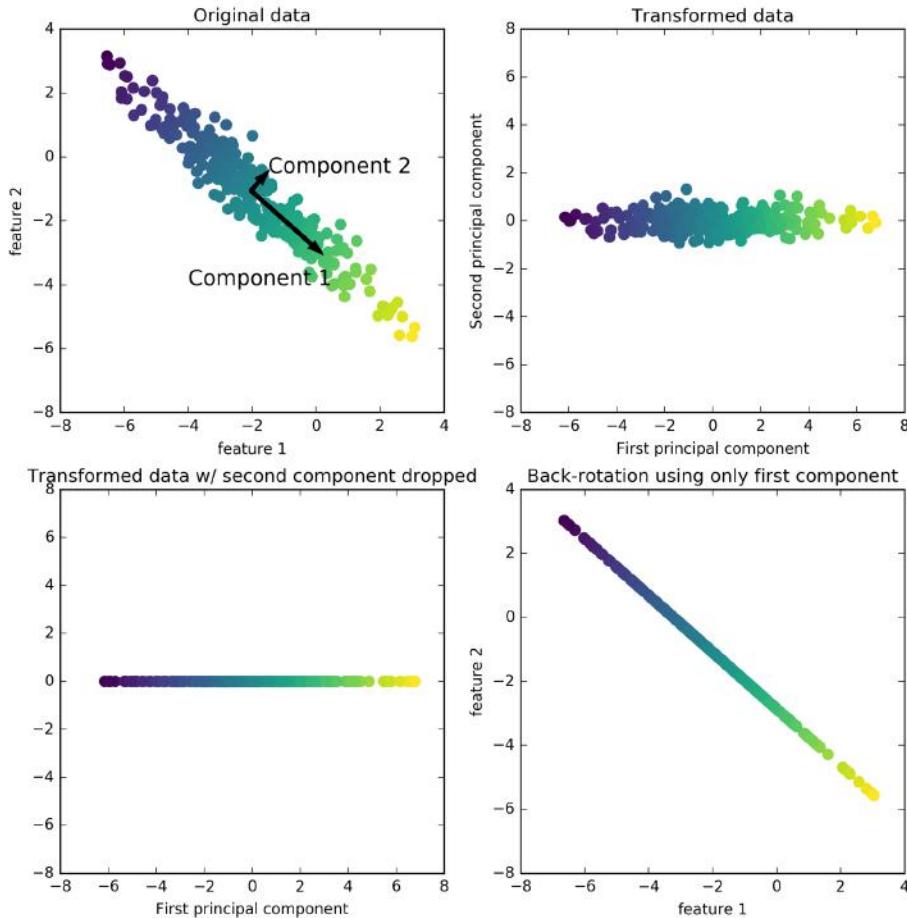


Figure 3-3. Transformation of data with PCA

The second plot (top right) shows the same data, but now rotated so that the first principal component aligns with the x-axis and the second principal component aligns with the y-axis. Before the rotation, the mean was subtracted from the data, so that the transformed data is centered around zero. In the rotated representation found by PCA, the two axes are uncorrelated, meaning that the correlation matrix of the data in this representation is zero except for the diagonal.

We can use PCA for dimensionality reduction by retaining only some of the principal components. In this example, we might keep only the first principal component, as

shown in the third panel in [Figure 3-3](#) (bottom left). This reduces the data from a two-dimensional dataset to a one-dimensional dataset. Note, however, that instead of keeping only one of the original features, we found the most interesting direction (top left to bottom right in the first panel) and kept this direction, the first principal component.

Finally, we can undo the rotation and add the mean back to the data. This will result in the data shown in the last panel in [Figure 3-3](#). These points are in the original feature space, but we kept only the information contained in the first principal component. This transformation is sometimes used to remove noise effects from the data or visualize what part of the information is retained using the principal components.

Applying PCA to the cancer dataset for visualization

One of the most common applications of PCA is visualizing high-dimensional datasets. As we saw in [Chapter 1](#), it is hard to create scatter plots of data that has more than two features. For the Iris dataset, we were able to create a pair plot ([Figure 1-3](#) in [Chapter 1](#)) that gave us a partial picture of the data by showing us all the possible combinations of two features. But if we want to look at the Breast Cancer dataset, even using a pair plot is tricky. This dataset has 30 features, which would result in $30 * 14 = 420$ scatter plots! We'd never be able to look at all these plots in detail, let alone try to understand them.

There is an even simpler visualization we can use, though—computing histograms of each of the features for the two classes, benign and malignant cancer ([Figure 3-4](#)):

In[14]:

```
fig, axes = plt.subplots(15, 2, figsize=(10, 20))
malignant = cancer.data[cancer.target == 0]
benign = cancer.data[cancer.target == 1]

ax = axes.ravel()

for i in range(30):
    _, bins = np.histogram(cancer.data[:, i], bins=50)
    ax[i].hist(malignant[:, i], bins=bins, color=mglearn.cm3(0), alpha=.5)
    ax[i].hist(benign[:, i], bins=bins, color=mglearn.cm3(2), alpha=.5)
    ax[i].set_title(cancer.feature_names[i])
    ax[i].set_yticks(())
ax[0].set_xlabel("Feature magnitude")
ax[0].set_ylabel("Frequency")
ax[0].legend(["malignant", "benign"], loc="best")
fig.tight_layout()
```

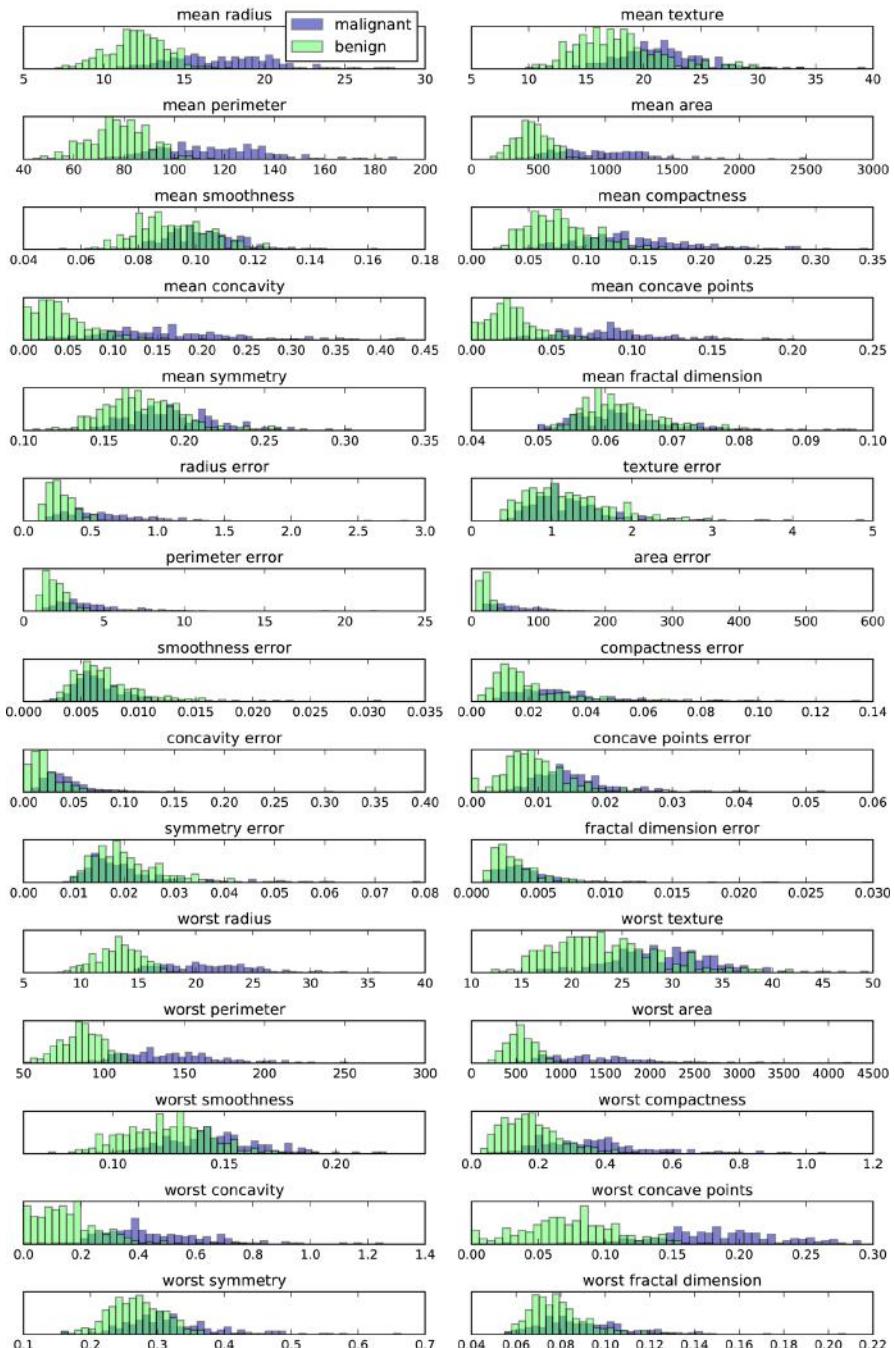


Figure 3-4. Per-class feature histograms on the Breast Cancer dataset

Here we create a histogram for each of the features, counting how often a data point appears with a feature in a certain range (called a *bin*). Each plot overlays two histograms, one for all of the points in the benign class (blue) and one for all the points in the malignant class (red). This gives us some idea of how each feature is distributed across the two classes, and allows us to venture a guess as to which features are better at distinguishing malignant and benign samples. For example, the feature “smoothness error” seems quite uninformative, because the two histograms mostly overlap, while the feature “worst concave points” seems quite informative, because the histograms are quite disjoint.

However, this plot doesn’t show us anything about the interactions between variables and how these relate to the classes. Using PCA, we can capture the main interactions and get a slightly more complete picture. We can find the first two principal components, and visualize the data in this new two-dimensional space with a single scatter plot.

Before we apply PCA, we scale our data so that each feature has unit variance using `StandardScaler`:

In[15]:

```
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()

scaler = StandardScaler()
scaler.fit(cancer.data)
X_scaled = scaler.transform(cancer.data)
```

Learning the PCA transformation and applying it is as simple as applying a preprocessing transformation. We instantiate the PCA object, find the principal components by calling the `fit` method, and then apply the rotation and dimensionality reduction by calling `transform`. By default, PCA only rotates (and shifts) the data, but keeps all principal components. To reduce the dimensionality of the data, we need to specify how many components we want to keep when creating the PCA object:

In[16]:

```
from sklearn.decomposition import PCA
# keep the first two principal components of the data
pca = PCA(n_components=2)
# fit PCA model to breast cancer data
pca.fit(X_scaled)

# transform data onto the first two principal components
X_pca = pca.transform(X_scaled)
print("Original shape: {}".format(str(X_scaled.shape)))
print("Reduced shape: {}".format(str(X_pca.shape)))
```

Out[16]:

```
Original shape: (569, 30)
Reduced shape: (569, 2)
```

We can now plot the first two principal components (Figure 3-5):

In[17]:

```
# plot first vs. second principal component, colored by class
plt.figure(figsize=(8, 8))
mglearn.discrete_scatter(X_pca[:, 0], X_pca[:, 1], cancer.target)
plt.legend(cancer.target_names, loc="best")
plt.gca().set_aspect("equal")
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```

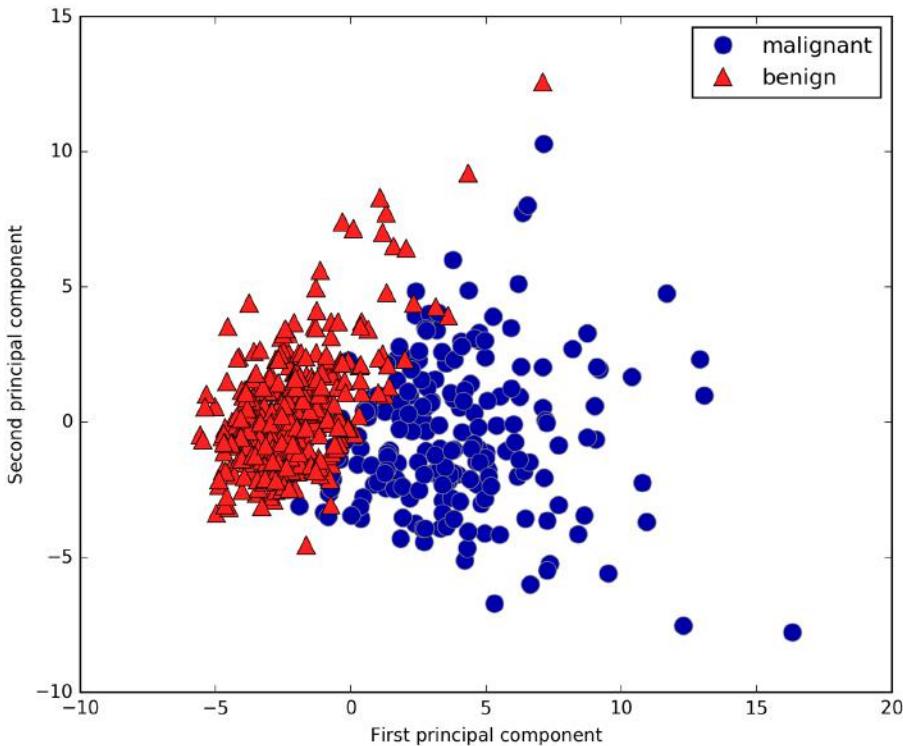


Figure 3-5. Two-dimensional scatter plot of the Breast Cancer dataset using the first two principal components

It is important to note that PCA is an unsupervised method, and does not use any class information when finding the rotation. It simply looks at the correlations in the data. For the scatter plot shown here, we plotted the first principal component against the

second principal component, and then used the class information to color the points. You can see that the two classes separate quite well in this two-dimensional space. This leads us to believe that even a linear classifier (that would learn a line in this space) could do a reasonably good job at distinguishing the two classes. We can also see that the malignant (red) points are more spread out than the benign (blue) points —something that we could already see a bit from the histograms in [Figure 3-4](#).

A downside of PCA is that the two axes in the plot are often not very easy to interpret. The principal components correspond to directions in the original data, so they are combinations of the original features. However, these combinations are usually very complex, as we'll see shortly. The principal components themselves are stored in the `components_` attribute of the PCA object during fitting:

In[18]:

```
print("PCA component shape: {}".format(pca.components_.shape))
```

Out[18]:

```
PCA component shape: (2, 30)
```

Each row in `components_` corresponds to one principal component, and they are sorted by their importance (the first principal component comes first, etc.). The columns correspond to the `original features` attribute of the PCA in this example, “mean radius,” “mean texture,” and so on. Let's have a look at the content of `components_`:

In[19]:

```
print("PCA components:\n{}".format(pca.components_))
```

Out[19]:

```
PCA components:  
[[ 0.219  0.104  0.228  0.221  0.143  0.239  0.258  0.261  0.138  0.064  
   0.206  0.017  0.211  0.203  0.015  0.17   0.154  0.183  0.042  0.103  
   0.228  0.104  0.237  0.225  0.128  0.21   0.229  0.251  0.123  0.132]  
[-0.234 -0.06  -0.215 -0.231  0.186  0.152  0.06  -0.035  0.19   0.367  
 -0.106  0.09  -0.089 -0.152  0.204  0.233  0.197  0.13   0.184  0.28  
 -0.22  -0.045 -0.2   -0.219  0.172  0.144  0.098 -0.008  0.142  0.275]]
```

We can also visualize the coefficients using a heat map ([Figure 3-6](#)), which might be easier to understand:

In[20]:

```
plt.matshow(pca.components_, cmap='viridis')  
plt.yticks([0, 1], ["First component", "Second component"])  
plt.colorbar()  
plt.xticks(range(len(cancer.feature_names)),  
           cancer.feature_names, rotation=60, ha='left')  
plt.xlabel("Feature")  
plt.ylabel("Principal components")
```

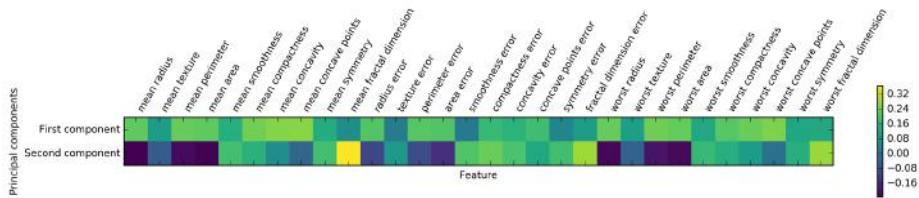


Figure 3-6. Heat map of the first two principal components on the Breast Cancer dataset

You can see that in the first component, all features have the same sign (it's negative, but as we mentioned earlier, it doesn't matter which direction the arrow points in). That means that there is a general correlation between all features. As one measurement is high, the others are likely to be high as well. The second component has mixed signs, and both of the components involve all of the 30 features. This mixing of all features is what makes explaining the axes in [Figure 3-6](#) so tricky.

Eigenfaces for feature extraction

Another application of PCA that we mentioned earlier is feature extraction. The idea behind feature extraction is that it is possible to find a representation of your data that is better suited to analysis than the raw representation you were given. A great example of an application where feature extraction is helpful is with images. Images are made up of pixels, usually stored as red, green, and blue (RGB) intensities. Objects in images are usually made up of thousands of pixels, and only together are they meaningful.

We will give a very simple application of feature extraction on images using PCA, by working with face images from the Labeled Faces in the Wild dataset. This dataset contains face images of celebrities downloaded from the Internet, and it includes faces of politicians, singers, actors, and athletes from the early 2000s. We use gray-scale versions of these images, and scale them down for faster processing. You can see some of the images in [Figure 3-7](#):

In[21]:

```
from sklearn.datasets import fetch_lfw_people
people = fetch_lfw_people(min_faces_per_person=20, resize=0.7)
image_shape = people.images[0].shape

fix, axes = plt.subplots(2, 5, figsize=(15, 8),
                       subplot_kw={'xticks': (), 'yticks': ()})
for target, image, ax in zip(people.target, people.images, axes.ravel()):
    ax.imshow(image)
    ax.set_title(people.target_names[target])
```



Figure 3-7. Some images from the Labeled Faces in the Wild dataset

There are 3,023 images, each 87×65 pixels large, belonging to 62 different people:

In[22]:

```
print("people.images.shape: {}".format(people.images.shape))
print("Number of classes: {}".format(len(people.target_names)))
```

Out[22]:

```
people.images.shape: (3023, 87, 65)
Number of classes: 62
```

The dataset is a bit skewed, however, containing a lot of images of George W. Bush and Colin Powell, as you can see here:

In[23]:

```
# count how often each target appears
counts = np.bincount(people.target)
# print counts next to target names
for i, (count, name) in enumerate(zip(counts, people.target_names)):
    print("{}: {} {}".format(name, count, "" if i % 3 == 2 else "\n"))
    if (i + 1) % 3 == 0:
        print()
```

Out[23]:

Alejandro Toledo	39	Alvaro Uribe	35
Amelie Mauresmo	21	Andre Agassi	36
Angelina Jolie	20	Arnold Schwarzenegger	42
Atal Bihari Vajpayee	24	Bill Clinton	29
Carlos Menem	21	Colin Powell	236
David Beckham	31	Donald Rumsfeld	121
George W Bush	530	George Robertson	22
Gerhard Schroeder	109	Gloria Macapagal Arroyo	44
Gray Davis	26	Guillermo Coria	30
Hamid Karzai	22	Hans Blix	39
Hugo Chavez	71	Igor Ivanov	20
[...]	[...]		
Laura Bush	41	Lindsay Davenport	22
Lleyton Hewitt	41	Luiz Inacio Lula da Silva	48
Mahmoud Abbas	29	Megawati Sukarnoputri	33
Michael Bloomberg	20	Naomi Watts	22
Nestor Kirchner	37	Paul Bremer	20
Pete Sampras	22	Recep Tayyip Erdogan	30
Ricardo Lagos	27	Roh Moo-hyun	32
Rudolph Giuliani	26	Saddam Hussein	23
Serena Williams	52	Silvio Berlusconi	33
Tiger Woods	23	Tom Daschle	25
Tom Ridge	33	Tony Blair	144
Vicente Fox	32	Vladimir Putin	49
Winona Ryder	24		

To make the data less skewed, we will only take up to 50 images of each person (otherwise, the feature extraction would be overwhelmed by the likelihood of George W. Bush):

In[24]:

```
mask = np.zeros(people.target.shape, dtype=np.bool)
for target in np.unique(people.target):
    mask[np.where(people.target == target)[0][:50]] = 1

X_people = people.data[mask]
y_people = people.target[mask]

# scale the grayscale values to be between 0 and 1
# instead of 0 and 255 for better numeric stability
X_people = X_people / 255.
```

A common task in face recognition is to ask if a previously unseen face belongs to a known person from a database. This has applications in photo collection, social media, and security applications. One way to solve this problem would be to build a classifier where each person is a separate class. However, there are usually many different people in face databases, and very few images of the same person (i.e., very few training examples per class). That makes it hard to train most classifiers. Additionally,

you often want to be able to add new people easily, without needing to retrain a large model.

A simple solution is to use a one-nearest-neighbor classifier that looks for the most similar face image to the face you are classifying. This classifier could in principle work with only a single training example per class. Let's take a look at how well `KNeighborsClassifier` does here:

In[25]:

```
from sklearn.neighbors import KNeighborsClassifier
# split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X_people, y_people, stratify=y_people, random_state=0)
# build a KNeighborsClassifier using one neighbor
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train, y_train)
print("Test set score of 1-nn: {:.2f}".format(knn.score(X_test, y_test)))
```

Out[25]:

```
Test set score of 1-nn: 0.27
```

We obtain an accuracy of 26.6%, which is not actually that bad for a 62-class classification problem (random guessing would give you around $1/62 = 1.5\%$ accuracy), but is also not great. We only correctly identify a person every fourth time.

This is where PCA comes in. Computing distances in the original pixel space is quite a bad way to measure similarity between faces. When using a pixel representation to compare two images, we compare the grayscale value of each individual pixel to the value of the pixel in the corresponding position in the other image. This representation is quite different from how humans would interpret the image of a face, and it is hard to capture the facial features using this raw representation. For example, using pixel distances means that shifting a face by one pixel to the right corresponds to a drastic change, with a completely different representation. We hope that using distances along principal components can improve our accuracy. Here, we enable the `whitening` option of PCA, which rescales the principal components to have the same scale. This is the same as using `StandardScaler` after the transformation. Reusing the data from [Figure 3-3](#) again, whitening corresponds to not only rotating the data, but also rescaling it so that the center panel is a circle instead of an ellipse (see [Figure 3-8](#)):

In[26]:

```
mlearn.plots.plot_pca_whitening()
```

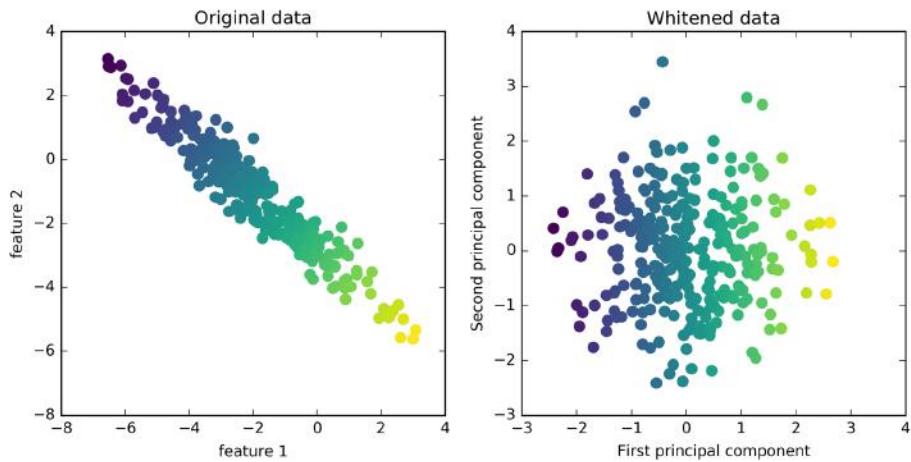


Figure 3-8. Transformation of data with PCA using whitening

We fit the PCA object to the training data and extract the first 100 principal components. Then we transform the training and test data:

In[27]:

```
pca = PCA(n_components=100, whiten=True, random_state=0).fit(X_train)
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)

print("X_train_pca.shape: {}".format(X_train_pca.shape))
```

Out[27]:

```
X_train_pca.shape: (1537, 100)
```

The new data has 100 features, the first 100 principal components. Now, we can use the new representation to classify our images using a one-nearest-neighbors classifier:

In[28]:

```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(X_train_pca, y_train)
print("Test set accuracy: {:.2f}".format(knn.score(X_test_pca, y_test)))
```

Out[28]:

```
Test set accuracy: 0.36
```

Our accuracy improved quite significantly, from 26.6% to 35.7%, confirming our intuition that the principal components might provide a better representation of the data.

For image data, we can also easily visualize the principal components that are found. Remember that components correspond to directions in the input space. The input space here is 50×37-pixel grayscale images, so directions within this space are also 50×37-pixel grayscale images.

Let's look at the first couple of principal components (Figure 3-9):

In[29]:

```
print("pca.components_.shape: {}".format(pca.components_.shape))
```

Out[29]:

```
pca.components_.shape: (100, 5655)
```

In[30]:

```
fix, axes = plt.subplots(3, 5, figsize=(15, 12),
                       subplot_kw={'xticks': (), 'yticks': ()})
for i, (component, ax) in enumerate(zip(pca.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape),
              cmap='viridis')
    ax.set_title("{} component".format((i + 1)))
```

While we certainly cannot understand all aspects of these components, we can guess which aspects of the face images some of the components are capturing. The first component seems to mostly encode the contrast between the face and the background, the second component encodes differences in lighting between the right and the left half of the face, and so on. While this representation is slightly more semantic than the raw pixel values, it is still quite far from how a human might perceive a face. As the PCA model is based on pixels, the alignment of the face (the position of eyes, chin, and nose) and the lighting both have a strong influence on how similar two images are in their pixel representation. But alignment and lighting are probably not what a human would perceive first. When asking people to rate similarity of faces, they are more likely to use attributes like age, gender, facial expression, and hair style, which are attributes that are hard to infer from the pixel intensities. It's important to keep in mind that algorithms often interpret data (particularly visual data, such as images, which humans are very familiar with) quite differently from how a human would.

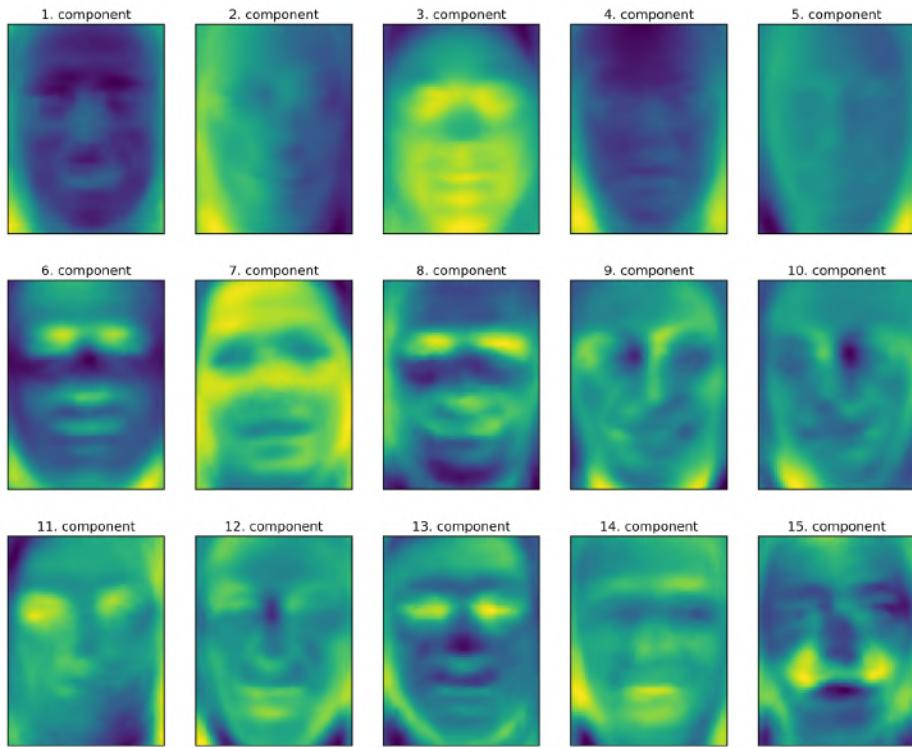


Figure 3-9. Component vectors of the first 15 principal components of the faces dataset

Let's come back to the specific case of PCA, though. We introduced the PCA transformation as rotating the data and then dropping the components with low variance. Another useful interpretation is to try to find some numbers (the new feature values after the PCA rotation) so that we can express the test points as a weighted sum of the principal components (see [Figure 3-10](#)).

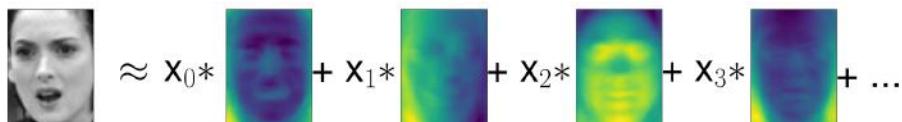


Figure 3-10. Schematic view of PCA as decomposing an image into a weighted sum of components

Here, x_0 , x_1 , and so on are the coefficients of the principal components for this data point; in other words, they are the representation of the image in the rotated space.

Another way we can try to understand what a PCA model is doing is by looking at the reconstructions of the original data using only some components. In [Figure 3-3](#), after dropping the second component and arriving at the third panel, we undid the rotation and added the mean back to obtain new points in the original space with the second component removed, as shown in the last panel. We can do a similar transformation for the faces by reducing the data to only some principal components and then rotating back into the original space. This return to the original feature space can be done using the `inverse_transform` method. Here, we visualize the reconstruction of some faces using 10, 50, 100, 500, or 2,000 components ([Figure 3-11](#)):

In[32]:

```
mlearn.plots.plot_pca_faces(X_train, X_test, image_shape)
```



Figure 3-11. Reconstructing three face images using increasing numbers of principal components

You can see that when we use only the first 10 principal components, only the essence of the picture, like the face orientation and lighting, is captured. By using more and more principal components, more and more details in the image are preserved. This

corresponds to extending the sum in [Figure 3-10](#) to include more and more terms. Using as many components as there are pixels would mean that we would not discard any information after the rotation, and we would reconstruct the image perfectly.

We can also try to use PCA to visualize all the faces in the dataset in a scatter plot using the first two principal components ([Figure 3-12](#)), with classes given by who is shown in the image, similarly to what we did for the cancer dataset:

In[33]:

```
mglearn.discrete_scatter(X_train_pca[:, 0], X_train_pca[:, 1], y_train)
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```

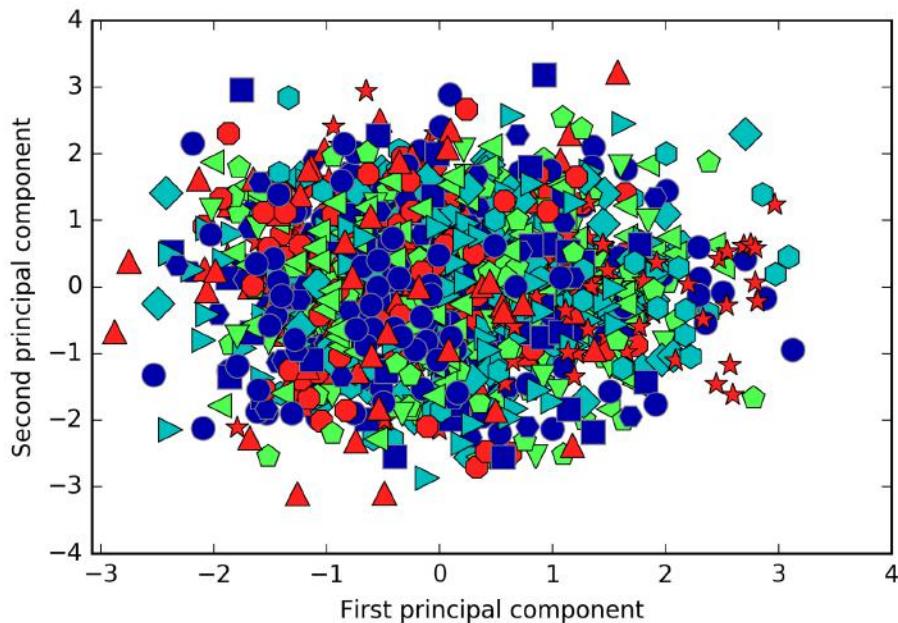


Figure 3-12. Scatter plot of the faces dataset using the first two principal components (see [Figure 3-5](#) for the corresponding image for the cancer dataset)

As you can see, when we use only the first two principal components the whole data is just a big blob, with no separation of classes visible. This is not very surprising, given that even with 10 components, as shown earlier in [Figure 3-11](#), PCA only captures very rough characteristics of the faces.

Non-Negative Matrix Factorization (NMF)

Non-negative matrix factorization is another unsupervised learning algorithm that aims to extract useful features. It works similarly to PCA and can also be used for dimensionality reduction. As in PCA, we are trying to write each data point as a weighted sum of some components, as illustrated in [Figure 3-10](#). But whereas in PCA we wanted components that were orthogonal and that explained as much variance of the data as possible, in NMF, we want the components and the coefficients to be non-negative; that is, we want both the components and the coefficients to be greater than or equal to zero. Consequently, this method can only be applied to data where each feature is non-negative, as a non-negative sum of non-negative components cannot become negative.

The process of decomposing data into a non-negative weighted sum is particularly helpful for data that is created as the addition (or overlay) of several independent sources, such as an audio track of multiple people speaking, or music with many instruments. In these situations, NMF can identify the original components that make up the combined data. Overall, NMF leads to more interpretable components than PCA, as negative components and coefficients can lead to hard-to-interpret cancellation effects. The eigenfaces in [Figure 3-9](#), for example, contain both positive and negative parts, and as we mentioned in the description of PCA, the sign is actually arbitrary. Before we apply NMF to the face dataset, let's briefly revisit the synthetic data.

Applying NMF to synthetic data

In contrast to when using PCA, we need to ensure that our data is positive for NMF to be able to operate on the data. This means where the data lies relative to the origin $(0, 0)$ actually matters for NMF. Therefore, you can think of the non-negative components that are extracted as directions from $(0, 0)$ toward the data.

The following example ([Figure 3-13](#)) shows the results of NMF on the two-dimensional toy data:

In[34]:

```
mlearn.plots.plot_nmf_illustration()
```

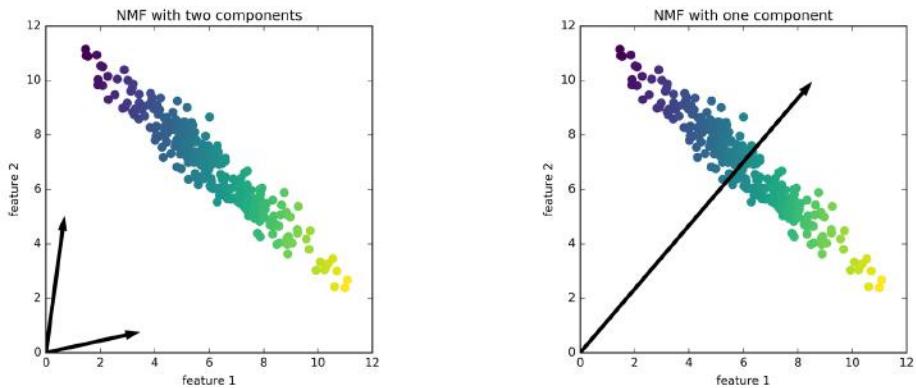


Figure 3-13. Components found by non-negative matrix factorization with two components (left) and one component (right)

For NMF with two components, as shown on the left, it is clear that all points in the data can be written as a positive combination of the two components. If there are enough components to perfectly reconstruct the data (as many components as there are features), the algorithm will choose directions that point toward the extremes of the data.

If we only use a single component, NMF creates a component that points toward the mean, as pointing there best explains the data. You can see that in contrast with PCA, reducing the number of components not only removes some directions, but creates an entirely different set of components! Components in NMF are also not ordered in any specific way, so there is no “first non-negative component”: all components play an equal part.

NMF uses a random initialization, which might lead to different results depending on the random seed. In relatively simple cases such as the synthetic data with two components, where all the data can be explained perfectly, the randomness has little effect (though it might change the order or scale of the components). In more complex situations, there might be more drastic changes.

Applying NMF to face images

Now, let’s apply NMF to the Labeled Faces in the Wild dataset we used earlier. The main parameter of NMF is how many components we want to extract. Usually this is lower than the number of input features (otherwise, the data could be explained by making each pixel a separate component).

First, let’s inspect how the number of components impacts how well the data can be reconstructed using NMF ([Figure 3-14](#)):

In[35]:

```
mglearn.plots.plot_nmf_faces(X_train, X_test, image_shape)
```



Figure 3-14. Reconstructing three face images using increasing numbers of components found by NMF

The quality of the back-transformed data is similar to when using PCA, but slightly worse. This is expected, as PCA finds the optimum directions in terms of reconstruction. NMF is usually not used for its ability to reconstruct or encode data, but rather for finding interesting patterns within the data.

As a first look into the data, let's try extracting only a few components (say, 15). [Figure 3-15](#) shows the result:

In[36]:

```
from sklearn.decomposition import NMF
nmf = NMF(n_components=15, random_state=0)
nmf.fit(X_train)
X_train_nmf = nmf.transform(X_train)
X_test_nmf = nmf.transform(X_test)

fix, axes = plt.subplots(3, 5, figsize=(15, 12),
                       subplot_kw={'xticks': (), 'yticks': ()})
for i, (component, ax) in enumerate(zip(nmf.components_, axes.ravel())):
    ax.imshow(component.reshape(image_shape))
    ax.set_title("{} component".format(i))
```



Figure 3-15. The components found by NMF on the faces dataset when using 15 components

These components are all positive, and so resemble prototypes of faces much more so than the components shown for PCA in Figure 3-9. For example, one can clearly see that component 3 shows a face rotated somewhat to the right, while component 7 shows a face somewhat rotated to the left. Let's look at the images for which these components are particularly strong, shown in Figures 3-16 and 3-17:

In[37]:

```
compn = 3
# sort by 3rd component, plot first 10 images
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                        subplot_kw={'xticks': (), 'yticks': ()})
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
    ax.imshow(X_train[ind].reshape(image_shape))

compn = 7
# sort by 7th component, plot first 10 images
inds = np.argsort(X_train_nmf[:, compn])[::-1]
fig, axes = plt.subplots(2, 5, figsize=(15, 8),
                        subplot_kw={'xticks': (), 'yticks': ()})
for i, (ind, ax) in enumerate(zip(inds, axes.ravel())):
    ax.imshow(X_train[ind].reshape(image_shape))
```



Figure 3-16. Faces that have a large coefficient for component 3



Figure 3-17. Faces that have a large coefficient for component 7

As expected, faces that have a high coefficient for component 3 are faces looking to the right ([Figure 3-16](#)), while faces with a high coefficient for component 7 are looking to the left ([Figure 3-17](#)). As mentioned earlier, extracting patterns like these works best for data with additive structure, including audio, gene expression, and text data. Let's walk through one example on synthetic data to see what this might look like.

Let's say we are interested in a signal that is a combination of three different sources ([Figure 3-18](#)):

In[38]:

```
S = mglearn.datasets.make_signals()
plt.figure(figsize=(6, 1))
plt.plot(S, '-')
plt.xlabel("Time")
plt.ylabel("Signal")
```

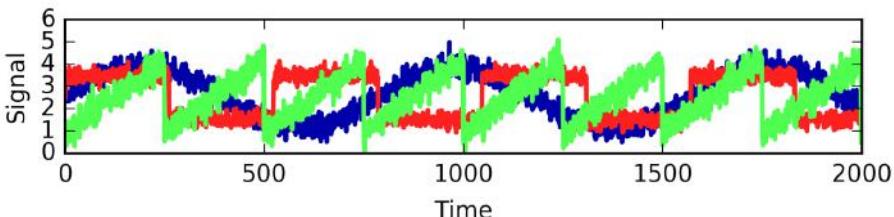


Figure 3-18. Original signal sources

Unfortunately we cannot observe the original signals, but only an additive mixture of all three of them. We want to recover the decomposition of the mixed signal into the original components. We assume that we have many different ways to observe the mixture (say 100 measurement devices), each of which provides us with a series of measurements:

In[39]:

```
# mix data into a 100-dimensional state
A = np.random.RandomState(0).uniform(size=(100, 3))
X = np.dot(S, A.T)
print("Shape of measurements: {}".format(X.shape))
```

Out[39]:

```
Shape of measurements: (2000, 100)
```

We can use NMF to recover the three signals:

In[40]:

```
nmf = NMF(n_components=3, random_state=42)
S_ = nmf.fit_transform(X)
print("Recovered signal shape: {}".format(S_.shape))
```

Out[40]:

```
Recovered signal shape: (2000, 3)
```

For comparison, we also apply PCA:

In[41]:

```
pca = PCA(n_components=3)
H = pca.fit_transform(X)
```

Figure 3-19 shows the signal activity that was discovered by NMF and PCA:

In[42]:

```
models = [X, S, S_, H]
names = ['Observations (first three measurements)',
         'True sources',
         'NMF recovered signals',
         'PCA recovered signals']

fig, axes = plt.subplots(4, figsize=(8, 4), gridspec_kw={'hspace': .5},
                       subplot_kw={'xticks': (), 'yticks': ()})

for model, name, ax in zip(models, names, axes):
    ax.set_title(name)
    ax.plot(model[:, :3], '-')
```

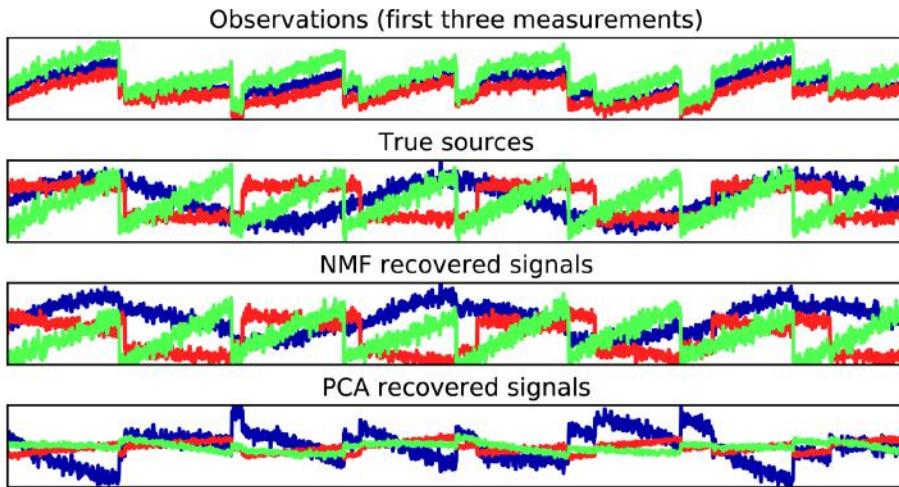


Figure 3-19. Recovering mixed sources using NMF and PCA

The figure includes 3 of the 100 measurements from X for reference. As you can see, NMF did a reasonable job of discovering the original sources, while PCA failed and used the first component to explain the majority of the variation in the data. Keep in mind that the components produced by NMF have no natural ordering. In this example, the ordering of the NMF components is the same as in the original signal (see the shading of the three curves), but this is purely accidental.

There are many other algorithms that can be used to decompose each data point into a weighted sum of a fixed set of components, as PCA and NMF do. Discussing all of them is beyond the scope of this book, and describing the constraints made on the components and coefficients often involves probability theory. If you are interested in this kind of pattern extraction, we recommend that you study the sections of the `scikit_learn` user guide on independent component analysis (ICA), factor analysis (FA), and sparse coding (dictionary learning), all of which you can find on the page about [decomposition methods](#).

Manifold Learning with t-SNE

While PCA is often a good first approach for transforming your data so that you might be able to visualize it using a scatter plot, the nature of the method (applying a rotation and then dropping directions) limits its usefulness, as we saw with the scatter plot of the Labeled Faces in the Wild dataset. There is a class of algorithms for visualization called *manifold learning algorithms* that allow for much more complex mappings, and often provide better visualizations. A particularly useful one is the t-SNE algorithm.

Manifold learning algorithms are mainly aimed at visualization, and so are rarely used to generate more than two new features. Some of them, including t-SNE, compute a new representation of the training data, but don't allow transformations of new data. This means these algorithms cannot be applied to a test set: rather, they can only transform the data they were trained for. Manifold learning can be useful for exploratory data analysis, but is rarely used if the final goal is supervised learning. The idea behind t-SNE is to find a two-dimensional representation of the data that preserves the distances between points as best as possible. t-SNE starts with a random two-dimensional representation for each data point, and then tries to make points that are close in the original feature space closer, and points that are far apart in the original feature space farther apart. t-SNE puts more emphasis on points that are close by, rather than preserving distances between far-apart points. In other words, it tries to preserve the information indicating which points are neighbors to each other.

We will apply the t-SNE manifold learning algorithm on a dataset of handwritten digits that is included in `scikit-learn`.² Each data point in this dataset is an 8×8 gray-scale image of a handwritten digit between 0 and 1. Figure 3-20 shows an example image for each class:

In[43]:

```
from sklearn.datasets import load_digits
digits = load_digits()

fig, axes = plt.subplots(2, 5, figsize=(10, 5),
                       subplot_kw={'xticks':(), 'yticks': ()})
for ax, img in zip(axes.ravel(), digits.images):
    ax.imshow(img)
```

² Not to be confused with the much larger MNIST dataset.

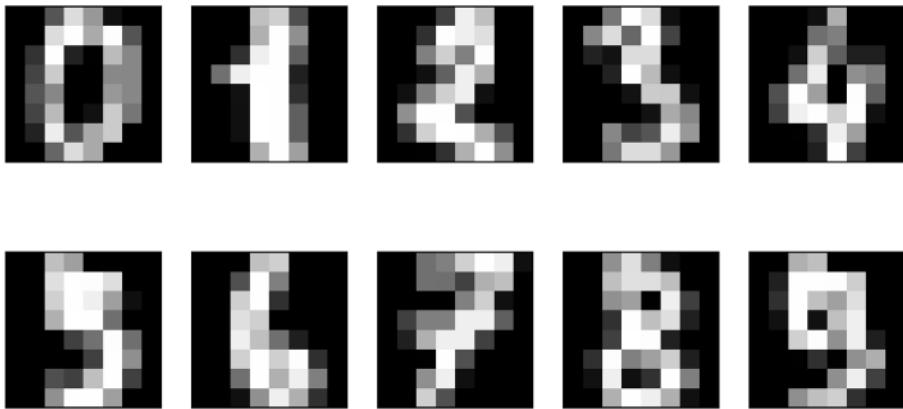


Figure 3-20. Example images from the digits dataset

Let's use PCA to visualize the data reduced to two dimensions. We plot the first two principal components, and color each dot by its class (see [Figure 3-21](#)):

In[44]:

```
# build a PCA model
pca = PCA(n_components=2)
pca.fit(digits.data)
# transform the digits data onto the first two principal components
digits_pca = pca.transform(digits.data)
colors = ["#476A2A", "#7851B8", "#BD3430", "#4A2D4E", "#875525",
          "#A83683", "#4E655E", "#853541", "#3A3120", "#535D8E"]
plt.figure(figsize=(10, 10))
plt.xlim(digits_pca[:, 0].min(), digits_pca[:, 0].max())
plt.ylim(digits_pca[:, 1].min(), digits_pca[:, 1].max())
for i in range(len(digits.data)):
    # actually plot the digits as text instead of using scatter
    plt.text(digits_pca[i, 0], digits_pca[i, 1], str(digits.target[i]),
              color = colors[digits.target[i]],
              fontdict={'weight': 'bold', 'size': 9})
plt.xlabel("First principal component")
plt.ylabel("Second principal component")
```

Here, we actually used the true digit classes as glyphs, to show which class is where. The digits zero, six, and four are relatively well separated using the first two principal components, though they still overlap. Most of the other digits overlap significantly.

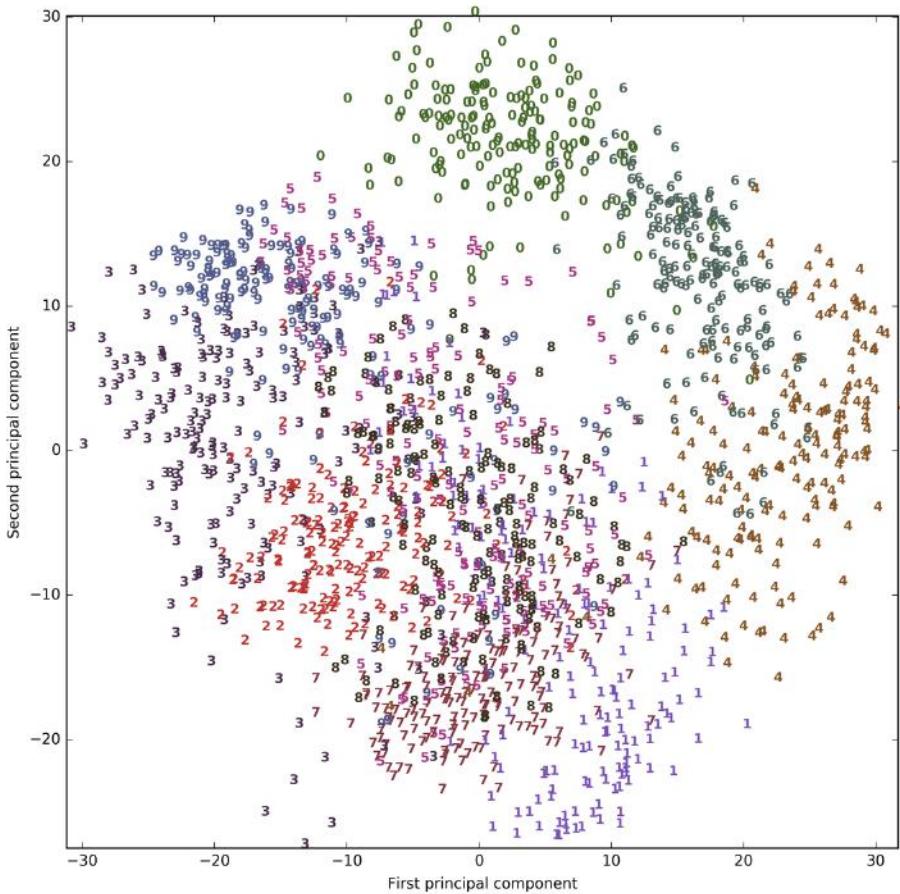


Figure 3-21. Scatter plot of the digits dataset using the first two principal components

Let's apply t-SNE to the same dataset, and compare the results. As t-SNE does not support transforming new data, the TSNE class has no `transform` method. Instead, we can call the `fit_transform` method, which will build the model and immediately return the transformed data (see Figure 3-22):

In[45]:

```
from sklearn.manifold import TSNE
tsne = TSNE(random_state=42)
# use fit_transform instead of fit, as TSNE has no transform method
digits_tsne = tsne.fit_transform(digits.data)
```

In[46]:

```
plt.figure(figsize=(10, 10))
plt.xlim(digits_tsne[:, 0].min(), digits_tsne[:, 0].max() + 1)
plt.ylim(digits_tsne[:, 1].min(), digits_tsne[:, 1].max() + 1)
for i in range(len(digits.data)):
    # actually plot the digits as text instead of using scatter
    plt.text(digits_tsne[i, 0], digits_tsne[i, 1], str(digits.target[i]),
              color = colors[digits.target[i]],
              fontdict={ 'weight': 'bold', 'size': 9})
plt.xlabel("t-SNE feature 0")
plt.xlabel("t-SNE feature 1")
```

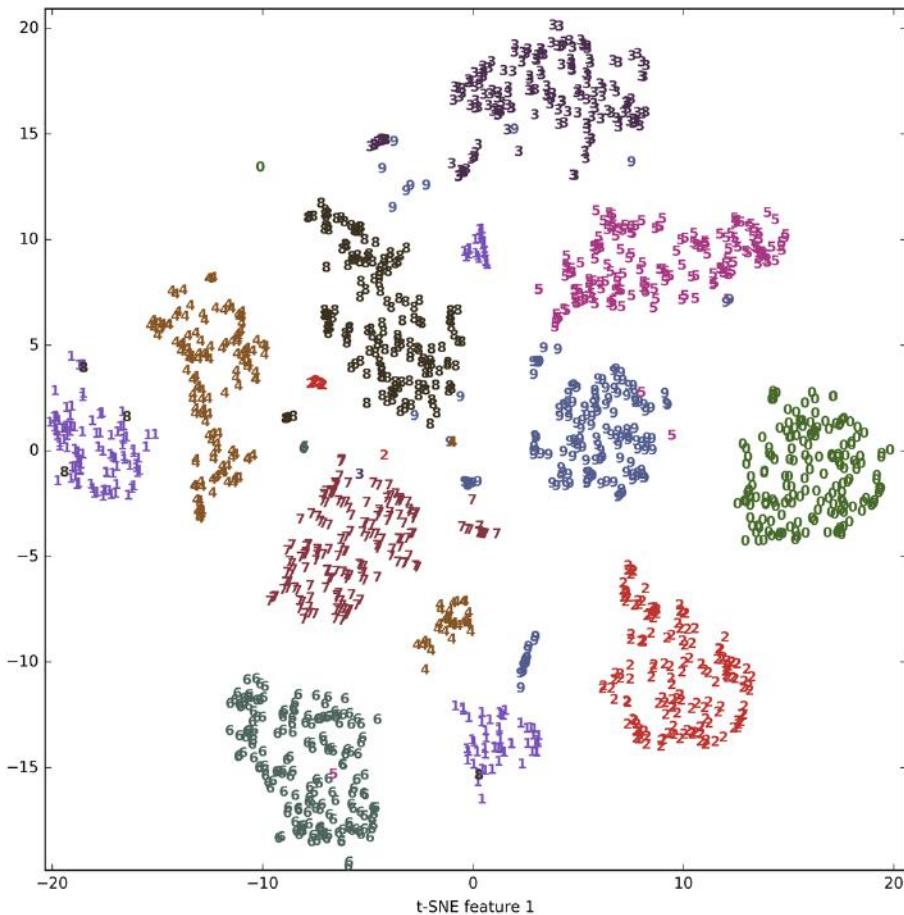


Figure 3-22. Scatter plot of the digits dataset using two components found by t-SNE

The result of t-SNE is quite remarkable. All the classes are quite clearly separated. The ones and nines are somewhat split up, but most of the classes form a single dense group. Keep in mind that this method has no knowledge of the class labels: it is completely unsupervised. Still, it can find a representation of the data in two dimensions that clearly separates the classes, based solely on how close points are in the original space.

The t-SNE algorithm has some tuning parameters, though it often works well with the default settings. You can try playing with `perplexity` and `early_exaggeration`, but the effects are usually minor.

Clustering

As we described earlier, *clustering* is the task of partitioning the dataset into groups, called clusters. The goal is to split up the data in such a way that points within a single cluster are very similar and points in different clusters are different. Similarly to classification algorithms, clustering algorithms assign (or predict) a number to each data point, indicating which cluster a particular point belongs to.

k-Means Clustering

k-means clustering is one of the simplest and most commonly used clustering algorithms. It tries to find *cluster centers* that are representative of certain regions of the data. The algorithm alternates between two steps: assigning each data point to the closest cluster center, and then setting each cluster center as the mean of the data points that are assigned to it. The algorithm is finished when the assignment of instances to clusters no longer changes. The following example ([Figure 3-23](#)) illustrates the algorithm on a synthetic dataset:

In[47]:

```
mglearn.plots.plot_kmeans_algorithm()
```

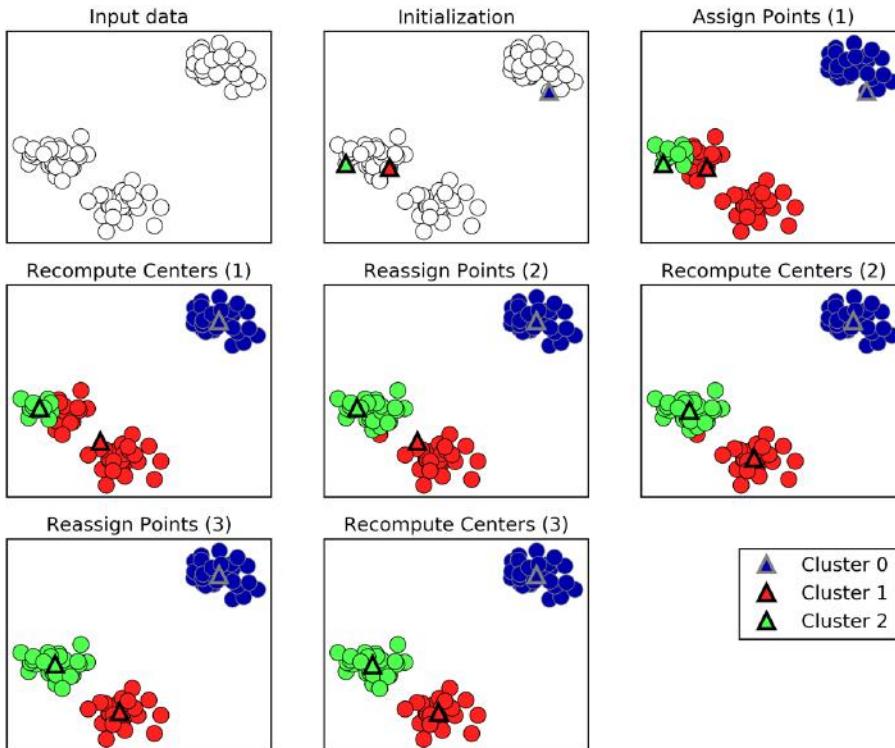


Figure 3-23. Input data and three steps of the k-means algorithm

Cluster centers are shown as triangles, while data points are shown as circles. Colors indicate cluster membership. We specified that we are looking for three clusters, so the algorithm was initialized by declaring three data points randomly as cluster centers (see “Initialization”). Then the iterative algorithm starts. First, each data point is assigned to the cluster center it is closest to (see “Assign Points (1)”). Next, the cluster centers are updated to be the mean of the assigned points (see “Recompute Centers (1)”). Then the process is repeated two more times. After the third iteration, the assignment of points to cluster centers remained unchanged, so the algorithm stops.

Given new data points, k-means will assign each to the closest cluster center. The next example ([Figure 3-24](#)) shows the boundaries of the cluster centers that were learned in [Figure 3-23](#):

In[48]:

```
mglearn.plots.plot_kmeans_boundaries()
```

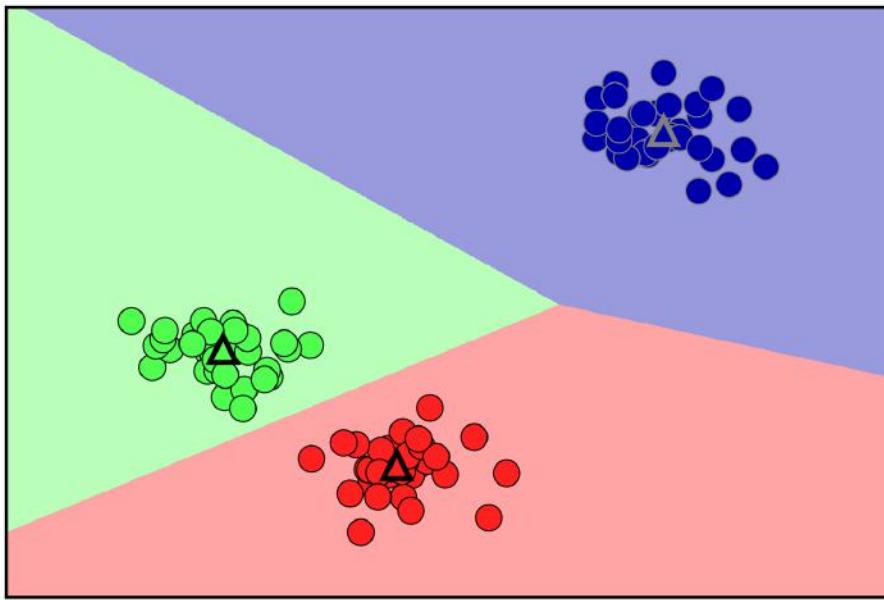


Figure 3-24. Cluster centers and cluster boundaries found by the k-means algorithm

Applying k -means with `scikit-learn` is quite straightforward. Here, we apply it to the synthetic data that we used for the preceding plots. We instantiate the `KMeans` class, and set the number of clusters we are looking for.³ Then we call the `fit` method with the data:

In[49]:

```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# generate synthetic two-dimensional data
X, y = make_blobs(random_state=1)

# build the clustering model
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
```

During the algorithm, each training data point in `X` is assigned a cluster label. You can find these labels in the `kmeans.labels_` attribute:

³ If you don't provide `n_clusters`, it is set to 8 by default. There is no particular reason why you should use this value.

In[50]:

```
print("Cluster memberships:\n{}".format(kmeans.labels_))
```

Out[50]:

```
Cluster memberships:  
[1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2  
 2 2 0 0 2 1 2 2 0 1 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1  
 1 2 0 0 1 2 1 2 2 0 1 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1]
```

As we asked for three clusters, the clusters are numbered 0 to 2.

You can also assign cluster labels to new points, using the `predict` method. Each new point is assigned to the closest cluster center when predicting, but the existing model is not changed. Running `predict` on the training set returns the same result as `labels_`:

In[51]:

```
print(kmeans.predict(X))
```

Out[51]:

```
[1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2  
 2 2 0 0 2 1 2 2 0 1 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1  
 1 2 0 0 1 2 1 2 2 0 1 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1]
```

You can see that clustering is somewhat similar to classification, in that each item gets a label. However, there is no ground truth, and consequently the labels themselves have no *a priori* meaning. Let's go back to the example of clustering face images that we discussed before. It might be that the cluster 3 found by the algorithm contains only faces of your friend Bela. You can only know that after you look at the pictures, though, and the number 3 is arbitrary. The only information the algorithm gives you is that all faces labeled as 3 are similar.

For the clustering we just computed on the two-dimensional toy dataset, that means that we should not assign any significance to the fact that one group was labeled 0 and another one was labeled 1. Running the algorithm again might result in a different numbering of clusters because of the random nature of the initialization.

Here is a plot of this data again (Figure 3-25). The cluster centers are stored in the `cluster_centers_` attribute, and we plot them as triangles:

In[52]:

```
mglearn.discrete_scatter(X[:, 0], X[:, 1], kmeans.labels_, markers='o')  
mglearn.discrete_scatter(  
    kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], [0, 1, 2],  
    markers='^', markeredgeweighth=2)
```

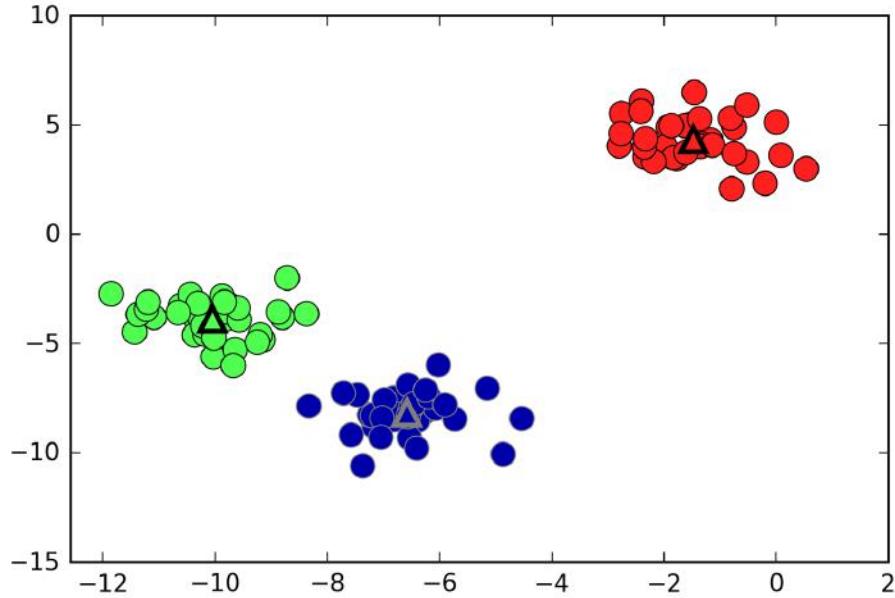


Figure 3-25. Cluster assignments and cluster centers found by k-means with three clusters

We can also use more or fewer cluster centers (Figure 3-26):

In[53]:

```
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

# using two cluster centers:
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
assignments = kmeans.labels_

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[0])

# using five cluster centers:
kmeans = KMeans(n_clusters=5)
kmeans.fit(X)
assignments = kmeans.labels_

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignments, ax=axes[1])
```

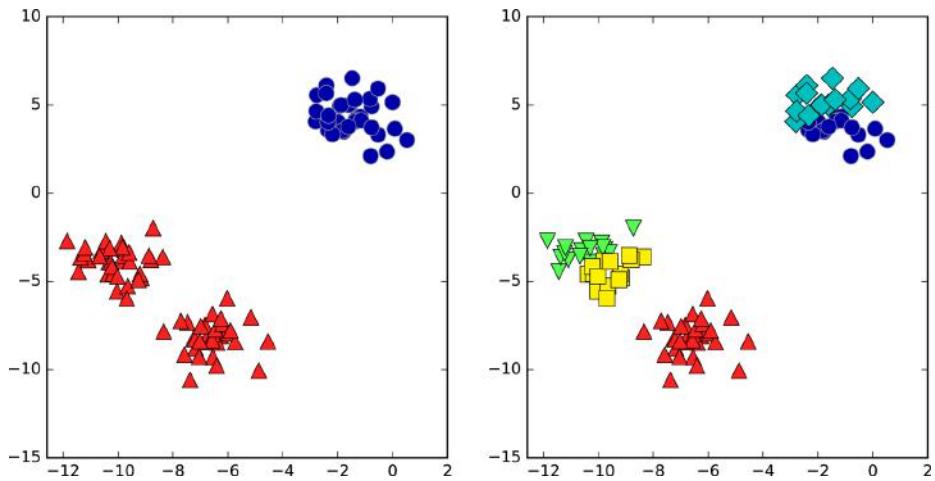


Figure 3-26. Cluster assignments found by k-means using two clusters (left) and five clusters (right)

Failure cases of k-means

Even if you know the “right” number of clusters for a given dataset, k-means might not always be able to recover them. Each cluster is defined solely by its center, which means that each cluster is a convex shape. As a result of this, k-means can only capture relatively simple shapes. k-means also assumes that all clusters have the same “diameter” in some sense; it always draws the boundary between clusters to be exactly in the middle between the cluster centers. That can sometimes lead to surprising results, as shown in [Figure 3-27](#):

In[54]:

```
X_varied, y_varied = make_blobs(n_samples=200,
                                 cluster_std=[1.0, 2.5, 0.5],
                                 random_state=170)
y_pred = KMeans(n_clusters=3, random_state=0).fit_predict(X_varied)

mglearn.discrete_scatter(X_varied[:, 0], X_varied[:, 1], y_pred)
plt.legend(["cluster 0", "cluster 1", "cluster 2"], loc='best')
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

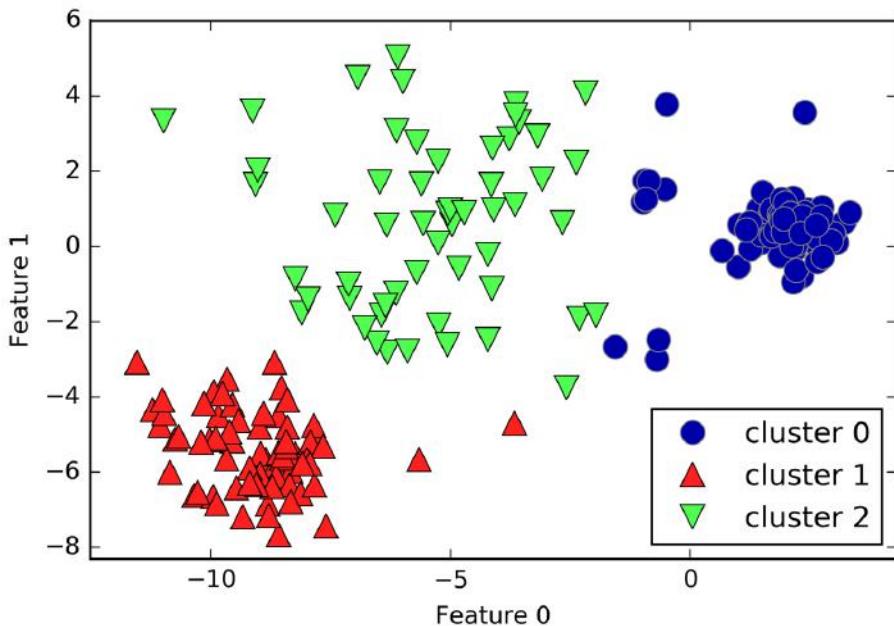


Figure 3-27. Cluster assignments found by k-means when clusters have different densities

One might have expected the dense region in the lower left to be the first cluster, the dense region in the upper right to be the second, and the less dense region in the center to be the third. Instead, both cluster 0 and cluster 1 have some points that are far away from all the other points in these clusters that “reach” toward the center.

k-means also assumes that all directions are equally important for each cluster. The following plot (Figure 3-28) shows a two-dimensional dataset where there are three clearly separated parts in the data. However, these groups are stretched toward the diagonal. As k-means only considers the distance to the nearest cluster center, it can’t handle this kind of data:

In[55]:

```
# generate some random cluster data
X, y = make_blobs(random_state=170, n_samples=600)
rng = np.random.RandomState(74)

# transform the data to be stretched
transformation = rng.normal(size=(2, 2))
X = np.dot(X, transformation)
```

```

# cluster the data into three clusters
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
y_pred = kmeans.predict(X)

# plot the cluster assignments and cluster centers
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm3)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            marker='^', c=[0, 1, 2], s=100, linewidth=2, cmap=mglearn.cm3)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")

```

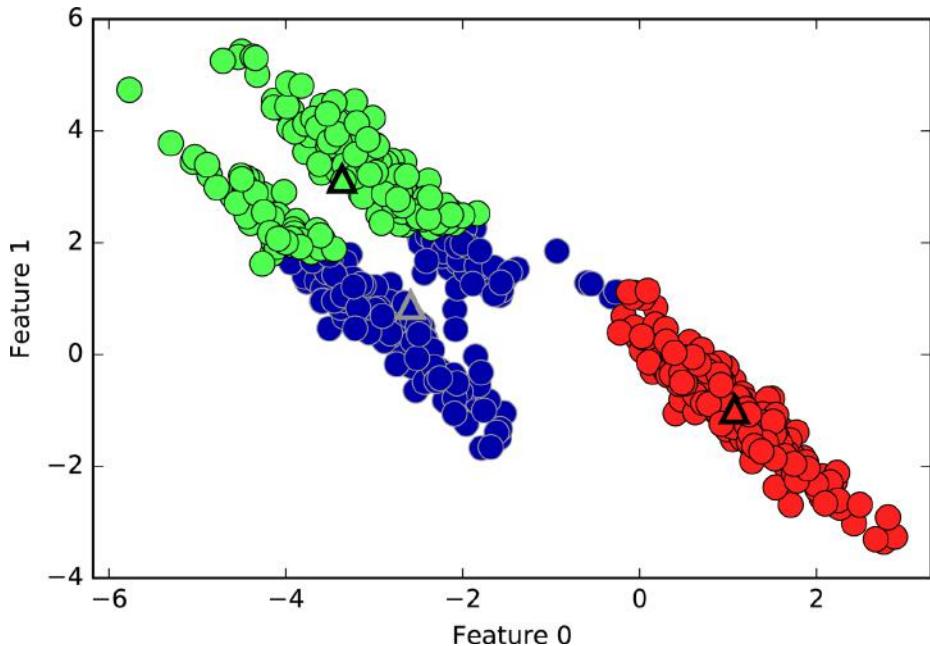


Figure 3-28. k-means fails to identify nonspherical clusters

k-means also performs poorly if the clusters have more complex shapes, like the two_moons data we encountered in [Chapter 2](#) (see [Figure 3-29](#)):

In[56]:

```

# generate synthetic two_moons data (with less noise this time)
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# cluster the data into two clusters
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
y_pred = kmeans.predict(X)

```

```

# plot the cluster assignments and cluster centers
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap=mglearn.cm2, s=60)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            marker='^', c=[mglearn.cm2(0), mglearn.cm2(1)], s=100, linewidth=2)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")

```

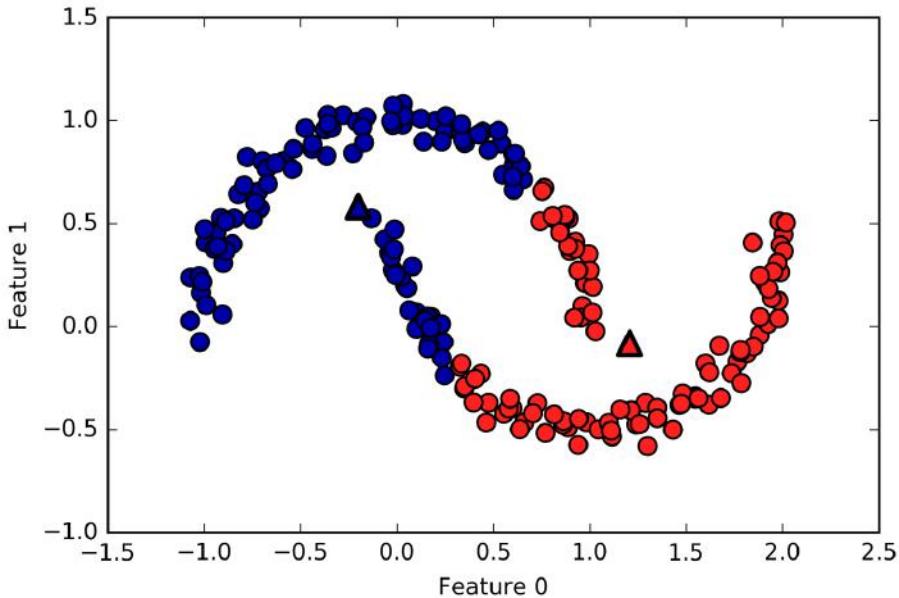


Figure 3-29. k-means fails to identify clusters with complex shapes

Here, we would hope that the clustering algorithm can discover the two half-moon shapes. However, this is not possible using the *k*-means algorithm.

Vector quantization, or seeing k-means as decomposition

Even though *k*-means is a clustering algorithm, there are interesting parallels between *k*-means and the decomposition methods like PCA and NMF that we discussed earlier. You might remember that PCA tries to find directions of maximum variance in the data, while NMF tries to find additive components, which often correspond to “extremes” or “parts” of the data (see [Figure 3-13](#)). Both methods tried to express the data points as a sum over some components. *k*-means, on the other hand, tries to represent each data point using a cluster center. You can think of that as each point being represented using only a single component, which is given by the cluster center. This view of *k*-means as a decomposition method, where each point is represented using a single component, is called *vector quantization*.

Let's do a side-by-side comparison of PCA, NMF, and k -means, showing the components extracted (Figure 3-30), as well as reconstructions of faces from the test set using 100 components (Figure 3-31). For k -means, the reconstruction is the closest cluster center found on the training set:

In[57]:

```
X_train, X_test, y_train, y_test = train_test_split(  
    X_people, y_people, stratify=y_people, random_state=0)  
nmf = NMF(n_components=100, random_state=0)  
nmf.fit(X_train)  
pca = PCA(n_components=100, random_state=0)  
pca.fit(X_train)  
kmeans = KMeans(n_clusters=100, random_state=0)  
kmeans.fit(X_train)  
  
X_reconstructed_pca = pca.inverse_transform(pca.transform(X_test))  
X_reconstructed_kmeans = kmeans.cluster_centers_[kmeans.predict(X_test)]  
X_reconstructed_nmf = np.dot(nmf.transform(X_test), nmf.components_)
```

In[58]:

```
fig, axes = plt.subplots(3, 5, figsize=(8, 8),  
                      subplot_kw={'xticks': (), 'yticks': ()})  
fig.suptitle("Extracted Components")  
for ax, comp_kmeans, comp_pca, comp_nmf in zip(  
    axes.T, kmeans.cluster_centers_, pca.components_, nmf.components_):  
    ax[0].imshow(comp_kmeans.reshape(image_shape))  
    ax[1].imshow(comp_pca.reshape(image_shape), cmap='viridis')  
    ax[2].imshow(comp_nmf.reshape(image_shape))  
  
axes[0, 0].set_ylabel("kmeans")  
axes[1, 0].set_ylabel("pca")  
axes[2, 0].set_ylabel("nmf")  
  
fig, axes = plt.subplots(4, 5, subplot_kw={'xticks': (), 'yticks': ()},  
                      figsize=(8, 8))  
fig.suptitle("Reconstructions")  
for ax, orig, rec_kmeans, rec_pca, rec_nmf in zip(  
    axes.T, X_test, X_reconstructed_kmeans, X_reconstructed_pca,  
    X_reconstructed_nmf):  
  
    ax[0].imshow(orig.reshape(image_shape))  
    ax[1].imshow(rec_kmeans.reshape(image_shape))  
    ax[2].imshow(rec_pca.reshape(image_shape))  
    ax[3].imshow(rec_nmf.reshape(image_shape))  
  
axes[0, 0].set_ylabel("original")  
axes[1, 0].set_ylabel("kmeans")  
axes[2, 0].set_ylabel("pca")  
axes[3, 0].set_ylabel("nmf")
```

Extracted Components

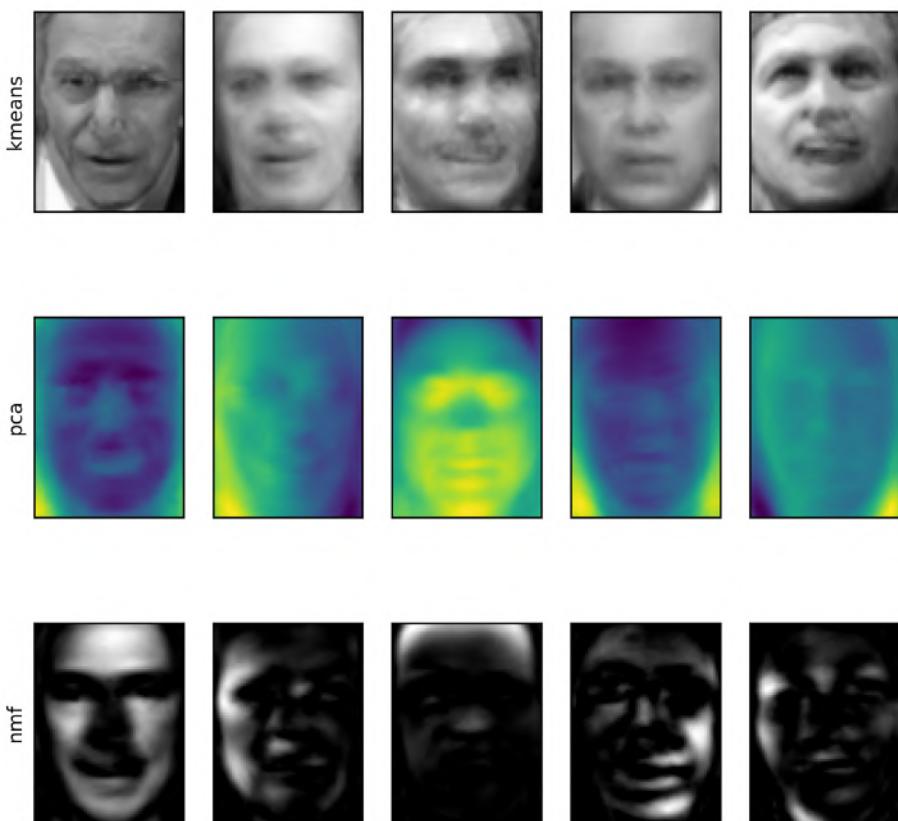


Figure 3-30. Comparing k -means cluster centers to components found by PCA and NMF



Figure 3-31. Comparing image reconstructions using k-means, PCA, and NMF with 100 components (or cluster centers)—k-means uses only a single cluster center per image

An interesting aspect of vector quantization using k -means is that we can use many more clusters than input dimensions to encode our data. Let's go back to the `two_moons` data. Using PCA or NMF, there is nothing much we can do to this data, as it lives in only two dimensions. Reducing it to one dimension with PCA or NMF would completely destroy the structure of the data. But we can find a more expressive representation with k -means, by using more cluster centers (see [Figure 3-32](#)):

In[59]:

```
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

kmeans = KMeans(n_clusters=10, random_state=0)
kmeans.fit(X)
y_pred = kmeans.predict(X)

plt.scatter(X[:, 0], X[:, 1], c=y_pred, s=60, cmap='Paired')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=60,
            marker='^', c=range(kmeans.n_clusters), linewidth=2, cmap='Paired')
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
print("Cluster memberships:\n{}".format(y_pred))
```

Out[59]:

```
Cluster memberships:
[9 2 5 4 2 7 9 6 9 6 1 0 2 6 1 9 3 0 3 1 7 6 8 6 8 5 2 7 5 8 9 8 6 5 3 7 0
 9 4 5 0 1 3 5 2 8 9 1 5 6 1 0 7 4 6 3 3 6 3 8 0 4 2 9 6 4 8 2 8 4 0 4 0 5
 6 4 5 9 3 0 7 8 0 7 5 8 9 8 0 7 3 9 7 1 7 2 2 0 4 5 6 7 8 9 4 5 4 1 2 3 1
 8 8 4 9 2 3 7 0 9 9 1 5 8 5 1 9 5 6 7 9 1 4 0 6 2 6 4 7 9 5 5 3 8 1 9 5 6
 3 5 0 2 9 3 0 8 6 0 3 3 5 6 3 2 0 2 3 0 2 6 3 4 4 1 5 6 7 1 1 3 2 4 7 2 7
 3 8 6 4 1 4 3 9 9 5 1 7 5 8 2]
```

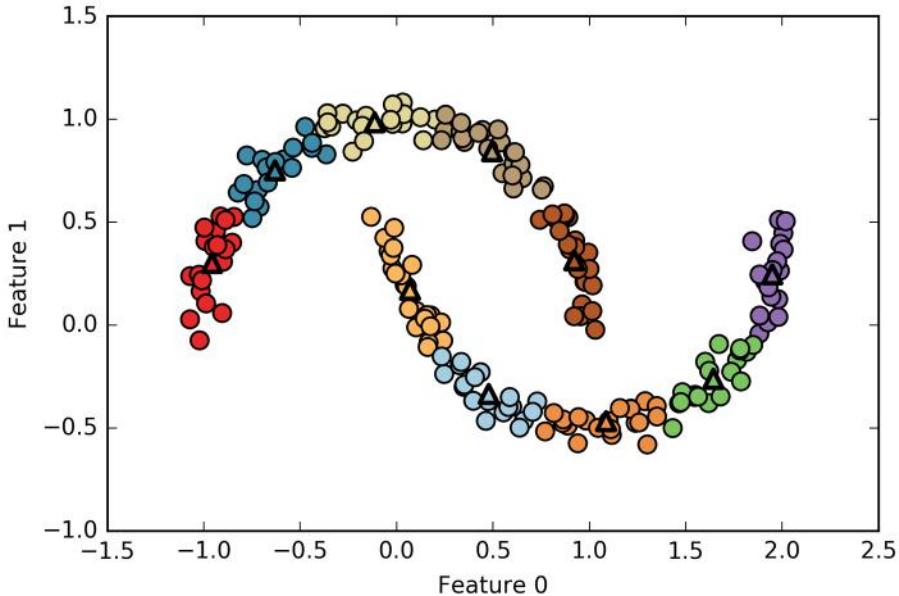


Figure 3-32. Using many k-means clusters to cover the variation in a complex dataset

We used 10 cluster centers, which means each point is now assigned a number between 0 and 9. We can see this as the data being represented using 10 components (that is, we have 10 new features), with all features being 0, apart from the one that represents the cluster center the point is assigned to. Using this 10-dimensional representation, it would now be possible to separate the two half-moon shapes using a linear model, which would not have been possible using the original two features. It is also possible to get an even more expressive representation of the data by using the distances to each of the cluster centers as features. This can be accomplished using the `transform` method of `kmeans`:

In[60]:

```
distance_features = kmeans.transform(X)
print("Distance feature shape: {}".format(distance_features.shape))
print("Distance features:\n{}".format(distance_features))
```

Out[60]:

```
Distance feature shape: (200, 10)
Distance features:
[[ 0.922  1.466  1.14 ...,  1.166  1.039  0.233]
 [ 1.142  2.517  0.12 ...,  0.707  2.204  0.983]
 [ 0.788  0.774  1.749 ...,  1.971  0.716  0.944]
 ...
 [ 0.446  1.106  1.49 ...,  1.791  1.032  0.812]
 [ 1.39   0.798  1.981 ...,  1.978  0.239  1.058]
 [ 1.149  2.454  0.045 ...,  0.572  2.113  0.882]]
```

k-means is a very popular algorithm for clustering, not only because it is relatively easy to understand and implement, but also because it runs relatively quickly. *k*-means scales easily to large datasets, and `scikit-learn` even includes a more scalable variant in the `MiniBatchKMeans` class, which can handle very large datasets.

One of the drawbacks of *k*-means is that it relies on a random initialization, which means the outcome of the algorithm depends on a random seed. By default, `scikit-learn` runs the algorithm 10 times with 10 different random initializations, and returns the best result.⁴ Further downsides of *k*-means are the relatively restrictive assumptions made on the shape of clusters, and the requirement to specify the number of clusters you are looking for (which might not be known in a real-world application).

Next, we will look at two more clustering algorithms that improve upon these properties in some ways.

⁴ In this case, “best” means that the sum of variances of the clusters is small.

Agglomerative Clustering

Agglomerative clustering refers to a collection of clustering algorithms that all build upon the same principles: the algorithm starts by declaring each point its own cluster, and then merges the two most similar clusters until some stopping criterion is satisfied. The stopping criterion implemented in `scikit-learn` is the number of clusters, so similar clusters are merged until only the specified number of clusters are left. There are several *linkage* criteria that specify how exactly the “most similar cluster” is measured. This measure is always defined between two existing clusters.

The following three choices are implemented in `scikit-learn`:

`ward`

The default choice, `ward` picks the two clusters to merge such that the variance within all clusters increases the least. This often leads to clusters that are relatively equally sized.

`average`

`average` linkage merges the two clusters that have the smallest average distance between all their points.

`complete`

`complete` linkage (also known as maximum linkage) merges the two clusters that have the smallest maximum distance between their points.

`ward` works on most datasets, and we will use it in our examples. If the clusters have very dissimilar numbers of members (if one is much bigger than all the others, for example), `average` or `complete` might work better.

The following plot (Figure 3-33) illustrates the progression of agglomerative clustering on a two-dimensional dataset, looking for three clusters:

In[61]:

```
mglearn.plots.plot_agglomerative_algorithm()
```

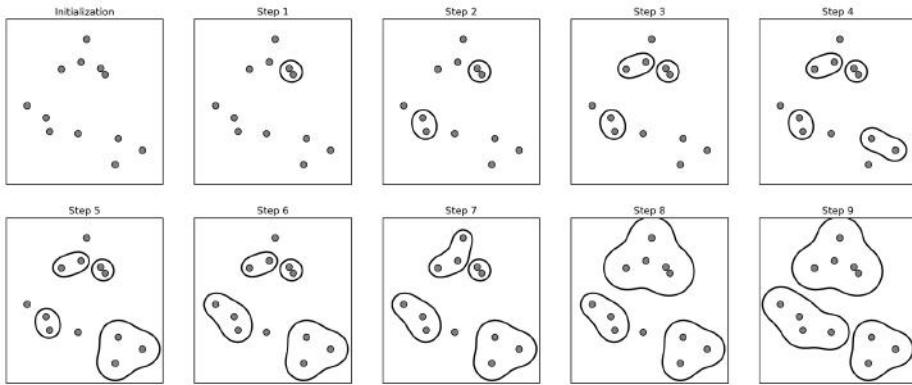


Figure 3-33. Agglomerative clustering iteratively joins the two closest clusters

Initially, each point is its own cluster. Then, in each step, the two clusters that are closest are merged. In the first four steps, two single-point clusters are picked and these are joined into two-point clusters. In step 5, one of the two-point clusters is extended to a third point, and so on. In step 9, there are only three clusters remaining. As we specified that we are looking for three clusters, the algorithm then stops.

Let's have a look at how agglomerative clustering performs on the simple three-cluster data we used here. Because of the way the algorithm works, agglomerative clustering cannot make predictions for new data points. Therefore, Agglomerative Clustering has no `predict` method. To build the model and get the cluster memberships on the training set, use the `fit_predict` method instead.⁵ The result is shown in Figure 3-34:

In[62]:

```
from sklearn.cluster import AgglomerativeClustering
X, y = make_blobs(random_state=1)

agg = AgglomerativeClustering(n_clusters=3)
assignment = agg.fit_predict(X)

mglearn.discrete_scatter(X[:, 0], X[:, 1], assignment)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

⁵ We could also use the `labels_` attribute, as we did for k -means.

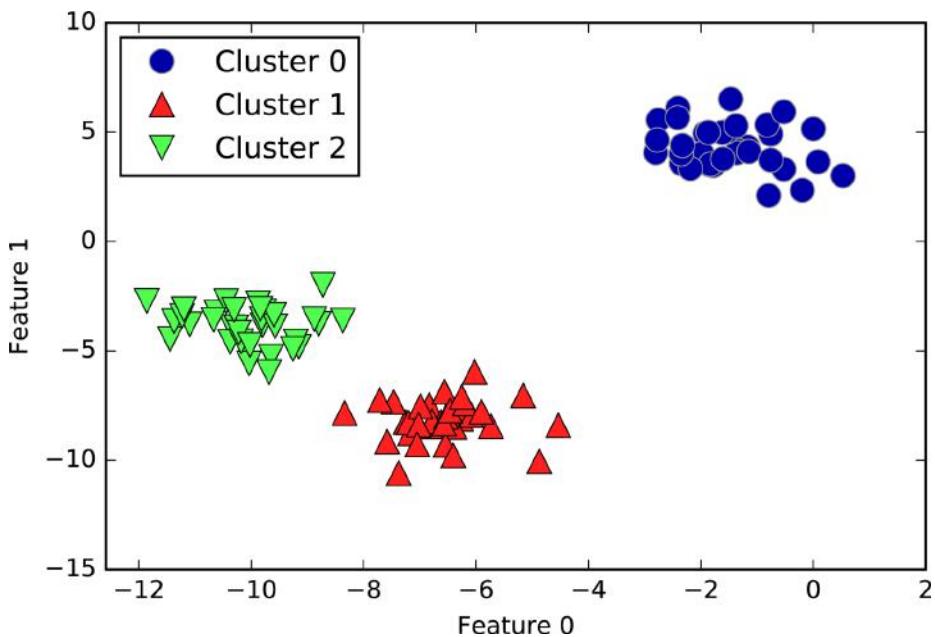


Figure 3-34. Cluster assignment using agglomerative clustering with three clusters

As expected, the algorithm recovers the clustering perfectly. While the scikit-learn implementation of agglomerative clustering requires you to specify the number of clusters you want the algorithm to find, agglomerative clustering methods provide some help with choosing the right number, which we will discuss next.

Hierarchical clustering and dendograms

Agglomerative clustering produces what is known as a *hierarchical clustering*. The clustering proceeds iteratively, and every point makes a journey from being a single point cluster to belonging to some final cluster. Each intermediate step provides a clustering of the data (with a different number of clusters). It is sometimes helpful to look at all possible clusterings jointly. The next example (Figure 3-35) shows an overlay of all the possible clusterings shown in Figure 3-33, providing some insight into how each cluster breaks up into smaller clusters:

In[63]:

```
mlearn.plots.plot_agglomerative()
```

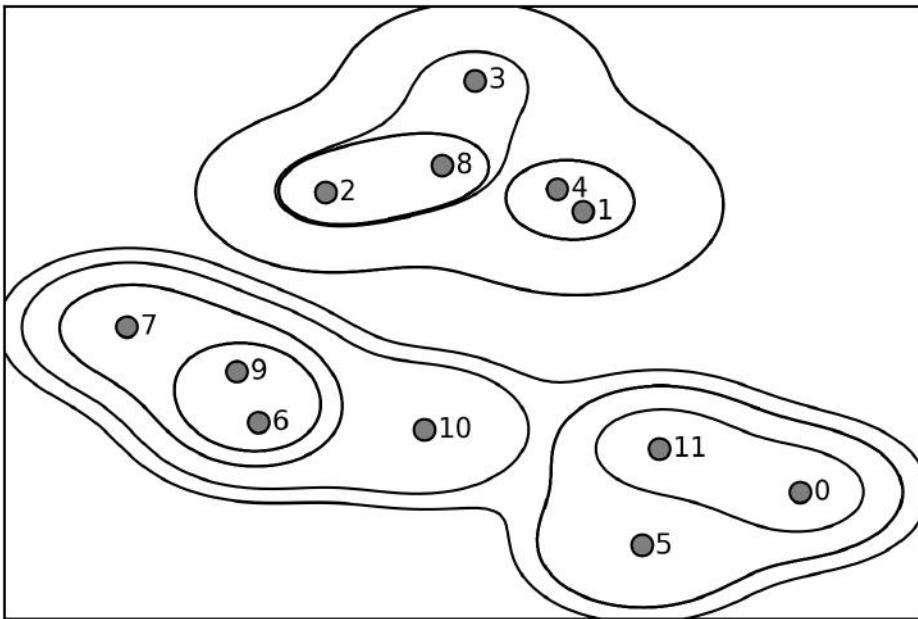


Figure 3-35. Hierarchical cluster assignment (shown as lines) generated with agglomerative clustering, with numbered data points (cf. Figure 3-36)

While this visualization provides a very detailed view of the hierarchical clustering, it relies on the two-dimensional nature of the data and therefore cannot be used on datasets that have more than two features. There is, however, another tool to visualize hierarchical clustering, called a *dendrogram*, that can handle multidimensional datasets.

Unfortunately, `scikit-learn` currently does not have the functionality to draw dendograms. However, you can generate them easily using SciPy. The SciPy clustering algorithms have a slightly different interface to the `scikit-learn` clustering algorithms. SciPy provides a function that takes a data array `X` and computes a *linkage array*, which encodes hierarchical cluster similarities. We can then feed this linkage array into the `scipy.dendrogram` function to plot the dendrogram (Figure 3-36):

In[64]:

```
# Import the dendrogram function and the ward clustering function from SciPy
from scipy.cluster.hierarchy import dendrogram, ward

X, y = make_blobs(random_state=0, n_samples=12)
# Apply the ward clustering to the data array X
# The SciPy ward function returns an array that specifies the distances
# bridged when performing agglomerative clustering
linkage_array = ward(X)
```

```

# Now we plot the dendrogram for the linkage_array containing the distances
# between clusters
dendrogram(linkage_array)

# Mark the cuts in the tree that signify two or three clusters
ax = plt.gca()
bounds = ax.get_xbound()
ax.plot(bounds, [7.25, 7.25], '--', c='k')
ax.plot(bounds, [4, 4], '--', c='k')

ax.text(bounds[1], 7.25, ' two clusters', va='center', fontdict={'size': 15})
ax.text(bounds[1], 4, ' three clusters', va='center', fontdict={'size': 15})
plt.xlabel("Sample index")
plt.ylabel("Cluster distance")

```

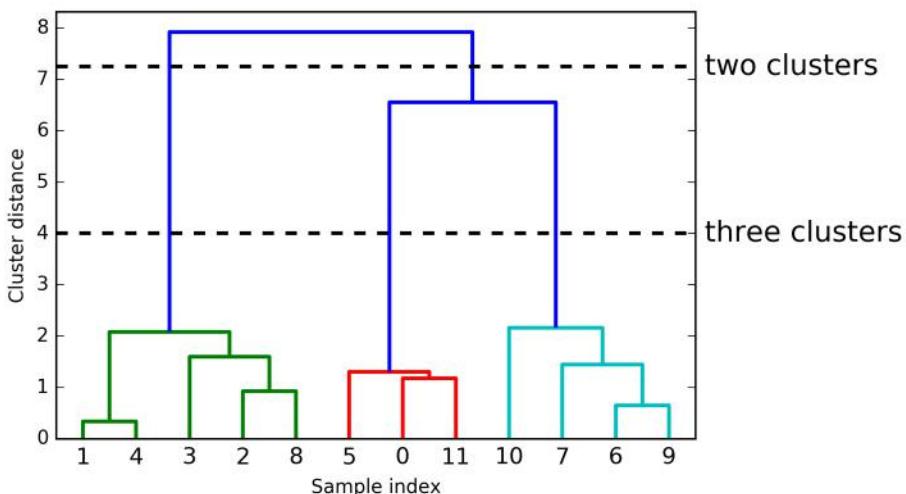


Figure 3-36. Dendrogram of the clustering shown in Figure 3-35 with lines indicating splits into two and three clusters

The dendrogram shows data points as points on the bottom (numbered from 0 to 11). Then, a tree is plotted with these points (representing single-point clusters) as the leaves, and a new node parent is added for each two clusters that are joined.

Reading from bottom to top, the data points 1 and 4 are joined first (as you could see in Figure 3-33). Next, points 6 and 9 are joined into a cluster, and so on. At the top level, there are two branches, one consisting of points 11, 0, 5, 10, 7, 6, and 9, and the other consisting of points 1, 4, 3, 2, and 8. These correspond to the two largest clusters in the lefthand side of the plot.

The y-axis in the dendrogram doesn't just specify when in the agglomerative algorithm two clusters get merged. The length of each branch also shows how far apart the merged clusters are. The longest branches in this dendrogram are the three lines that are marked by the dashed line labeled "three clusters." That these are the longest branches indicates that going from three to two clusters meant merging some very far-apart points. We see this again at the top of the chart, where merging the two remaining clusters into a single cluster again bridges a relatively large distance.

Unfortunately, agglomerative clustering still fails at separating complex shapes like the `two_moons` dataset. But the same is not true for the next algorithm we will look at, DBSCAN.

DBSCAN

Another very useful clustering algorithm is DBSCAN (which stands for "density-based spatial clustering of applications with noise"). The main benefits of DBSCAN are that it does not require the user to set the number of clusters *a priori*, it can capture clusters of complex shapes, and it can identify points that are not part of any cluster. DBSCAN is somewhat slower than agglomerative clustering and k -means, but still scales to relatively large datasets.

DBSCAN works by identifying points that are in "crowded" regions of the feature space, where many data points are close together. These regions are referred to as *dense* regions in feature space. The idea behind DBSCAN is that clusters form dense regions of data, separated by regions that are relatively empty.

Points that are within a dense region are called *core samples* (or core points), and they are defined as follows. There are two parameters in DBSCAN: `min_samples` and `eps`. If there are at least `min_samples` many data points within a distance of `eps` to a given data point, that data point is classified as a core sample. Core samples that are closer to each other than the distance `eps` are put into the same cluster by DBSCAN.

The algorithm works by picking an arbitrary point to start with. It then finds all points with distance `eps` or less from that point. If there are less than `min_samples` points within distance `eps` of the starting point, this point is labeled as *noise*, meaning that it doesn't belong to any cluster. If there are more than `min_samples` points within a distance of `eps`, the point is labeled a core sample and assigned a new cluster label. Then, all neighbors (within `eps`) of the point are visited. If they have not been assigned a cluster yet, they are assigned the new cluster label that was just created. If they are core samples, their neighbors are visited in turn, and so on. The cluster grows until there are no more core samples within distance `eps` of the cluster. Then another point that hasn't yet been visited is picked, and the same procedure is repeated.

In the end, there are three kinds of points: core points, points that are within distance `eps` of core points (called *boundary points*), and noise. When the DBSCAN algorithm is run on a particular dataset multiple times, the clustering of the core points is always the same, and the same points will always be labeled as noise. However, a boundary point might be neighbor to core samples of more than one cluster. Therefore, the cluster membership of boundary points depends on the order in which points are visited. Usually there are only few boundary points, and this slight dependence on the order of points is not important.

Let's apply DBSCAN on the synthetic dataset we used to demonstrate agglomerative clustering. Like agglomerative clustering, DBSCAN does not allow predictions on new test data, so we will use the `fit_predict` method to perform clustering and return the cluster labels in one step:

In[65]:

```
from sklearn.cluster import DBSCAN
X, y = make_blobs(random_state=0, n_samples=12)

dbSCAN = DBSCAN()
clusters = dbSCAN.fit_predict(X)
print("Cluster memberships:\n{}".format(clusters))
```

Out[65]:

```
Cluster memberships:
[-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

As you can see, all data points were assigned the label `-1`, which stands for noise. This is a consequence of the default parameter settings for `eps` and `min_samples`, which are not tuned for small toy datasets. The cluster assignments for different values of `min_samples` and `eps` are shown below, and visualized in [Figure 3-37](#):

In[66]:

```
mglearn.plots.plot_dbSCAN()
```

Out[66]:

```
min_samples: 2 eps: 1.000000  cluster: [-1  0  0 -1  0 -1  1  1  0  1 -1 -1]
min_samples: 2 eps: 1.500000  cluster: [0 1 1 1 1 0 2 2 1 2 2 0]
min_samples: 2 eps: 2.000000  cluster: [0 1 1 1 1 0 0 0 1 0 0 0]
min_samples: 2 eps: 3.000000  cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
min_samples: 3 eps: 1.000000  cluster: [-1  0  0 -1  0 -1  1  1  0  1 -1 -1]
min_samples: 3 eps: 1.500000  cluster: [0 1 1 1 1 0 2 2 1 2 2 0]
min_samples: 3 eps: 2.000000  cluster: [0 1 1 1 1 0 0 0 1 0 0 0]
min_samples: 3 eps: 3.000000  cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
min_samples: 5 eps: 1.000000  cluster: [-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
min_samples: 5 eps: 1.500000  cluster: [-1  0  0  0  0 -1 -1 -1  0 -1 -1 -1]
min_samples: 5 eps: 2.000000  cluster: [-1  0  0  0 -1 -1 -1  0 -1 -1 -1 -1]
min_samples: 5 eps: 3.000000  cluster: [0 0 0 0 0 0 0 0 0 0 0 0]
```

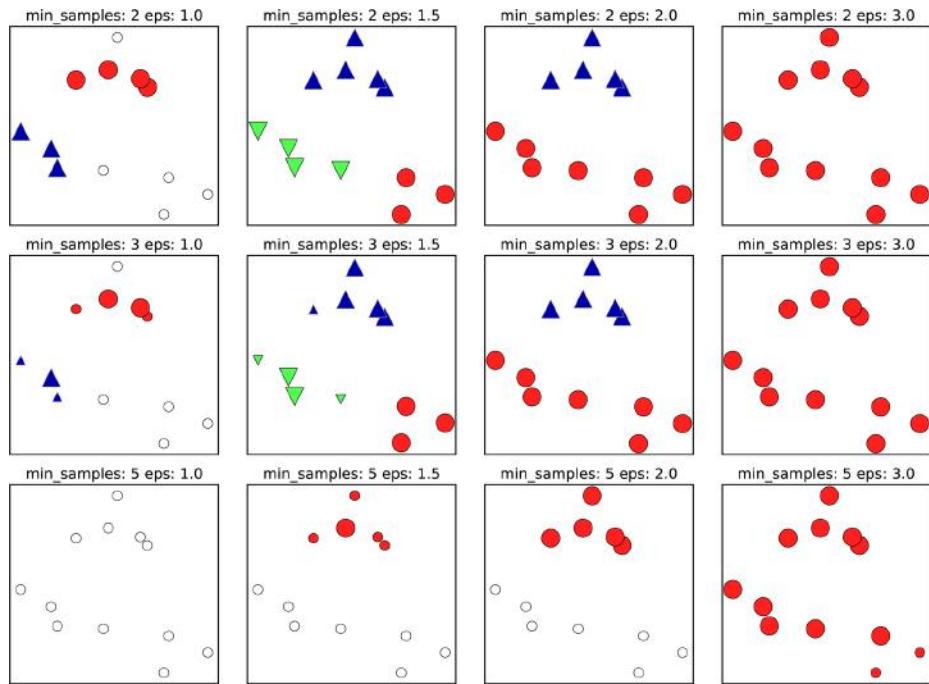


Figure 3-37. Cluster assignments found by DBSCAN with varying settings for the min_samples and eps parameters

In this plot, points that belong to clusters are solid, while the noise points are shown in white. Core samples are shown as large markers, while boundary points are displayed as smaller markers. Increasing eps (going from left to right in the figure) means that more points will be included in a cluster. This makes clusters grow, but might also lead to multiple clusters joining into one. Increasing min_samples (going from top to bottom in the figure) means that fewer points will be core points, and more points will be labeled as noise.

The parameter eps is somewhat more important, as it determines what it means for points to be “close.” Setting eps to be very small will mean that no points are core samples, and may lead to all points being labeled as noise. Setting eps to be very large will result in all points forming a single cluster.

The min_samples setting mostly determines whether points in less dense regions will be labeled as outliers or as their own clusters. If you decrease min_samples, anything that would have been a cluster with less than min_samples many samples will now be labeled as noise. min_samples therefore determines the minimum cluster size. You can see this very clearly in Figure 3-37, when going from min_samples=3 to min_samples=5 with eps=1.5. With min_samples=3, there are three clusters: one of four

points, one of five points, and one of three points. Using `min_samples=5`, the two smaller clusters (with three and four points) are now labeled as noise, and only the cluster with five samples remains.

While DBSCAN doesn't require setting the number of clusters explicitly, setting `eps` implicitly controls how many clusters will be found. Finding a good setting for `eps` is sometimes easier after scaling the data using `StandardScaler` or `MinMaxScaler`, as using these scaling techniques will ensure that all features have similar ranges.

Figure 3-38 shows the result of running DBSCAN on the `two_moons` dataset. The algorithm actually finds the two half-circles and separates them using the default settings:

In[67]:

```
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# rescale the data to zero mean and unit variance
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

dbSCAN = DBSCAN()
clusters = dbSCAN.fit_predict(X_scaled)
# plot the cluster assignments
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap=mlearn.cm2, s=60)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

As the algorithm produced the desired number of clusters (two), the parameter settings seem to work well. If we decrease `eps` to 0.2 (from the default of 0.5), we will get eight clusters, which is clearly too many. Increasing `eps` to 0.7 results in a single cluster.

When using DBSCAN, you need to be careful about handling the returned cluster assignments. The use of -1 to indicate noise might result in unexpected effects when using the cluster labels to index another array.

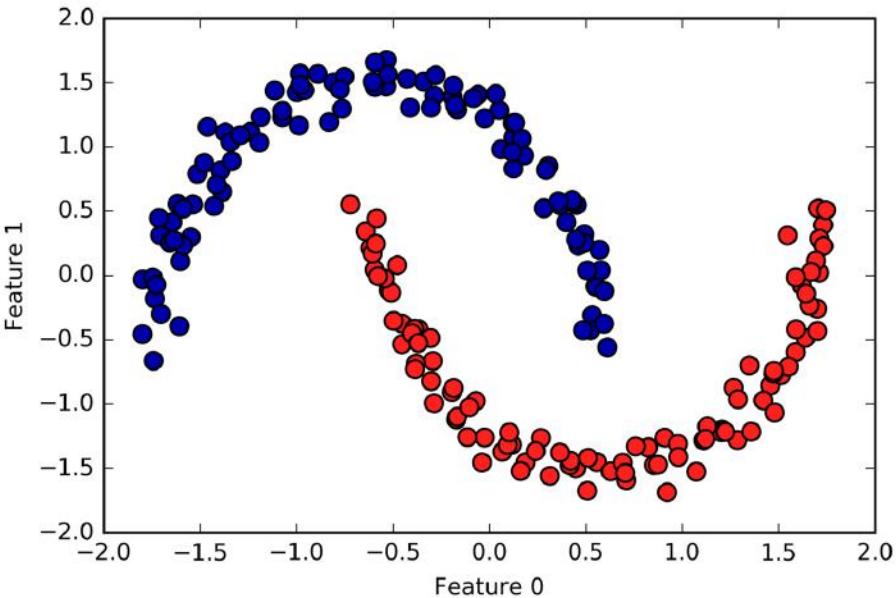


Figure 3-38. Cluster assignment found by DBSCAN using the default value of $\text{eps}=0.5$

Comparing and Evaluating Clustering Algorithms

One of the challenges in applying clustering algorithms is that it is very hard to assess how well an algorithm worked, and to compare outcomes between different algorithms. After talking about the algorithms behind k -means, agglomerative clustering, and DBSCAN, we will now compare them on some real-world datasets.

Evaluating clustering with ground truth

There are metrics that can be used to assess the outcome of a clustering algorithm relative to a ground truth clustering, the most important ones being the *adjusted rand index* (ARI) and *normalized mutual information* (NMI), which both provide a quantitative measure between 0 and 1.

Here, we compare the k -means, agglomerative clustering, and DBSCAN algorithms using ARI. We also include what it looks like when we randomly assign points to two clusters for comparison (see Figure 3-39):

In[68]:

```
from sklearn.metrics.cluster import adjusted_rand_score
X, y = make_moons(n_samples=200, noise=0.05, random_state=0)

# rescale the data to zero mean and unit variance
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)

fig, axes = plt.subplots(1, 4, figsize=(15, 3),
                      subplot_kw={'xticks': (), 'yticks': ()})

# make a list of algorithms to use
algorithms = [KMeans(n_clusters=2), AgglomerativeClustering(n_clusters=2),
              DBSCAN()]

# create a random cluster assignment for reference
random_state = np.random.RandomState(seed=0)
random_clusters = random_state.randint(low=0, high=2, size=len(X))

# plot random assignment
axes[0].scatter(X_scaled[:, 0], X_scaled[:, 1], c=random_clusters,
                 cmap=mglearn.cm3, s=60)
axes[0].set_title("Random assignment - ARI: {:.2f}".format(
    adjusted_rand_score(y, random_clusters)))

for ax, algorithm in zip(axes[1:], algorithms):
    # plot the cluster assignments and cluster centers
    clusters = algorithm.fit_predict(X_scaled)
    ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters,
               cmap=mglearn.cm3, s=60)
    ax.set_title("{} - ARI: {:.2f}".format(algorithm.__class__.__name__,
                                           adjusted_rand_score(y, clusters)))
```

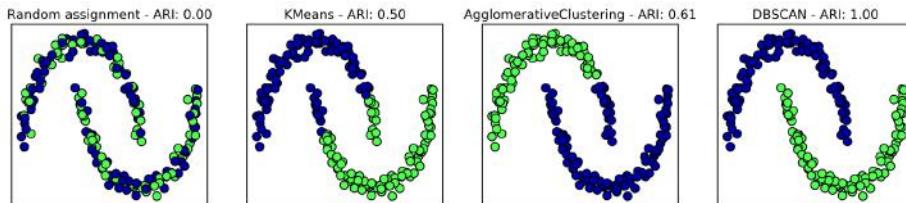


Figure 3-39. Comparing random assignment, k-means, agglomerative clustering, and DBSCAN on the two_moons dataset using the supervised ARI score

The adjusted rand index provides intuitive results, with a random cluster assignment having a score of 0 and DBSCAN (which recovers the desired clustering perfectly) having a score of 1.

A common mistake when evaluating clustering in this way is to use `accuracy_score` instead of `adjusted_rand_score`, `normalized_mutual_info_score`, or some other clustering metric. The problem in using accuracy is that it requires the assigned cluster labels to exactly match the ground truth. However, the cluster labels themselves are meaningless—the only thing that matters is which points are in the same cluster:

In[69]:

```
from sklearn.metrics import accuracy_score

# these two labelings of points correspond to the same clustering
clusters1 = [0, 0, 1, 1, 0]
clusters2 = [1, 1, 0, 0, 1]
# accuracy is zero, as none of the labels are the same
print("Accuracy: {:.2f}".format(accuracy_score(clusters1, clusters2)))
# adjusted rand score is 1, as the clustering is exactly the same
print("ARI: {:.2f}".format(adjusted_rand_score(clusters1, clusters2)))
```

Out[69]:

```
Accuracy: 0.00
ARI: 1.00
```

Evaluating clustering without ground truth

Although we have just shown one way to evaluate clustering algorithms, in practice, there is a big problem with using measures like ARI. When applying clustering algorithms, there is usually no ground truth to which to compare the results. If we knew the right clustering of the data, we could use this information to build a supervised model like a classifier. Therefore, using metrics like ARI and NMI usually only helps in developing algorithms, not in assessing success in an application.

There are scoring metrics for clustering that don't require ground truth, like the *silhouette coefficient*. However, these often don't work well in practice. The silhouette score computes the compactness of a cluster, where higher is better, with a perfect score of 1. While compact clusters are good, compactness doesn't allow for complex shapes.

Here is an example comparing the outcome of k -means, agglomerative clustering, and DBSCAN on the `two-moons` dataset using the silhouette score (Figure 3-40):

In[70]:

```
from sklearn.metrics.cluster import silhouette_score

X, y = make_moons(n_samples=200, noise=0.05, random_state=0)
# rescale the data to zero mean and unit variance
scaler = StandardScaler()
scaler.fit(X)
X_scaled = scaler.transform(X)
```

```

fig, axes = plt.subplots(1, 4, figsize=(15, 3),
                      subplot_kw={'xticks': (), 'yticks': ()})

# create a random cluster assignment for reference
random_state = np.random.RandomState(seed=0)
random_clusters = random_state.randint(low=0, high=2, size=len(X))

# plot random assignment
axes[0].scatter(X_scaled[:, 0], X_scaled[:, 1], c=random_clusters,
                 cmap=mlearn.cm3, s=60)
axes[0].set_title("Random assignment: {:.2f}".format(
    silhouette_score(X_scaled, random_clusters)))

algorithms = [KMeans(n_clusters=2), AgglomerativeClustering(n_clusters=2),
              DBSCAN()]

for ax, algorithm in zip(axes[1:], algorithms):
    clusters = algorithm.fit_predict(X_scaled)
    # plot the cluster assignments and cluster centers
    ax.scatter(X_scaled[:, 0], X_scaled[:, 1], c=clusters, cmap=mlearn.cm3,
               s=60)
    ax.set_title("{} : {:.2f}".format(algorithm.__class__.__name__,
                                      silhouette_score(X_scaled, clusters)))

```

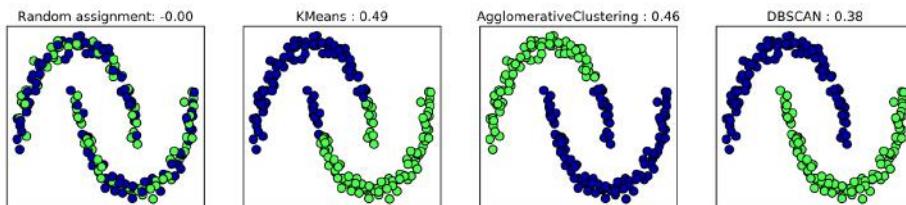


Figure 3-40. Comparing random assignment, k-means, agglomerative clustering, and DBSCAN on the two_moons dataset using the unsupervised silhouette score—the more intuitive result of DBSCAN has a lower silhouette score than the assignments found by k-means

As you can see, *k*-means gets the highest silhouette score, even though we might prefer the result produced by DBSCAN. A slightly better strategy for evaluating clusters is using *robustness-based* clustering metrics. These run an algorithm after adding some noise to the data, or using different parameter settings, and compare the outcomes. The idea is that if many algorithm parameters and many perturbations of the data return the same result, it is likely to be trustworthy. Unfortunately, this strategy is not implemented in `scikit-learn` at the time of writing.

Even if we get a very robust clustering, or a very high silhouette score, we still don't know if there is any semantic meaning in the clustering, or whether the clustering

reflects an aspect of the data that we are interested in. Let's go back to the example of face images. We hope to find groups of similar faces—say, men and women, or old people and young people, or people with beards and without. Let's say we cluster the data into two clusters, and all algorithms agree about which points should be clustered together. We still don't know if the clusters that are found correspond in any way to the concepts we are interested in. It could be that they found side views versus front views, or pictures taken at night versus pictures taken during the day, or pictures taken with iPhones versus pictures taken with Android phones. The only way to know whether the clustering corresponds to anything we are interested in is to analyze the clusters manually.

Comparing algorithms on the faces dataset

Let's apply the k -means, DBSCAN, and agglomerative clustering algorithms to the Labeled Faces in the Wild dataset, and see if any of them find interesting structure. We will use the eigenface representation of the data, as produced by `PCA(whiten=True)`, with 100 components:

In[71]:

```
# extract eigenfaces from lfw data and transform data
from sklearn.decomposition import PCA
pca = PCA(n_components=100, whiten=True, random_state=0)
pca.fit_transform(X_people)
X_pca = pca.transform(X_people)
```

We saw earlier that this is a more semantic representation of the face images than the raw pixels. It will also make computation faster. A good exercise would be for you to run the following experiments on the original data, without PCA, and see if you find similar clusters.

Analyzing the faces dataset with DBSCAN. We will start by applying DBSCAN, which we just discussed:

In[72]:

```
# apply DBSCAN with default parameters
dbSCAN = DBSCAN()
labels = dbSCAN.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

Out[72]:

```
Unique labels: [-1]
```

We see that all the returned labels are -1 , so all of the data was labeled as “noise” by DBSCAN. There are two things we can change to help this: we can make `eps` higher, to expand the neighborhood of each point, and set `min_samples` lower, to consider smaller groups of points as clusters. Let's try changing `min_samples` first:

In[73]:

```
dbSCAN = DBSCAN(min_samples=3)
labels = dbSCAN.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

Out[73]:

```
Unique labels: [-1]
```

Even when considering groups of three points, everything is labeled as noise. So, we need to increase eps:

In[74]:

```
dbSCAN = DBSCAN(min_samples=3, eps=15)
labels = dbSCAN.fit_predict(X_pca)
print("Unique labels: {}".format(np.unique(labels)))
```

Out[74]:

```
Unique labels: [-1  0]
```

Using a much larger eps of 15, we get only a single cluster and noise points. We can use this result to find out what the “noise” looks like compared to the rest of the data. To understand better what’s happening, let’s look at how many points are noise, and how many points are inside the cluster:

In[75]:

```
# Count number of points in all clusters and noise.
# bincount doesn't allow negative numbers, so we need to add 1.
# The first number in the result corresponds to noise points.
print("Number of points per cluster: {}".format(np.bincount(labels + 1)))
```

Out[75]:

```
Number of points per cluster: [ 27 2036]
```

There are very few noise points—only 27—so we can look at all of them (see Figure 3-41):

In[76]:

```
noise = X_people[labels == -1]

fig, axes = plt.subplots(3, 9, subplot_kw={'xticks': (), 'yticks': ()},
                       figsize=(12, 4))
for image, ax in zip(noise, axes.ravel()):
    ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
```



Figure 3-41. Samples from the faces dataset labeled as noise by DBSCAN

Comparing these images to the random sample of face images from Figure 3-7, we can guess why they were labeled as noise: the fifth image in the first row shows a person drinking from a glass, there are images of people wearing hats, and in the last image there's a hand in front of the person's face. The other images contain odd angles or crops that are too close or too wide.

This kind of analysis—trying to find “the odd one out”—is called *outlier detection*. If this was a real application, we might try to do a better job of cropping images, to get more homogeneous data. There is little we can do about people in photos sometimes wearing hats, drinking, or holding something in front of their faces, but it’s good to know that these are issues in the data that any algorithm we might apply needs to handle.

If we want to find more interesting clusters than just one large one, we need to set `eps` smaller, somewhere between 15 and 0.5 (the default). Let’s have a look at what different values of `eps` result in:

In[77]:

```
for eps in [1, 3, 5, 7, 9, 11, 13]:
    print("\neps={}".format(eps))
    dbscan = DBSCAN(eps=eps, min_samples=3)
    labels = dbscan.fit_predict(X_pca)
    print("Clusters present: {}".format(np.unique(labels)))
    print("Cluster sizes: {}".format(np.bincount(labels + 1)))
```

Out[78]:

```
eps=1
Clusters present: [-1]
Cluster sizes: [2063]

eps=3
Clusters present: [-1]
Cluster sizes: [2063]
```

```

eps=5
Clusters present: [-1]
Cluster sizes: [2063]

eps=7
Clusters present: [-1  0  1  2  3  4  5  6  7  8  9 10 11 12]
Cluster sizes: [2006  4  6  6  6  9  3  3  4  3  3  3  3  3  4]

eps=9
Clusters present: [-1  0  1  2]
Cluster sizes: [1269  788    3    3]

eps=11
Clusters present: [-1  0]
Cluster sizes: [ 430 1633]

eps=13
Clusters present: [-1  0]
Cluster sizes: [ 112 1951]

```

For low settings of `eps`, all points are labeled as noise. For `eps=7`, we get many noise points and many smaller clusters. For `eps=9` we still get many noise points, but we get one big cluster and some smaller clusters. Starting from `eps=11`, we get only one large cluster and noise.

What is interesting to note is that there is never more than one large cluster. At most, there is one large cluster containing most of the points, and some smaller clusters. This indicates that there are not two or three different kinds of face images in the data that are very distinct, but rather that all images are more or less equally similar to (or dissimilar from) the rest.

The results for `eps=7` look most interesting, with many small clusters. We can investigate this clustering in more detail by visualizing all of the points in each of the 13 small clusters ([Figure 3-42](#)):

In[78]:

```

dbSCAN = DBSCAN(min_samples=3, eps=7)
labels = dbSCAN.fit_predict(X_pca)

for cluster in range(max(labels) + 1):
    mask = labels == cluster
    n_images = np.sum(mask)
    fig, axes = plt.subplots(1, n_images, figsize=(n_images * 1.5, 4),
                           subplot_kw={'xticks': (), 'yticks': ()})
    for image, label, ax in zip(X_people[mask], y_people[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1])

```



Figure 3-42. Clusters found by DBSCAN with $\text{eps}=7$

Some of the clusters correspond to people with very distinct faces (within this dataset), such as Sharon or Koizumi. Within each cluster, the orientation of the face is also

quite fixed, as well as the facial expression. Some of the clusters contain faces of multiple people, but they share a similar orientation and expression.

This concludes our analysis of the DBSCAN algorithm applied to the faces dataset. As you can see, we are doing a manual analysis here, different from the much more automatic search approach we could use for supervised learning based on R^2 score or accuracy.

Let's move on to applying k -means and agglomerative clustering.

Analyzing the faces dataset with k -means. We saw that it was not possible to create more than one big cluster using DBSCAN. Agglomerative clustering and k -means are much more likely to create clusters of even size, but we do need to set a target number of clusters. We could set the number of clusters to the known number of people in the dataset, though it is very unlikely that an unsupervised clustering algorithm will recover them. Instead, we can start with a low number of clusters, like 10, which might allow us to analyze each of the clusters:

In[79]:

```
# extract clusters with k-means
km = KMeans(n_clusters=10, random_state=0)
labels_km = km.fit_predict(X_pca)
print("Cluster sizes k-means: {}".format(np.bincount(labels_km)))
```

Out[79]:

```
Cluster sizes k-means: [269 128 170 186 386 222 237 64 253 148]
```

As you can see, k -means clustering partitioned the data into relatively similarly sized clusters from 64 to 386. This is quite different from the result of DBSCAN.

We can further analyze the outcome of k -means by visualizing the cluster centers ([Figure 3-43](#)). As we clustered in the representation produced by PCA, we need to rotate the cluster centers back into the original space to visualize them, using `pca.inverse_transform`:

In[80]:

```
fig, axes = plt.subplots(2, 5, subplot_kw={'xticks': (), 'yticks': ()},
                       figsize=(12, 4))
for center, ax in zip(km.cluster_centers_, axes.ravel()):
    ax.imshow(pca.inverse_transform(center).reshape(image_shape),
              vmin=0, vmax=1)
```



Figure 3-43. Cluster centers found by k -means when setting the number of clusters to 10

The cluster centers found by k -means are very smooth versions of faces. This is not very surprising, given that each center is an average of 64 to 386 face images. Working with a reduced PCA representation adds to the smoothness of the images (compared to the faces reconstructed using 100 PCA dimensions in Figure 3-11). The clustering seems to pick up on different orientations of the face, different expressions (the third cluster center seems to show a smiling face), and the presence of shirt collars (see the second-to-last cluster center).

For a more detailed view, in Figure 3-44 we show for each cluster center the five most typical images in the cluster (the images assigned to the cluster that are closest to the cluster center) and the five most atypical images in the cluster (the images assigned to the cluster that are furthest from the cluster center):

In[81]:

```
mglearn.plots.plot_kmeans_faces(km, pca, X_pca, X_people,  
y_people, people.target_names)
```



Figure 3-44. Sample images for each cluster found by k-means—the cluster centers are on the left, followed by the five closest points to each center and the five points that are assigned to the cluster but are furthest away from the center

[Figure 3-44](#) confirms our intuition about smiling faces for the third cluster, and also the importance of orientation for the other clusters. The “atypical” points are not very similar to the cluster centers, though, and their assignment seems somewhat arbitrary. This can be attributed to the fact that k -means partitions all the data points and doesn’t have a concept of “noise” points, as DBSCAN does. Using a larger number of clusters, the algorithm could find finer distinctions. However, adding more clusters makes manual inspection even harder.

Analyzing the faces dataset with agglomerative clustering. Now, let’s look at the results of agglomerative clustering:

In[82]:

```
# extract clusters with ward agglomerative clustering
agglomerative = AgglomerativeClustering(n_clusters=10)
labels_agg = agglomerative.fit_predict(X_pca)
print("Cluster sizes agglomerative clustering: {}".format(
    np.bincount(labels_agg)))
```

Out[82]:

```
Cluster sizes agglomerative clustering: [255 623  86 102 122 199 265  26 230 155]
```

Agglomerative clustering also produces relatively equally sized clusters, with cluster sizes between 26 and 623. These are more uneven than those produced by k -means, but much more even than the ones produced by DBSCAN.

We can compute the ARI to measure whether the two partitions of the data given by agglomerative clustering and k -means are similar:

In[83]:

```
print("ARI: {:.2f}".format(adjusted_rand_score(labels_agg, labels_km)))
```

Out[83]:

```
ARI: 0.13
```

An ARI of only 0.13 means that the two clusterings `labels_agg` and `labels_km` have little in common. This is not very surprising, given the fact that points further away from the cluster centers seem to have little in common for k -means.

Next, we might want to plot the dendrogram ([Figure 3-45](#)). We’ll limit the depth of the tree in the plot, as branching down to the individual 2,063 data points would result in an unreadably dense plot:

In[84]:

```
linkage_array = ward(X_pca)
# now we plot the dendrogram for the linkage_array
# containing the distances between clusters
plt.figure(figsize=(20, 5))
dendrogram(linkage_array, p=7, truncate_mode='level', no_labels=True)
plt.xlabel("Sample index")
plt.ylabel("Cluster distance")
```

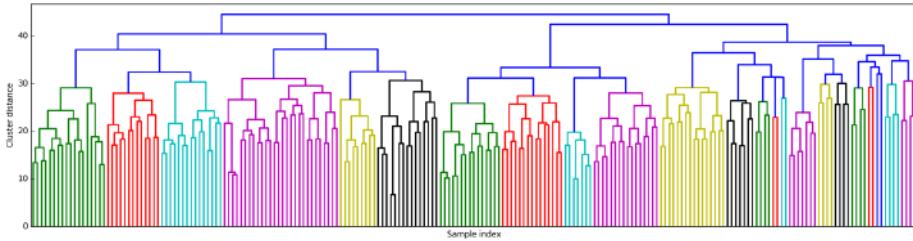


Figure 3-45. Dendrogram of agglomerative clustering on the faces dataset

Creating 10 clusters, we cut across the tree at the very top, where there are 10 vertical lines. In the dendrogram for the toy data shown in Figure 3-36, you could see by the length of the branches that two or three clusters might capture the data appropriately. For the faces data, there doesn't seem to be a very natural cutoff point. There are some branches that represent more distinct groups, but there doesn't appear to be a particular number of clusters that is a good fit. This is not surprising, given the results of DBSCAN, which tried to cluster all points together.

Let's visualize the 10 clusters, as we did for k -means earlier (Figure 3-46). Note that there is no notion of cluster center in agglomerative clustering (though we could compute the mean), and we simply show the first couple of points in each cluster. We show the number of points in each cluster to the left of the first image:

In[85]:

```
n_clusters = 10
for cluster in range(n_clusters):
    mask = labels_agg == cluster
    fig, axes = plt.subplots(1, 10, subplot_kw={'xticks': (), 'yticks': ()},
                           figsize=(15, 8))
    axes[0].set_ylabel(np.sum(mask))
    for image, label, asdf, ax in zip(X_people[mask], y_people[mask],
                                      labels_agg[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1],
                     fontdict={'fontsize': 9})
```



Figure 3-46. Random images from the clusters generated by In[82]—each row corresponds to one cluster; the number to the left lists the number of images in each cluster

While some of the clusters seem to have a semantic theme, many of them are too large to be actually homogeneous. To get more homogeneous clusters, we can run the algorithm again, this time with 40 clusters, and pick out some of the clusters that are particularly interesting (Figure 3-47):

In[86]:

```
# extract clusters with ward agglomerative clustering
agglomerative = AgglomerativeClustering(n_clusters=40)
labels_agg = agglomerative.fit_predict(X_pca)
print("cluster sizes agglomerative clustering: {}".format(np.bincount(labels_agg)))

n_clusters = 40
for cluster in [10, 13, 19, 22, 36]: # hand-picked "interesting" clusters
    mask = labels_agg == cluster
    fig, axes = plt.subplots(1, 15, subplot_kw={'xticks': (), 'yticks': ()},
                           figsize=(15, 8))
    cluster_size = np.sum(mask)
    axes[0].set_ylabel("#{}: {}".format(cluster, cluster_size))
    for image, label, asdf, ax in zip(X_people[mask], y_people[mask],
                                      labels_agg[mask], axes):
        ax.imshow(image.reshape(image_shape), vmin=0, vmax=1)
        ax.set_title(people.target_names[label].split()[-1],
                     fontdict={'fontsize': 9})
    for i in range(cluster_size, 15):
        axes[i].set_visible(False)
```

Out[86]:

```
cluster sizes agglomerative clustering:
[ 58  80  79  40 222  50  55  78 172  28  26  34  14  11  60  66 152  27
 47  31  54   5   8  56   3   5   8  18  22  82  37  89  28  24  41  40
 21  10 113  69]
```



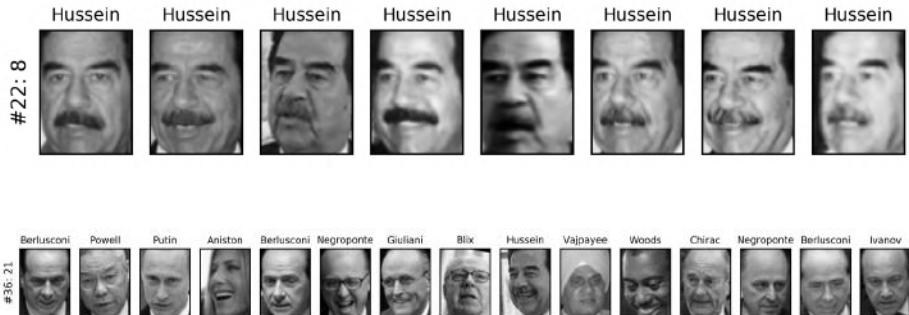


Figure 3-47. Images from selected clusters found by agglomerative clustering when setting the number of clusters to 40—the text to the left shows the index of the cluster and the total number of points in the cluster

Here, the clustering seems to have picked up on “dark skinned and smiling,” “collared shirt,” “smiling woman,” “Hussein,” and “high forehead.” We could also find these highly similar clusters using the dendrogram, if we did more a detailed analysis.

Summary of Clustering Methods

This section has shown that applying and evaluating clustering is a highly qualitative procedure, and often most helpful in the exploratory phase of data analysis. We looked at three clustering algorithms: k -means, DBSCAN, and agglomerative clustering. All three have a way of controlling the granularity of clustering. k -means and agglomerative clustering allow you to specify the number of desired clusters, while DBSCAN lets you define proximity using the `eps` parameter, which indirectly influences cluster size. All three methods can be used on large, real-world datasets, are relatively easy to understand, and allow for clustering into many clusters.

Each of the algorithms has somewhat different strengths. k -means allows for a characterization of the clusters using the cluster means. It can also be viewed as a decomposition method, where each data point is represented by its cluster center. DBSCAN allows for the detection of “noise points” that are not assigned any cluster, and it can help automatically determine the number of clusters. In contrast to the other two methods, it allows for complex cluster shapes, as we saw in the `two_moons` example. DBSCAN sometimes produces clusters of very differing size, which can be a strength or a weakness. Agglomerative clustering can provide a whole hierarchy of possible partitions of the data, which can be easily inspected via dendrograms.

Summary and Outlook

This chapter introduced a range of unsupervised learning algorithms that can be applied for exploratory data analysis and preprocessing. Having the right representation of the data is often crucial for supervised or unsupervised learning to succeed, and preprocessing and decomposition methods play an important part in data preparation.

Decomposition, manifold learning, and clustering are essential tools to further your understanding of your data, and can be the only ways to make sense of your data in the absence of supervision information. Even in a supervised setting, exploratory tools are important for a better understanding of the properties of the data. Often it is hard to quantify the usefulness of an unsupervised algorithm, though this shouldn't deter you from using them to gather insights from your data. With these methods under your belt, you are now equipped with all the essential learning algorithms that machine learning practitioners use every day.

We encourage you to try clustering and decomposition methods both on two-dimensional toy data and on real-world datasets included in `scikit-learn`, like the `digits`, `iris`, and `cancer` datasets.

Summary of the Estimator Interface

Let's briefly review the API that we introduced in Chapters 2 and 3. All algorithms in scikit-learn, whether preprocessing, supervised learning, or unsupervised learning algorithms, are implemented as classes. These classes are called *estimators* in scikit-learn. To apply an algorithm, you first have to instantiate an object of the particular class:

In[87]:

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression()
```

The estimator class contains the algorithm, and also stores the model that is learned from data using the algorithm.

You should set any parameters of the model when constructing the model object. These parameters include regularization, complexity control, number of clusters to find, etc. All estimators have a `fit` method, which is used to build the model. The `fit` method always requires as its first argument the data `X`, represented as a NumPy array or a SciPy sparse matrix, where each row represents a single data point. The data `X` is always assumed to be a NumPy array or SciPy sparse matrix that has continuous (floating-point) entries. Supervised algorithms also require a `y` argument, which is a one-dimensional NumPy array containing target values for regression or classification (i.e., the known output labels or responses).

There are two main ways to apply a learned model in scikit-learn. To create a prediction in the form of a new output like `y`, you use the `predict` method. To create a new representation of the input data `X`, you use the `transform` method. Table 3-1 summarizes the use cases of the `predict` and `transform` methods.

Table 3-1. scikit-learn API summary

	<code>estimator.fit(x_train, [y_train])</code>
	<code>estimator.predict(X_text)</code> <code>estimator.transform(X_test)</code>
Classification	Preprocessing
Regression	Dimensionality reduction
Clustering	Feature extraction
	Feature selection

Additionally, all supervised models have a `score(X_test, y_test)` method that allows an evaluation of the model. In Table 3-1, `X_train` and `y_train` refer to the training data and training labels, while `X_test` and `y_test` refer to the test data and test labels (if applicable).

CHAPTER 4

Representing Data and Engineering Features

So far, we've assumed that our data comes in as a two-dimensional array of floating-point numbers, where each column is a *continuous feature* that describes the data points. For many applications, this is not how the data is collected. A particularly common type of feature is the *categorical features*. Also known as *discrete features*, these are usually not numeric. The distinction between categorical features and continuous features is analogous to the distinction between classification and regression, only on the input side rather than the output side. Examples of continuous features that we have seen are pixel brightnesses and size measurements of plant flowers. Examples of categorical features are the brand of a product, the color of a product, or the department (books, clothing, hardware) it is sold in. These are all properties that can describe a product, but they don't vary in a continuous way. A product belongs either in the clothing department or in the books department. There is no middle ground between books and clothing, and no natural order for the different categories (books is not greater or less than clothing, hardware is not between books and clothing, etc.).

Regardless of the types of features your data consists of, how you represent them can have an enormous effect on the performance of machine learning models. We saw in Chapters 2 and 3 that scaling of the data is important. In other words, if you don't rescale your data (say, to unit variance), then it makes a difference whether you represent a measurement in centimeters or inches. We also saw in Chapter 2 that it can be helpful to *augment* your data with additional features, like adding interactions (products) of features or more general polynomials.

The question of how to represent your data best for a particular application is known as *feature engineering*, and it is one of the main tasks of data scientists and machine

learning practitioners trying to solve real-world problems. Representing your data in the right way can have a bigger influence on the performance of a supervised model than the exact parameters you choose.

In this chapter, we will first go over the important and very common case of categorical features, and then give some examples of helpful transformations for specific combinations of features and models.

Categorical Variables

As an example, we will use the dataset of adult incomes in the United States, derived from the 1994 census database. The task of the `adult` dataset is to predict whether a worker has an income of over \$50,000 or under \$50,000. The features in this dataset include the workers' ages, how they are employed (self employed, private industry employee, government employee, etc.), their education, their gender, their working hours per week, occupation, and more. [Table 4-1](#) shows the first few entries in the dataset.

Table 4-1. The first few entries in the adult dataset

	age	workclass	education	gender	hours-per-week	occupation	income
0	39	State-gov	Bachelors	Male	40	Adm-clerical	<=50K
1	50	Self-emp-not-inc	Bachelors	Male	13	Exec-managerial	<=50K
2	38	Private	HS-grad	Male	40	Handlers-cleaners	<=50K
3	53	Private	11th	Male	40	Handlers-cleaners	<=50K
4	28	Private	Bachelors	Female	40	Prof-specialty	<=50K
5	37	Private	Masters	Female	40	Exec-managerial	<=50K
6	49	Private	9th	Female	16	Other-service	<=50K
7	52	Self-emp-not-inc	HS-grad	Male	45	Exec-managerial	>50K
8	31	Private	Masters	Female	50	Prof-specialty	>50K
9	42	Private	Bachelors	Male	40	Exec-managerial	>50K
10	37	Private	Some-college	Male	80	Exec-managerial	>50K

The task is phrased as a classification task with the two classes being income $\leq 50\text{K}$ and $> 50\text{K}$. It would also be possible to predict the exact income, and make this a regression task. However, that would be much more difficult, and the 50K division is interesting to understand on its own.

In this dataset, `age` and `hours-per-week` are continuous features, which we know how to treat. The `workclass`, `education`, `sex`, and `occupation` features are categorical, however. All of them come from a fixed list of possible values, as opposed to a range, and denote a qualitative property, as opposed to a quantity.

As a starting point, let's say we want to learn a logistic regression classifier on this data. We know from [Chapter 2](#) that a logistic regression makes predictions, \hat{y} , using the following formula:

$$\hat{y} = w[0] * x[0] + w[1] * x[1] + \dots + w[p] * x[p] + b > 0$$

where $w[i]$ and b are coefficients learned from the training set and $x[i]$ are the input features. This formula makes sense when $x[i]$ are numbers, but not when $x[2]$ is "Masters" or "Bachelors". Clearly we need to represent our data in some different way when applying logistic regression. The next section will explain how we can overcome this problem.

One-Hot-Encoding (Dummy Variables)

By far the most common way to represent categorical variables is using the *one-hot-encoding* or *one-out-of-N encoding*, also known as *dummy variables*. The idea behind dummy variables is to replace a categorical variable with one or more new features that can have the values 0 and 1. The values 0 and 1 make sense in the formula for linear binary classification (and for all other models in `scikit-learn`), and we can represent any number of categories by introducing one new feature per category, as described here.

Let's say for the `workclass` feature we have possible values of "Government Employee", "Private Employee", "Self Employed", and "Self Employed Incorporated". To encode these four possible values, we create four new features, called "Government Employee", "Private Employee", "Self Employed", and "Self Employed Incorporated". A feature is 1 if `workclass` for this person has the corresponding value and 0 otherwise, so exactly one of the four new features will be 1 for each data point. This is why this is called one-hot or one-out-of-N encoding.

The principle is illustrated in [Table 4-2](#). A single feature is encoded using four new features. When using this data in a machine learning algorithm, we would drop the original `workclass` feature and only keep the 0-1 features.

Table 4-2. Encoding the `workclass` feature using one-hot encoding

<code>workclass</code>	Government Employee	Private Employee	Self Employed	Self Employed Incorporated
Government Employee	1	0	0	0
Private Employee	0	1	0	0
Self Employed	0	0	1	0
Self Employed Incorporated	0	0	0	1



The one-hot encoding we use is quite similar, but not identical, to the dummy encoding used in statistics. For simplicity, we encode each category with a different binary feature. In statistics, it is common to encode a categorical feature with k different possible values into $k-1$ features (the last one is represented as all zeros). This is done to simplify the analysis (more technically, this will avoid making the data matrix rank-deficient).

There are two ways to convert your data to a one-hot encoding of categorical variables, using either `pandas` or `scikit-learn`. At the time of writing, using `pandas` is slightly easier, so let's go this route. First we load the data using `pandas` from a comma-separated values (CSV) file:

In[2]:

```
import pandas as pd
# The file has no headers naming the columns, so we pass header=None
# and provide the column names explicitly in "names"
data = pd.read_csv(
    "/home/andy/datasets/adult.data", header=None, index_col=False,
    names=['age', 'workclass', 'fnlwgt', 'education', 'education-num',
           'marital-status', 'occupation', 'relationship', 'race', 'gender',
           'capital-gain', 'capital-loss', 'hours-per-week', 'native-country',
           'income'])
# For illustration purposes, we only select some of the columns
data = data[['age', 'workclass', 'education', 'gender', 'hours-per-week',
             'occupation', 'income']]
# IPython.display allows nice output formatting within the Jupyter notebook
display(data.head())
```

Table 4-3 shows the result.

Table 4-3. The first five rows of the adult dataset

	age	workclass	education	gender	hours-per-week	occupation	income
0	39	State-gov	Bachelors	Male	40	Adm-clerical	<=50K
1	50	Self-emp-not-inc	Bachelors	Male	13	Exec-managerial	<=50K
2	38	Private	HS-grad	Male	40	Handlers-cleaners	<=50K
3	53	Private	11th	Male	40	Handlers-cleaners	<=50K
4	28	Private	Bachelors	Female	40	Prof-specialty	<=50K

Checking string-encoded categorical data

After reading a dataset like this, it is often good to first check if a column actually contains meaningful categorical data. When working with data that was input by humans (say, users on a website), there might not be a fixed set of categories, and differences in spelling and capitalization might require preprocessing. For example, it might be that some people specified gender as “male” and some as “man,” and we

might want to represent these two inputs using the same category. A good way to check the contents of a column is using the `value_counts` function of a `pandas Series` (the type of a single column in a `DataFrame`), to show us what the unique values are and how often they appear:

In[3]:

```
print(data.gender.value_counts())
```

Out[3]:

```
Male      21790  
Female    10771  
Name: gender, dtype: int64
```

We can see that there are exactly two values for gender in this dataset, `Male` and `Female`, meaning the data is already in a good format to be represented using one-hot-encoding. In a real application, you should look at all columns and check their values. We will skip this here for brevity's sake.

There is a very simple way to encode the data in `pandas`, using the `get_dummies` function. The `get_dummies` function automatically transforms all columns that have object type (like strings) or are categorical (which is a special `pandas` concept that we haven't talked about yet):

In[4]:

```
print("Original features:\n", list(data.columns), "\n")  
data_dummies = pd.get_dummies(data)  
print("Features after get_dummies:\n", list(data_dummies.columns))
```

Out[4]:

```
Original features:  
['age', 'workclass', 'education', 'gender', 'hours-per-week', 'occupation',  
'income']  
  
Features after get_dummies:  
['age', 'hours-per-week', 'workclass_ ?', 'workclass_Federal-gov',  
'workclass_Local-gov', 'workclass_Never-worked', 'workclass_Private',  
'workclass_Self-emp-inc', 'workclass_Self-emp-not-inc',  
'workclass_State-gov', 'workclass_Without-pay', 'education_10th',  
'education_11th', 'education_12th', 'education_1st-4th',  
...  
'education_Preschool', 'education_Prof-school', 'education_Some-college',  
'gender_Female', 'gender_Male', 'occupation_?',  
'occupation_Adm-clerical', 'occupation_Armed-Forces',  
'occupation_Craft-repair', 'occupation_Exec-managerial',  
'occupation_Farming-fishing', 'occupation_Handlers-cleaners',  
...  
'occupation_Tech-support', 'occupation_Transport-moving',  
'income_<=50K', 'income_>50K']
```

You can see that the continuous features `age` and `hours-per-week` were not touched, while the categorical features were expanded into one new feature for each possible value:

In[5]:

```
data_dummies.head()
```

Out[5]:

	age	hours-per-week	workclass_?	workclass_Federal-gov	workclass_Local-gov	...	occupation_Tech-support	occupation_Transport-moving	income_<=50K	income_>50K
0	39	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
1	50	13	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
2	38	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
3	53	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0
4	28	40	0.0	0.0	0.0	...	0.0	0.0	1.0	0.0

5 rows × 46 columns

We can now use the `values` attribute to convert the `data_dummies` DataFrame into a NumPy array, and then train a machine learning model on it. Be careful to separate the target variable (which is now encoded in two `income` columns) from the data before training a model. Including the output variable, or some derived property of the output variable, into the feature representation is a very common mistake in building supervised machine learning models.



Be careful: column indexing in pandas includes the end of the range, so `'age':'occupation_Transport-moving'` is inclusive of `occupation_Transport-moving`. This is different from slicing a NumPy array, where the end of a range is not included: for example, `np.arange(11)[0:10]` doesn't include the entry with index 10.

In this case, we extract only the columns containing features—that is, all columns from `age` to `occupation_Transport-moving`. This range contains all the features but not the target:

In[6]:

```
features = data_dummies.ix[:, 'age':'occupation_Transport-moving']
# Extract NumPy arrays
X = features.values
y = data_dummies['income_>50K'].values
print("X.shape: {} y.shape: {}".format(X.shape, y.shape))
```

Out[6]:

```
X.shape: (32561, 44) y.shape: (32561,)
```

Now the data is represented in a way that `scikit-learn` can work with, and we can proceed as usual:

In[7]:

```
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(X, y, random_state=0)
logreg = LogisticRegression()
logreg.fit(X_train, y_train)
print("Test score: {:.2f}".format(logreg.score(X_test, y_test)))
```

Out[7]:

```
Test score: 0.81
```



In this example, we called `get_dummies` on a `DataFrame` containing both the training and the test data. This is important to ensure categorical values are represented in the same way in the training set and the test set.

Imagine we have the training and test sets in two different `DataFrames`. If the "Private Employee" value for the `workclass` feature does not appear in the test set, `pandas` will assume there are only three possible values for this feature and will create only three new dummy features. Now our training and test sets have different numbers of features, and we can't apply the model we learned on the training set to the test set anymore. Even worse, imagine the `workclass` feature has the values "Government Employee" and "Private Employee" in the training set, and "Self Employed" and "Self Employed Incorporated" in the test set. In both cases, `pandas` will create two new dummy features, so the encoded `DataFrames` will have the same number of features. However, the two dummy features have entirely different meanings in the training and test sets. The column that means "Government Employee" for the training set would encode "Self Employed" for the test set.

If we built a machine learning model on this data it would work very badly, because it would assume the columns mean the same things (because they are in the same position) when in fact they mean very different things. To fix this, either call `get_dummies` on a `DataFrame` that contains both the training and the test data points, or make sure that the column names are the same for the training and test sets after calling `get_dummies`, to ensure they have the same semantics.

Numbers Can Encode Categoricals

In the example of the `adult` dataset, the categorical variables were encoded as strings. On the one hand, that opens up the possibility of spelling errors, but on the other hand, it clearly marks a variable as categorical. Often, whether for ease of storage or because of the way the data is collected, categorical variables are encoded as integers. For example, imagine the census data in the `adult` dataset was collected using a questionnaire, and the answers for `workclass` were recorded as 0 (first box ticked), 1 (second box ticked), 2 (third box ticked), and so on. Now the column will contain numbers from 0 to 8, instead of strings like "Private", and it won't be immediately obvious to someone looking at the table representing the dataset whether they should treat this variable as continuous or categorical. Knowing that the numbers indicate employment status, however, it is clear that these are very distinct states and should not be modeled by a single continuous variable.



Categorical features are often encoded using integers. That they are numbers doesn't mean that they should necessarily be treated as continuous features. It is not always clear whether an integer feature should be treated as continuous or discrete (and one-hot-encoded). If there is no ordering between the semantics that are encoded (like in the `workclass` example), the feature must be treated as discrete. For other cases, like five-star ratings, the better encoding depends on the particular task and data and which machine learning algorithm is used.

The `get_dummies` function in `pandas` treats all numbers as continuous and will not create dummy variables for them. To get around this, you can either use `scikit-learn's OneHotEncoder`, for which you can specify which variables are continuous and which are discrete, or convert numeric columns in the `DataFrame` to strings. To illustrate, let's create a `DataFrame` object with two columns, one containing strings and one containing integers:

In[8]:

```
# create a DataFrame with an integer feature and a categorical string feature
demo_df = pd.DataFrame({'Integer Feature': [0, 1, 2, 1],
                        'Categorical Feature': ['socks', 'fox', 'socks', 'box']})
display(demo_df)
```

Table 4-4 shows the result.

Table 4-4. DataFrame containing categorical string features and integer features

Categorical Feature	Integer Feature
0 socks	0
1 fox	1
2 socks	2
3 box	1

Using `get_dummies` will only encode the string feature and will not change the integer feature, as you can see in [Table 4-5](#):

In[9]:

```
pd.get_dummies(demo_df)
```

Table 4-5. One-hot-encoded version of the data from [Table 4-4](#), leaving the integer feature unchanged

	Integer Feature	Categorical Feature_box	Categorical Feature_fox	Categorical Feature_socks
0	0	0.0	0.0	1.0
1	1	0.0	1.0	0.0
2	2	0.0	0.0	1.0
3	1	1.0	0.0	0.0

If you want dummy variables to be created for the “Integer Feature” column, you can explicitly list the columns you want to encode using the `columns` parameter. Then, both features will be treated as categorical (see [Table 4-6](#)):

In[10]:

```
demo_df['Integer Feature'] = demo_df['Integer Feature'].astype(str)
pd.get_dummies(demo_df, columns=['Integer Feature', 'Categorical Feature'])
```

Table 4-6. One-hot encoding of the data shown in [Table 4-4](#), encoding the integer and string features

	Integer Feature_0	Integer Feature_1	Integer Feature_2	Categorical Feature_box	Categorical Feature_fox	Categorical Feature_socks
0	1.0	0.0	0.0	0.0	0.0	1.0
1	0.0	1.0	0.0	0.0	1.0	0.0
2	0.0	0.0	1.0	0.0	0.0	1.0
3	0.0	1.0	0.0	1.0	0.0	0.0

Binning, Discretization, Linear Models, and Trees

The best way to represent data depends not only on the semantics of the data, but also on the kind of model you are using. Linear models and tree-based models (such as decision trees, gradient boosted trees, and random forests), two large and very commonly used families, have very different properties when it comes to how they work with different feature representations. Let's go back to the wave regression dataset that we used in [Chapter 2](#). It has only a single input feature. Here is a comparison of a linear regression model and a decision tree regressor on this dataset (see [Figure 4-1](#)):

In[11]:

```
from sklearn.linear_model import LinearRegression
from sklearn.tree import DecisionTreeRegressor

X, y = mglearn.datasets.make_wave(n_samples=100)
line = np.linspace(-3, 3, 1000, endpoint=False).reshape(-1, 1)

reg = DecisionTreeRegressor(min_samples_split=3).fit(X, y)
plt.plot(line, reg.predict(line), label="decision tree")

reg = LinearRegression().fit(X, y)
plt.plot(line, reg.predict(line), label="linear regression")

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.legend(loc="best")
```

As you know, linear models can only model linear relationships, which are lines in the case of a single feature. The decision tree can build a much more complex model of the data. However, this is strongly dependent on the representation of the data. One way to make linear models more powerful on continuous data is to use *binning* (also known as *discretization*) of the feature to split it up into multiple features, as described here.

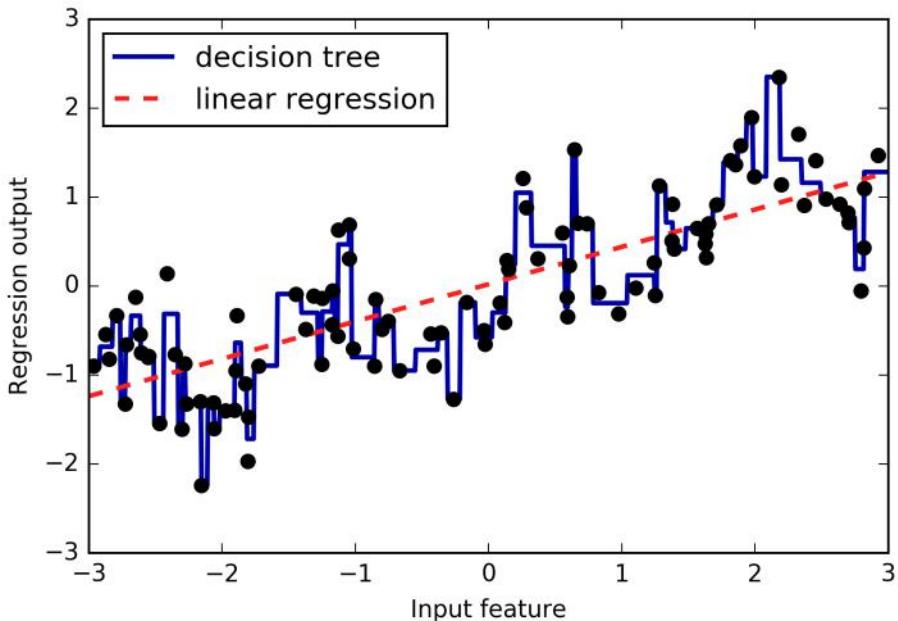


Figure 4-1. Comparing linear regression and a decision tree on the wave dataset

We imagine a partition of the input range for the feature (in this case, the numbers from -3 to 3) into a fixed number of *bins*—say, 10 . A data point will then be represented by which bin it falls into. To determine this, we first have to define the bins. In this case, we'll define 10 bins equally spaced between -3 and 3 . We use the `np.linspace` function for this, creating 11 entries, which will create 10 bins—they are the spaces in between two consecutive boundaries:

In[12]:

```
bins = np.linspace(-3, 3, 11)
print("bins: {}".format(bins))
```

Out[12]:

```
bins: [-3. -2.4 -1.8 -1.2 -0.6  0.   0.6  1.2  1.8  2.4  3. ]
```

Here, the first bin contains all data points with feature values -3 to -2.68 , the second bin contains all points with feature values from -2.68 to -2.37 , and so on.

Next, we record for each data point which bin it falls into. This can be easily computed using the `np.digitize` function:

In[13]:

```
which_bin = np.digitize(X, bins=bins)
print("\nData points:\n", X[:5])
print("\nBin membership for data points:\n", which_bin[:5])
```

Out[13]:

```
Data points:
[[-0.753]
 [ 2.704]
 [ 1.392]
 [ 0.592]
 [-2.064]]

Bin membership for data points:
[[ 4]
 [10]
 [ 8]
 [ 6]
 [ 2]]
```

What we did here is transform the single continuous input feature in the wave dataset into a categorical feature that encodes which bin a data point is in. To use a scikit-learn model on this data, we transform this discrete feature to a one-hot encoding using the `OneHotEncoder` from the `preprocessing` module. The `OneHotEncoder` does the same encoding as `pandas.get_dummies`, though it currently only works on categorical variables that are integers:

In[14]:

```
from sklearn.preprocessing import OneHotEncoder
# transform using the OneHotEncoder
encoder = OneHotEncoder(sparse=False)
# encoder.fit finds the unique values that appear in which_bin
encoder.fit(which_bin)
# transform creates the one-hot encoding
X_binned = encoder.transform(which_bin)
print(X_binned[:5])
```

Out[14]:

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

Because we specified 10 bins, the transformed dataset `X_binned` now is made up of 10 features:

In[15]:

```
print("X_binned.shape: {}".format(X_binned.shape))
```

Out[15]:

```
X_binned.shape: (100, 10)
```

Now we build a new linear regression model and a new decision tree model on the one-hot-encoded data. The result is visualized in [Figure 4-2](#), together with the bin boundaries, shown as dotted black lines:

In[16]:

```
line_binned = encoder.transform(np.digitize(line, bins=bins))

reg = LinearRegression().fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='linear regression binned')

reg = DecisionTreeRegressor(min_samples_split=3).fit(X_binned, y)
plt.plot(line, reg.predict(line_binned), label='decision tree binned')
plt.plot(X[:, 0], y, 'o', c='k')
plt.vlines(bins, -3, 3, linewidth=1, alpha=.2)
plt.legend(loc="best")
plt.ylabel("Regression output")
plt.xlabel("Input feature")
```

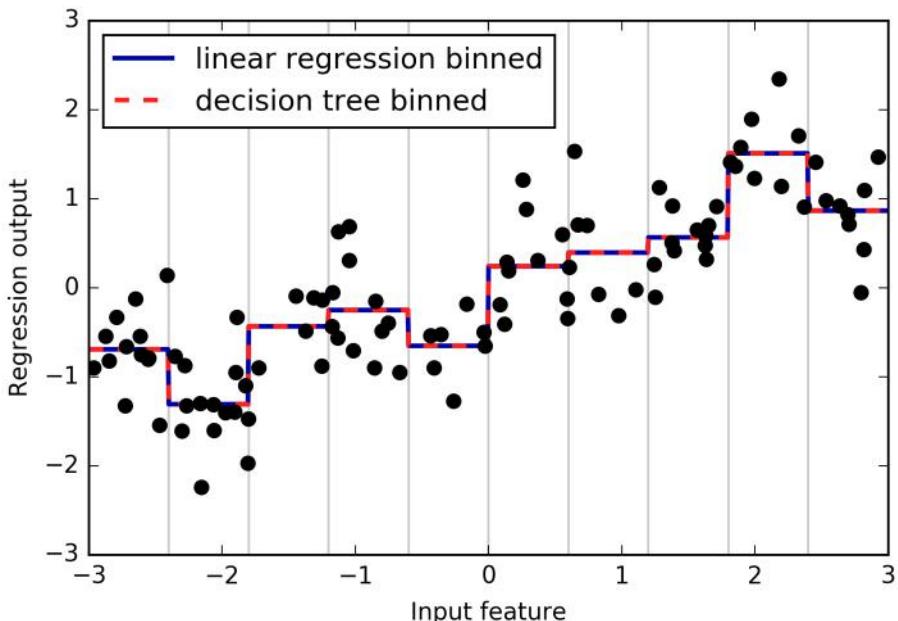


Figure 4-2. Comparing linear regression and decision tree regression on binned features

The dashed line and solid line are exactly on top of each other, meaning the linear regression model and the decision tree make exactly the same predictions. For each bin, they predict a constant value. As features are constant within each bin, any model must predict the same value for all points within a bin. Comparing what the models learned before binning the features and after, we see that the linear model became much more flexible, because it now has a different value for each bin, while the decision tree model got much less flexible. Binning features generally has no beneficial effect for tree-based models, as these models can learn to split up the data anywhere. In a sense, that means decision trees can learn whatever binning is most useful for predicting on this data. Additionally, decision trees look at multiple features at once, while binning is usually done on a per-feature basis. However, the linear model benefited greatly in expressiveness from the transformation of the data.

If there are good reasons to use a linear model for a particular dataset—say, because it is very large and high-dimensional, but some features have nonlinear relations with the output—binning can be a great way to increase modeling power.

Interactions and Polynomials

Another way to enrich a feature representation, particularly for linear models, is adding *interaction features* and *polynomial features* of the original data. This kind of feature engineering is often used in statistical modeling, but it's also common in many practical machine learning applications.

As a first example, look again at [Figure 4-2](#). The linear model learned a constant value for each bin in the wave dataset. We know, however, that linear models can learn not only offsets, but also slopes. One way to add a slope to the linear model on the binned data is to add the original feature (the x-axis in the plot) back in. This leads to an 11-dimensional dataset, as seen in [Figure 4-3](#):

In[17]:

```
X_combined = np.hstack([X, X_binned])
print(X_combined.shape)
```

Out[17]:

```
(100, 11)
```

In[18]:

```
reg = LinearRegression().fit(X_combined, y)

line_combined = np.hstack([line, line_binned])
plt.plot(line, reg.predict(line_combined), label='linear regression combined')

for bin in bins:
    plt.plot([bin, bin], [-3, 3], ':', c='k')
```

```

plt.legend(loc="best")
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.plot(X[:, 0], y, 'o', c='k')

```

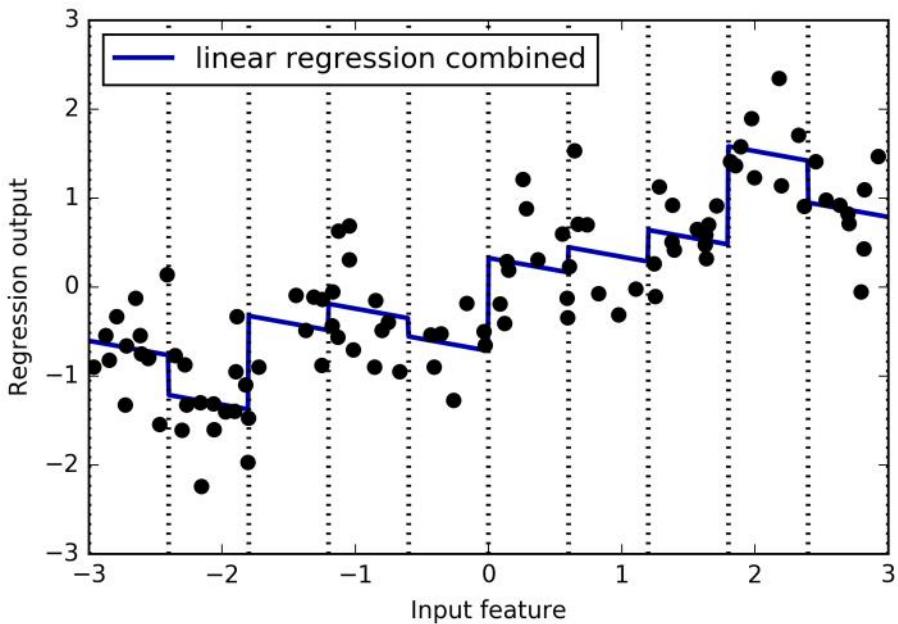


Figure 4-3. Linear regression using binned features and a single global slope

In this example, the model learned an offset for each bin, together with a slope. The learned slope is downward, and shared across all the bins—there is a single x-axis feature, which has a single slope. Because the slope is shared across all bins, it doesn’t seem to be very helpful. We would rather have a separate slope for each bin! We can achieve this by adding an interaction or product feature that indicates which bin a data point is in *and* where it lies on the x-axis. This feature is a product of the bin indicator and the original feature. Let’s create this dataset:

In[19]:

```

X_product = np.hstack([X_binned, X * X_binned])
print(X_product.shape)

```

Out[19]:

```
(100, 20)
```

The dataset now has 20 features: the indicators for which bin a data point is in, and a product of the original feature and the bin indicator. You can think of the product

feature as a separate copy of the x-axis feature for each bin. It is the original feature within the bin, and zero everywhere else. Figure 4-4 shows the result of the linear model on this new representation:

In[20]:

```
reg = LinearRegression().fit(X_product, y)

line_product = np.hstack([line_binned, line * line_binned])
plt.plot(line, reg.predict(line_product), label='linear regression product')

for bin in bins:
    plt.plot([bin, bin], [-3, 3], ':', c='k')

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.legend(loc="best")
```

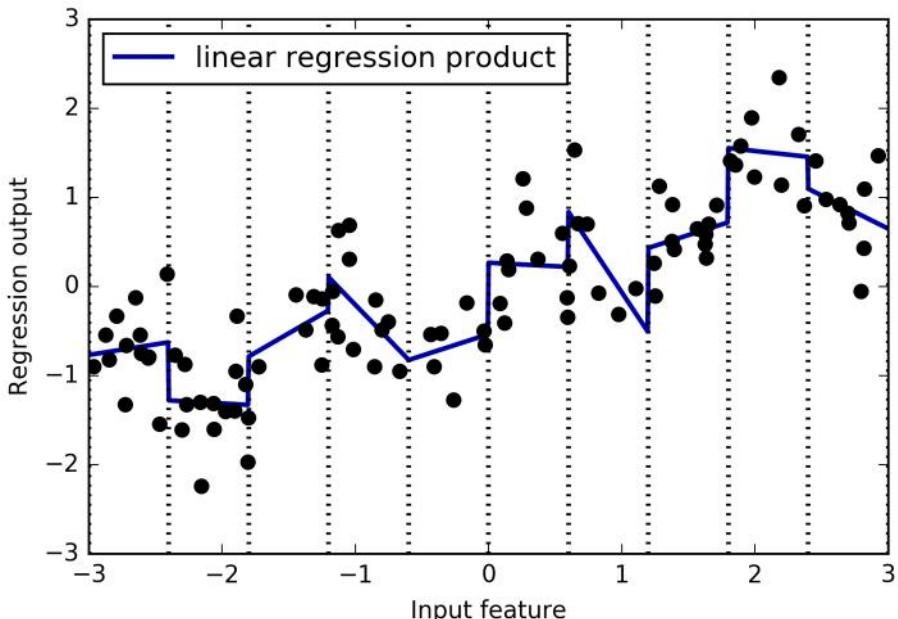


Figure 4-4. Linear regression with a separate slope per bin

As you can see, now each bin has its own offset and slope in this model.

Using binning is one way to expand a continuous feature. Another one is to use *polynomials* of the original features. For a given feature x , we might want to consider $x^{**} 2, x^{**} 3, x^{**} 4$, and so on. This is implemented in `PolynomialFeatures` in the `preprocessing` module:

In[21]:

```
from sklearn.preprocessing import PolynomialFeatures

# include polynomials up to x ** 10:
# the default "include_bias=True" adds a feature that's constantly 1
poly = PolynomialFeatures(degree=10, include_bias=False)
poly.fit(X)
X_poly = poly.transform(X)
```

Using a degree of 10 yields 10 features:

In[22]:

```
print("X_poly.shape: {}".format(X_poly.shape))
```

Out[22]:

```
X_poly.shape: (100, 10)
```

Let's compare the entries of `X_poly` to those of `X`:

In[23]:

```
print("Entries of X:\n{}".format(X[:5]))
print("Entries of X_poly:\n{}".format(X_poly[:5]))
```

Out[23]:

```
Entries of X:
[[-0.753]
 [ 2.704]
 [ 1.392]
 [ 0.592]
 [-2.064]]
Entries of X_poly:
[[ -0.753      0.567     -0.427      0.321     -0.242      0.182
   -0.137      0.103     -0.078      0.058]
 [ 2.704      7.313     19.777     53.482    144.632    391.125
  1057.714    2860.360   7735.232   20918.278]
 [ 1.392      1.938      2.697      3.754      5.226      7.274
   10.125     14.094     19.618     27.307]
 [ 0.592      0.350      0.207      0.123      0.073      0.043
   0.025      0.015      0.009      0.005]
 [-2.064      4.260     -8.791     18.144    -37.448      77.289
  -159.516    329.222   -679.478   1402.367]]
```

You can obtain the semantics of the features by calling the `get_feature_names` method, which provides the exponent for each feature:

In[24]:

```
print("Polynomial feature names:\n{}".format(poly.get_feature_names()))
```

Out[24]:

```
Polynomial feature names:  
['x0', 'x0^2', 'x0^3', 'x0^4', 'x0^5', 'x0^6', 'x0^7', 'x0^8', 'x0^9', 'x0^10']
```

You can see that the first column of X_{poly} corresponds exactly to X , while the other columns are the powers of the first entry. It's interesting to see how large some of the values can get. The second column has entries above 20,000, orders of magnitude different from the rest.

Using polynomial features together with a linear regression model yields the classical model of *polynomial regression* (see Figure 4-5):

In[26]:

```
reg = LinearRegression().fit(X_poly, y)  
  
line_poly = poly.transform(line)  
plt.plot(line, reg.predict(line_poly), label='polynomial linear regression')  
plt.plot(X[:, 0], y, 'o', c='k')  
plt.ylabel("Regression output")  
plt.xlabel("Input feature")  
plt.legend(loc="best")
```

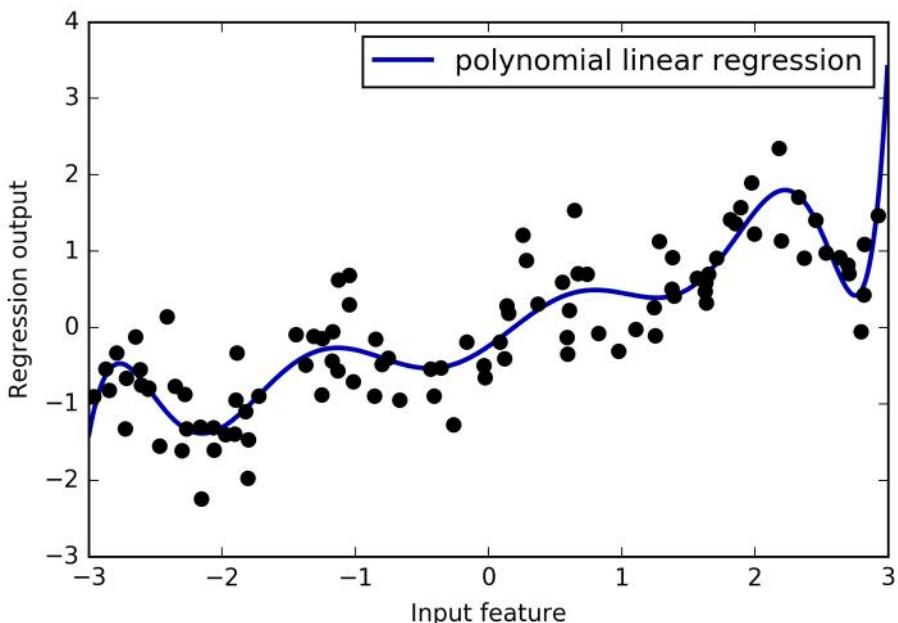


Figure 4-5. Linear regression with tenth-degree polynomial features

As you can see, polynomial features yield a very smooth fit on this one-dimensional data. However, polynomials of high degree tend to behave in extreme ways on the boundaries or in regions with little data.

As a comparison, here is a kernel SVM model learned on the original data, without any transformation (see [Figure 4-6](#)):

In[26]:

```
from sklearn.svm import SVR

for gamma in [1, 10]:
    svr = SVR(gamma=gamma).fit(X, y)
    plt.plot(line, svr.predict(line), label='SVR gamma={}'.format(gamma))

plt.plot(X[:, 0], y, 'o', c='k')
plt.ylabel("Regression output")
plt.xlabel("Input feature")
plt.legend(loc="best")
```

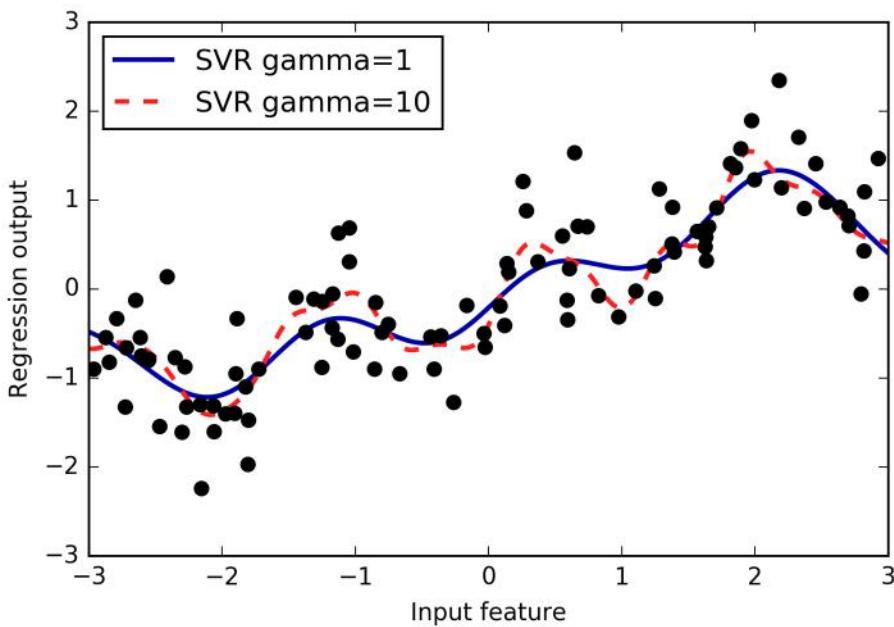


Figure 4-6. Comparison of different gamma parameters for an SVM with RBF kernel

Using a more complex model, a kernel SVM, we are able to learn a similarly complex prediction to the polynomial regression without an explicit transformation of the features.

As a more realistic application of interactions and polynomials, let's look again at the Boston Housing dataset. We already used polynomial features on this dataset in [Chapter 2](#). Now let's have a look at how these features were constructed, and at how much the polynomial features help. First we load the data, and rescale it to be between 0 and 1 using `MinMaxScaler`:

In[27]:

```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split
    (boston.data, boston.target, random_state=0)

# rescale data
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Now, we extract polynomial features and interactions up to a degree of 2:

In[28]:

```
poly = PolynomialFeatures(degree=2).fit(X_train_scaled)
X_train_poly = poly.transform(X_train_scaled)
X_test_poly = poly.transform(X_test_scaled)
print("X_train.shape: {}".format(X_train.shape))
print("X_train_poly.shape: {}".format(X_train_poly.shape))
```

Out[28]:

```
X_train.shape: (379, 13)
X_train_poly.shape: (379, 105)
```

The data originally had 13 features, which were expanded into 105 interaction features. These new features represent all possible interactions between two different original features, as well as the square of each original feature. `degree=2` here means that we look at all features that are the product of up to two original features. The exact correspondence between input and output features can be found using the `get_feature_names` method:

In[29]:

```
print("Polynomial feature names:\n{}".format(poly.get_feature_names()))
```

Out[29]:

```
Polynomial feature names:
['1', 'x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9', 'x10',
 'x11', 'x12', 'x0^2', 'x0 x1', 'x0 x2', 'x0 x3', 'x0 x4', 'x0 x5', 'x0 x6',
 'x0 x7', 'x0 x8', 'x0 x9', 'x0 x10', 'x0 x11', 'x0 x12', 'x1^2', 'x1 x2',
```

```
'x1 x3', 'x1 x4', 'x1 x5', 'x1 x6', 'x1 x7', 'x1 x8', 'x1 x9', 'x1 x10',
'x1 x11', 'x1 x12', 'x2^2', 'x2 x3', 'x2 x4', 'x2 x5', 'x2 x6', 'x2 x7',
'x2 x8', 'x2 x9', 'x2 x10', 'x2 x11', 'x2 x12', 'x3^2', 'x3 x4', 'x3 x5',
'x3 x6', 'x3 x7', 'x3 x8', 'x3 x9', 'x3 x10', 'x3 x11', 'x3 x12', 'x4^2',
'x4 x5', 'x4 x6', 'x4 x7', 'x4 x8', 'x4 x9', 'x4 x10', 'x4 x11', 'x4 x12',
'x5^2', 'x5 x6', 'x5 x7', 'x5 x8', 'x5 x9', 'x5 x10', 'x5 x11', 'x5 x12',
'x6^2', 'x6 x7', 'x6 x8', 'x6 x9', 'x6 x10', 'x6 x11', 'x6 x12', 'x7^2',
'x7 x8', 'x7 x9', 'x7 x10', 'x7 x11', 'x7 x12', 'x8^2', 'x8 x9', 'x8 x10',
'x8 x11', 'x8 x12', 'x9^2', 'x9 x10', 'x9 x11', 'x9 x12', 'x10^2', 'x10 x11',
'x10 x12', 'x11^2', 'x11 x12', 'x12^2']
```

The first new feature is a constant feature, called "1" here. The next 13 features are the original features (called "x0" to "x12"). Then follows the first feature squared ("x0^2") and combinations of the first and the other features.

Let's compare the performance using Ridge on the data with and without interactions:

In[30]:

```
from sklearn.linear_model import Ridge
ridge = Ridge().fit(X_train_scaled, y_train)
print("Score without interactions: {:.3f}".format(
    ridge.score(X_test_scaled, y_test)))
ridge = Ridge().fit(X_train_poly, y_train)
print("Score with interactions: {:.3f}".format(
    ridge.score(X_test_poly, y_test)))
```

Out[30]:

```
Score without interactions: 0.621
Score with interactions: 0.753
```

Clearly, the interactions and polynomial features gave us a good boost in performance when using Ridge. When using a more complex model like a random forest, the story is a bit different, though:

In[31]:

```
from sklearn.ensemble import RandomForestRegressor
rf = RandomForestRegressor(n_estimators=100).fit(X_train_scaled, y_train)
print("Score without interactions: {:.3f}".format(
    rf.score(X_test_scaled, y_test)))
rf = RandomForestRegressor(n_estimators=100).fit(X_train_poly, y_train)
print("Score with interactions: {:.3f}".format(rf.score(X_test_poly, y_test)))
```

Out[31]:

```
Score without interactions: 0.799
Score with interactions: 0.763
```

You can see that even without additional features, the random forest beats the performance of Ridge. Adding interactions and polynomials actually decreases performance slightly.

Univariate Nonlinear Transformations

We just saw that adding squared or cubed features can help linear models for regression. There are other transformations that often prove useful for transforming certain features: in particular, applying mathematical functions like `log`, `exp`, or `sin`. While tree-based models only care about the ordering of the features, linear models and neural networks are very tied to the scale and distribution of each feature, and if there is a nonlinear relation between the feature and the target, that becomes hard to model—particularly in regression. The functions `log` and `exp` can help by adjusting the relative scales in the data so that they can be captured better by a linear model or neural network. We saw an application of that in [Chapter 2](#) with the memory price data. The `sin` and `cos` functions can come in handy when dealing with data that encodes periodic patterns.

Most models work best when each feature (and in regression also the target) is loosely Gaussian distributed—that is, a histogram of each feature should have something resembling the familiar “bell curve” shape. Using transformations like `log` and `exp` is a hacky but simple and efficient way to achieve this. A particularly common case when such a transformation can be helpful is when dealing with integer count data. By count data, we mean features like “how often did user A log in?” Counts are never negative, and often follow particular statistical patterns. We are using a synthetic dataset of counts here that has properties similar to those you can find in the wild. The features are all integer-valued, while the response is continuous:

In[32]:

```
rnd = np.random.RandomState(0)
X_org = rnd.normal(size=(1000, 3))
w = rnd.normal(size=3)

X = rnd.poisson(10 * np.exp(X_org))
y = np.dot(X_org, w)
```

Let’s look at the first 10 entries of the first feature. All are integer values and positive, but apart from that it’s hard to make out a particular pattern.

If we count the appearance of each value, the distribution of values becomes clearer:

In[33]:

```
print("Number of feature appearances:\n{}".format(np.bincount(X[:, 0])))
```

Out[33]:

```
Number of feature appearances:  
[28 38 68 48 61 59 45 56 37 40 35 34 36 26 23 26 27 21 23 23 18 21 10 9 17  
 9 7 14 12 7 3 8 4 5 5 3 4 2 4 1 1 3 2 5 3 8 2 5 2 1  
 2 3 3 2 2 3 3 0 1 2 1 0 0 3 1 0 0 0 1 3 0 1 0 2 0  
 1 1 0 0 0 0 1 0 0 2 2 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0  
 0 0 1 0 0 0 0 1 1 0 0 1 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 1 0 0 0 0 0 0 0]
```

The value 2 seems to be the most common, with 62 appearances (bincount always starts at 0), and the counts for higher values fall quickly. However, there are some very high values, like 134 appearing twice. We visualize the counts in Figure 4-7:

In[34]:

```
bins = np.bincount(X[:, 0])  
plt.bar(range(len(bins)), bins, color='w')  
plt.ylabel("Number of appearances")  
plt.xlabel("Value")
```

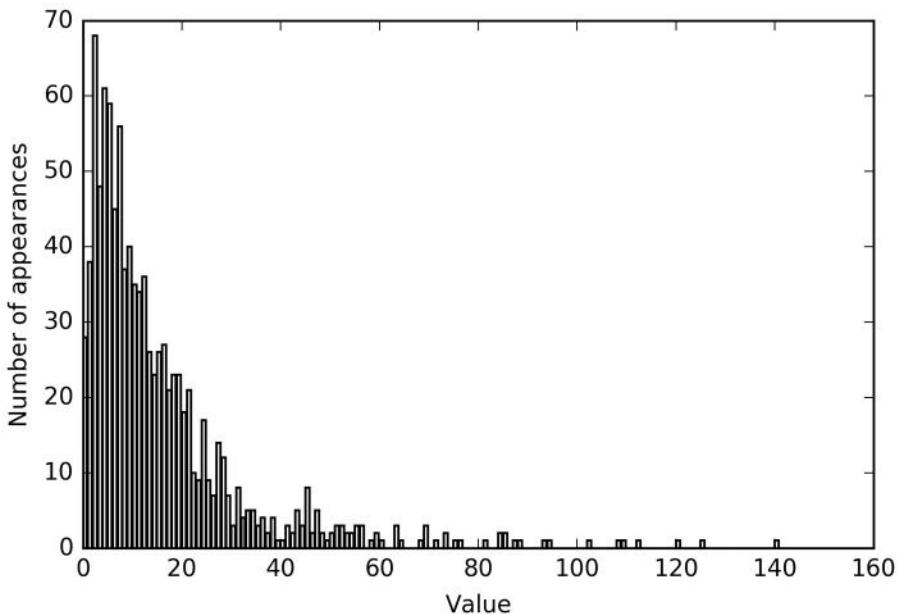


Figure 4-7. Histogram of feature values for $X[0]$

Features $X[:, 1]$ and $X[:, 2]$ have similar properties. This kind of distribution of values (many small ones and a few very large ones) is very common in practice.¹ However, it is something most linear models can't handle very well. Let's try to fit a ridge regression to this model:

In[35]:

```
from sklearn.linear_model import Ridge
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
score = Ridge().fit(X_train, y_train).score(X_test, y_test)
print("Test score: {:.3f}".format(score))
```

Out[35]:

```
Test score: 0.622
```

As you can see from the relatively low R^2 score, Ridge was not able to really capture the relationship between X and y . Applying a logarithmic transformation can help, though. Because the value 0 appears in the data (and the logarithm is not defined at 0), we can't actually just apply \log , but we have to compute $\log(X + 1)$:

In[36]:

```
X_train_log = np.log(X_train + 1)
X_test_log = np.log(X_test + 1)
```

After the transformation, the distribution of the data is less asymmetrical and doesn't have very large outliers anymore (see Figure 4-8):

In[37]:

```
plt.hist(np.log(X_train_log[:, 0] + 1), bins=25, color='gray')
plt.ylabel("Number of appearances")
plt.xlabel("Value")
```

¹ This is a Poisson distribution, which is quite fundamental to count data.

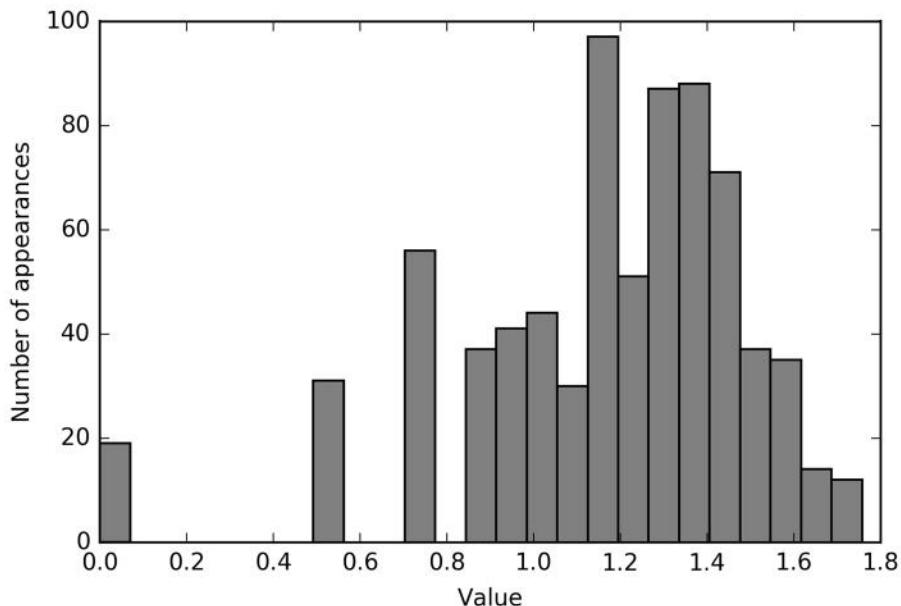


Figure 4-8. Histogram of feature values for $X[0]$ after logarithmic transformation

Building a ridge model on the new data provides a much better fit:

In[38]:

```
score = Ridge().fit(X_train_log, y_train).score(X_test_log, y_test)
print("Test score: {:.3f}".format(score))
```

Out[38]:

```
Test score: 0.875
```

Finding the transformation that works best for each combination of dataset and model is somewhat of an art. In this example, all the features had the same properties. This is rarely the case in practice, and usually only a subset of the features should be transformed, or sometimes each feature needs to be transformed in a different way. As we mentioned earlier, these kinds of transformations are irrelevant for tree-based models but might be essential for linear models. Sometimes it is also a good idea to transform the target variable y in regression. Trying to predict counts (say, number of orders) is a fairly common task, and using the $\log(y + 1)$ transformation often helps.²

² This is a very crude approximation of using Poisson regression, which would be the proper solution from a probabilistic standpoint.

As you saw in the previous examples, binning, polynomials, and interactions can have a huge influence on how models perform on a given dataset. This is particularly true for less complex models like linear models and naive Bayes models. Tree-based models, on the other hand, are often able to discover important interactions themselves, and don't require transforming the data explicitly most of the time. Other models, like SVMs, nearest neighbors, and neural networks, might sometimes benefit from using binning, interactions, or polynomials, but the implications there are usually much less clear than in the case of linear models.

Automatic Feature Selection

With so many ways to create new features, you might get tempted to increase the dimensionality of the data way beyond the number of original features. However, adding more features makes all models more complex, and so increases the chance of overfitting. When adding new features, or with high-dimensional datasets in general, it can be a good idea to reduce the number of features to only the most useful ones, and discard the rest. This can lead to simpler models that generalize better. But how can you know how good each feature is? There are three basic strategies: *univariate statistics*, *model-based selection*, and *iterative selection*. We will discuss all three of them in detail. All of these methods are supervised methods, meaning they need the target for fitting the model. This means we need to split the data into training and test sets, and fit the feature selection only on the training part of the data.

Univariate Statistics

In univariate statistics, we compute whether there is a statistically significant relationship between each feature and the target. Then the features that are related with the highest confidence are selected. In the case of classification, this is also known as *analysis of variance* (ANOVA). A key property of these tests is that they are *univariate*, meaning that they only consider each feature individually. Consequently, a feature will be discarded if it is only informative when combined with another feature. Univariate tests are often very fast to compute, and don't require building a model. On the other hand, they are completely independent of the model that you might want to apply after the feature selection.

To use univariate feature selection in `scikit-learn`, you need to choose a test, usually either `f_classif` (the default) for classification or `f_regression` for regression, and a method to discard features based on the p -values determined in the test. All methods for discarding parameters use a threshold to discard all features with too high a p -value (which means they are unlikely to be related to the target). The methods differ in how they compute this threshold, with the simplest ones being `SelectKBest`, which selects a fixed number k of features, and `SelectPercentile`, which selects a fixed percentage of features. Let's apply the feature selection for classification on the

cancer dataset. To make the task a bit harder, we'll add some noninformative noise features to the data. We expect the feature selection to be able to identify the features that are noninformative and remove them:

In[39]:

```
from sklearn.datasets import load_breast_cancer
from sklearn.feature_selection import SelectPercentile
from sklearn.model_selection import train_test_split

cancer = load_breast_cancer()

# get deterministic random numbers
rng = np.random.RandomState(42)
noise = rng.normal(size=(len(cancer.data), 50))
# add noise features to the data
# the first 30 features are from the dataset, the next 50 are noise
X_w_noise = np.hstack([cancer.data, noise])

X_train, X_test, y_train, y_test = train_test_split(
    X_w_noise, cancer.target, random_state=0, test_size=.5)
# use f_classif (the default) and SelectPercentile to select 50% of features
select = SelectPercentile(percentile=50)
select.fit(X_train, y_train)
# transform training set
X_train_selected = select.transform(X_train)

print("X_train.shape: {}".format(X_train.shape))
print("X_train_selected.shape: {}".format(X_train_selected.shape))
```

Out[39]:

```
X_train.shape: (284, 80)
X_train_selected.shape: (284, 40)
```

As you can see, the number of features was reduced from 80 to 40 (50 percent of the original number of features). We can find out which features have been selected using the `get_support` method, which returns a Boolean mask of the selected features (visualized in Figure 4-9):

In[40]:

```
mask = select.get_support()
print(mask)
# visualize the mask -- black is True, white is False
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Sample index")
```

Out[40]:

```
[ True  True  True  True  True  True  True  True  True  False  True  False
 True  True  True  True  True  True  False  False  True  True  True  True
 True  True  True  True  True  True  False  False  False  True  False  True]
```

```

False False True False False False True False False True False
False True False True False False False False True False
True False False False False True False True False False False
True True False True False False False

```

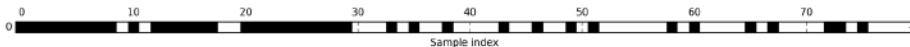


Figure 4-9. Features selected by SelectPercentile

As you can see from the visualization of the mask, most of the selected features are the original features, and most of the noise features were removed. However, the recovery of the original features is not perfect. Let's compare the performance of logistic regression on all features against the performance using only the selected features:

In[41]:

```

from sklearn.linear_model import LogisticRegression

# transform test data
X_test_selected = select.transform(X_test)

lr = LogisticRegression()
lr.fit(X_train, y_train)
print("Score with all features: {:.3f}".format(lr.score(X_test, y_test)))
lr.fit(X_train_selected, y_train)
print("Score with only selected features: {:.3f}".format(
    lr.score(X_test_selected, y_test)))

```

Out[41]:

```

Score with all features: 0.930
Score with only selected features: 0.940

```

In this case, removing the noise features improved performance, even though some of the original features were lost. This was a very simple synthetic example, and outcomes on real data are usually mixed. Univariate feature selection can still be very helpful, though, if there is such a large number of features that building a model on them is infeasible, or if you suspect that many features are completely uninformative.

Model-Based Feature Selection

Model-based feature selection uses a supervised machine learning model to judge the importance of each feature, and keeps only the most important ones. The supervised model that is used for feature selection doesn't need to be the same model that is used for the final supervised modeling. The feature selection model needs to provide some measure of importance for each feature, so that they can be ranked by this measure. Decision trees and decision tree-based models provide a `feature_importances_`

attribute, which directly encodes the importance of each feature. Linear models have coefficients, which can also be used to capture feature importances by considering the absolute values. As we saw in [Chapter 2](#), linear models with L1 penalty learn sparse coefficients, which only use a small subset of features. This can be viewed as a form of feature selection for the model itself, but can also be used as a preprocessing step to select features for another model. In contrast to univariate selection, model-based selection considers all features at once, and so can capture interactions (if the model can capture them). To use model-based feature selection, we need to use the `SelectFromModel` transformer:

In[42]:

```
from sklearn.feature_selection import SelectFromModel
from sklearn.ensemble import RandomForestClassifier
select = SelectFromModel(
    RandomForestClassifier(n_estimators=100, random_state=42),
    threshold="median")
```

The `SelectFromModel` class selects all features that have an importance measure of the feature (as provided by the supervised model) greater than the provided threshold. To get a comparable result to what we got with univariate feature selection, we used the median as a threshold, so that half of the features will be selected. We use a random forest classifier with 100 trees to compute the feature importances. This is a quite complex model and much more powerful than using univariate tests. Now let's actually fit the model:

In[43]:

```
select.fit(X_train, y_train)
X_train_l1 = select.transform(X_train)
print("X_train.shape: {}".format(X_train.shape))
print("X_train_l1.shape: {}".format(X_train_l1.shape))
```

Out[43]:

```
X_train.shape: (284, 80)
X_train_l1.shape: (284, 40)
```

Again, we can have a look at the features that were selected ([Figure 4-10](#)):

In[44]:

```
mask = select.get_support()
# visualize the mask -- black is True, white is False
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Sample index")
```



Figure 4-10. Features selected by `SelectFromModel` using the `RandomForestClassifier`

This time, all but two of the original features were selected. Because we specified to select 40 features, some of the noise features are also selected. Let's take a look at the performance:

In[45]:

```
X_test_l1 = select.transform(X_test)
score = LogisticRegression().fit(X_train_l1, y_train).score(X_test_l1, y_test)
print("Test score: {:.3f}".format(score))
```

Out[45]:

```
Test score: 0.951
```

With the better feature selection, we also gained some improvements here.

Iterative Feature Selection

In univariate testing we used no model, while in model-based selection we used a single model to select features. In iterative feature selection, a series of models are built, with varying numbers of features. There are two basic methods: starting with no features and adding features one by one until some stopping criterion is reached, or starting with all features and removing features one by one until some stopping criterion is reached. Because a series of models are built, these methods are much more computationally expensive than the methods we discussed previously. One particular method of this kind is *recursive feature elimination* (RFE), which starts with all features, builds a model, and discards the least important feature according to the model. Then a new model is built using all but the discarded feature, and so on until only a prespecified number of features are left. For this to work, the model used for selection needs to provide some way to determine feature importance, as was the case for the model-based selection. Here, we use the same random forest model that we used earlier, and get the results shown in Figure 4-11:

In[46]:

```
from sklearn.feature_selection import RFE
select = RFE(RandomForestClassifier(n_estimators=100, random_state=42),
              n_features_to_select=40)

select.fit(X_train, y_train)
# visualize the selected features:
mask = select.get_support()
plt.matshow(mask.reshape(1, -1), cmap='gray_r')
plt.xlabel("Sample index")
```

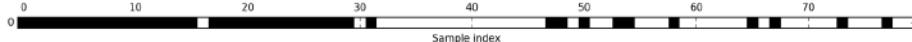


Figure 4-11. Features selected by recursive feature elimination with the random forest classifier model

The feature selection got better compared to the univariate and model-based selection, but one feature was still missed. Running this code also takes significantly longer than that for the model-based selection, because a random forest model is trained 40 times, once for each feature that is dropped. Let's test the accuracy of the logistic regression model when using RFE for feature selection:

In[47]:

```
X_train_rfe= select.transform(X_train)
X_test_rfe= select.transform(X_test)

score = LogisticRegression().fit(X_train_rfe, y_train).score(X_test_rfe, y_test)
print("Test score: {:.3f}".format(score))
```

Out[47]:

```
Test score: 0.951
```

We can also use the model used inside the RFE to make predictions. This uses only the feature set that was selected:

In[48]:

```
print("Test score: {:.3f}".format(select.score(X_test, y_test)))
```

Out[48]:

```
Test score: 0.951
```

Here, the performance of the random forest used inside the RFE is the same as that achieved by training a logistic regression model on top of the selected features. In other words, once we've selected the right features, the linear model performs as well as the random forest.

If you are unsure when selecting what to use as input to your machine learning algorithms, automatic feature selection can be quite helpful. It is also great for reducing the amount of features needed—for example, to speed up prediction or to allow for more interpretable models. In most real-world cases, applying feature selection is unlikely to provide large gains in performance. However, it is still a valuable tool in the toolbox of the feature engineer.

Utilizing Expert Knowledge

Feature engineering is often an important place to use *expert knowledge* for a particular application. While the purpose of machine learning in many cases is to avoid having to create a set of expert-designed rules, that doesn't mean that prior knowledge of the application or domain should be discarded. Often, domain experts can help in identifying useful features that are much more informative than the initial representation of the data. Imagine you work for a travel agency and want to predict flight prices. Let's say you have a record of prices together with dates, airlines, start locations, and destinations. A machine learning model might be able to build a decent model from that. Some important factors in flight prices, however, cannot be learned. For example, flights are usually more expensive during peak vacation months and around holidays. While the dates of some holidays (like Christmas) are fixed, and their effect can therefore be learned from the date, others might depend on the phases of the moon (like Hanukkah and Easter) or be set by authorities (like school holidays). These events cannot be learned from the data if each flight is only recorded using the (Gregorian) date. However, it is easy to add a feature that encodes whether a flight was on, preceding, or following a public or school holiday. In this way, prior knowledge about the nature of the task can be encoded in the features to aid a machine learning algorithm. Adding a feature does not force a machine learning algorithm to use it, and even if the holiday information turns out to be noninformative for flight prices, augmenting the data with this information doesn't hurt.

We'll now look at one particular case of using expert knowledge—though in this case it might be more rightfully called "common sense." The task is predicting bicycle rentals in front of Andreas's house.

In New York, Citi Bike operates a network of bicycle rental stations with a subscription system. The stations are all over the city and provide a convenient way to get around. Bike rental data is made public in [an anonymized form](#) and has been analyzed in various ways. The task we want to solve is to predict for a given time and day how many people will rent a bike in front of Andreas's house—so he knows if any bikes will be left for him.

We first load the data for August 2015 for this particular station as a pandas Data Frame. We resample the data into three-hour intervals to obtain the main trends for each day:

In[49]:

```
citibike = mglearn.datasets.load_citibike()
```

In[50]:

```
print("Citi Bike data:\n{}".format(citibike.head()))
```

Out[50]:

```
Citi Bike data:  
starttime  
2015-08-01 00:00:00      3.0  
2015-08-01 03:00:00      0.0  
2015-08-01 06:00:00      9.0  
2015-08-01 09:00:00     41.0  
2015-08-01 12:00:00     39.0  
Freq: 3H, Name: one, dtype: float64
```

The following example shows a visualization of the rental frequencies for the whole month (Figure 4-12):

In[51]:

```
plt.figure(figsize=(10, 3))  
xticks = pd.date_range(start=citibike.index.min(), end=citibike.index.max(),  
                      freq='D')  
plt.xticks(xticks, xticks.strftime("%a %m-%d"), rotation=90, ha="left")  
plt.plot(citibike, linewidth=1)  
plt.xlabel("Date")  
plt.ylabel("Rentals")
```

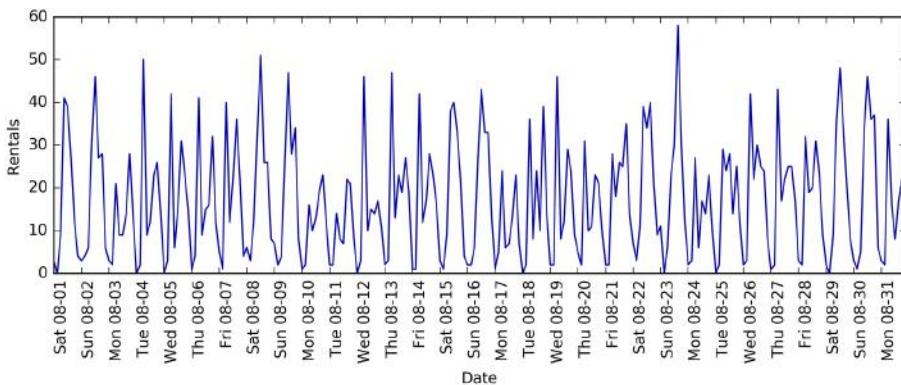


Figure 4-12. Number of bike rentals over time for a selected Citi Bike station

Looking at the data, we can clearly distinguish day and night for each 24-hour interval. The patterns for weekdays and weekends also seem to be quite different. When evaluating a prediction task on a time series like this, we usually want to *learn from the past and predict for the future*. This means when doing a split into a training and a test set, we want to use all the data up to a certain date as the training set and all the data past that date as the test set. This is how we would usually use time series prediction: given everything that we know about rentals in the past, what do we think will

happen tomorrow? We will use the first 184 data points, corresponding to the first 23 days, as our training set, and the remaining 64 data points, corresponding to the remaining 8 days, as our test set.

The only feature that we are using in our prediction task is the date and time when a particular number of rentals occurred. So, the input feature is the date and time—say, 2015-08-01 00:00:00—and the output is the number of rentals in the following three hours (three in this case, according to our DataFrame).

A (surprisingly) common way that dates are stored on computers is using POSIX time, which is the number of seconds since January 1970 00:00:00 (aka the beginning of Unix time). As a first try, we can use this single integer feature as our data representation:

In[52]:

```
# extract the target values (number of rentals)
y = citibike.values
# convert the time to POSIX time using "%s"
X = citibike.index.strftime("%s").astype("int").reshape(-1, 1)
```

We first define a function to split the data into training and test sets, build the model, and visualize the result:

In[54]:

```
# use the first 184 data points for training, and the rest for testing
n_train = 184

# function to evaluate and plot a regressor on a given feature set
def eval_on_features(features, target, regressor):
    # split the given features into a training and a test set
    X_train, X_test = features[:n_train], features[n_train:]
    # also split the target array
    y_train, y_test = target[:n_train], target[n_train:]
    regressor.fit(X_train, y_train)
    print("Test-set R^2: {:.2f}".format(regressor.score(X_test, y_test)))
    y_pred = regressor.predict(X_test)
    y_pred_train = regressor.predict(X_train)
    plt.figure(figsize=(10, 3))

    plt.xticks(range(0, len(X), 8), xticks.strftime("%a %m-%d"), rotation=90,
               ha="left")

    plt.plot(range(n_train), y_train, label="train")
    plt.plot(range(n_train, len(y_test) + n_train), y_test, '--', label="test")
    plt.plot(range(n_train), y_pred_train, '--', label="prediction train")

    plt.plot(range(n_train, len(y_test) + n_train), y_pred, '--',
            label="prediction test")
    plt.legend(loc=(1.01, 0))
    plt.xlabel("Date")
    plt.ylabel("Rentals")
```

We saw earlier that random forests require very little preprocessing of the data, which makes this seem like a good model to start with. We use the POSIX time feature X and pass a random forest regressor to our eval_on_features function. [Figure 4-13](#) shows the result:

In[55]:

```
from sklearn.ensemble import RandomForestRegressor
regressor = RandomForestRegressor(n_estimators=100, random_state=0)
plt.figure()
eval_on_features(X, y, regressor)
```

Out[55]:

Test-set R²: -0.04

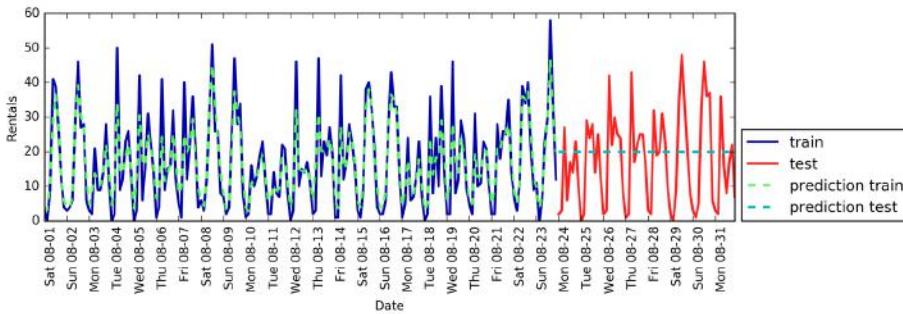


Figure 4-13. Predictions made by a random forest using only the POSIX time

The predictions on the training set are quite good, as is usual for random forests. However, for the test set, a constant line is predicted. The R^2 is -0.03 , which means that we learned nothing. What happened?

The problem lies in the combination of our feature and the random forest. The value of the POSIX time feature for the test set is outside of the range of the feature values in the training set: the points in the test set have timestamps that are later than all the points in the training set. Trees, and therefore random forests, cannot *extrapolate* to feature ranges outside the training set. The result is that the model simply predicts the target value of the closest point in the training set—which is the last time it observed any data.

Clearly we can do better than this. This is where our “expert knowledge” comes in. From looking at the rental figures in the training data, two factors seem to be very important: the time of day and the day of the week. So, let’s add these two features. We can’t really learn anything from the POSIX time, so we drop that feature. First, let’s use only the hour of the day. As [Figure 4-14](#) shows, now the predictions have the same pattern for each day of the week:

In[56]:

```
X_hour = citibike.index.hour.reshape(-1, 1)
eval_on_features(X_hour, y, regressor)
```

Out[56]:

Test-set R²: 0.60

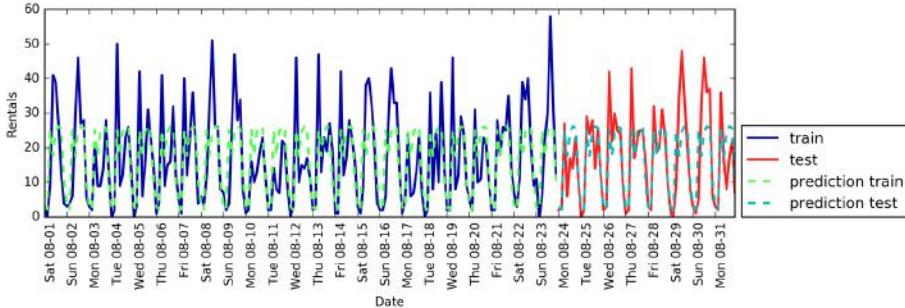


Figure 4-14. Predictions made by a random forest using only the hour of the day

The R^2 is already much better, but the predictions clearly miss the weekly pattern. Now let's also add the day of the week (see Figure 4-15):

In[57]:

```
X_hour_week = np.hstack([citibike.index.dayofweek.reshape(-1, 1),
                         citibike.index.hour.reshape(-1, 1)])
eval_on_features(X_hour_week, y, regressor)
```

Out[57]:

Test-set R²: 0.84

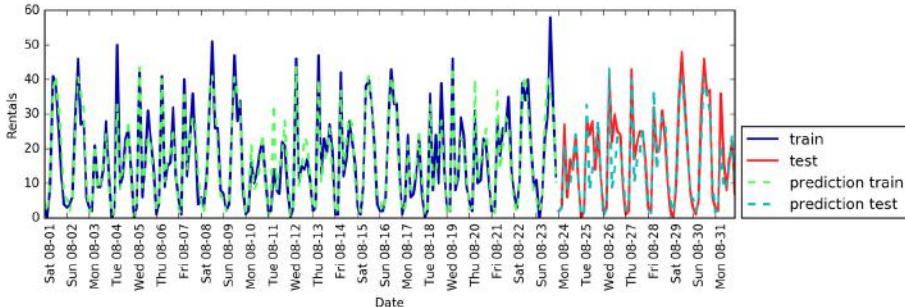


Figure 4-15. Predictions with a random forest using day of week and hour of day features

Now we have a model that captures the periodic behavior by considering the day of week and time of day. It has an R^2 of 0.84, and shows pretty good predictive performance. What this model likely is learning is the mean number of rentals for each combination of weekday and time of day from the first 23 days of August. This actually does not require a complex model like a random forest, so let's try with a simpler model, `LinearRegression` (see Figure 4-16):

In[58]:

```
from sklearn.linear_model import LinearRegression  
eval_on_features(X_hour_week, y, LinearRegression())
```

Out[58]:

Test-set R²: 0.13

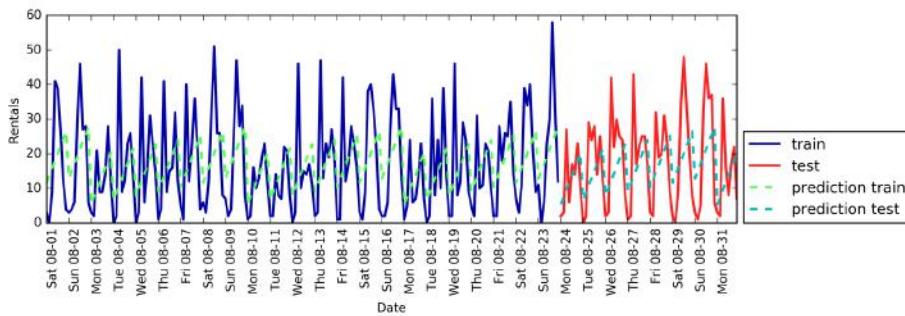


Figure 4-16. Predictions made by linear regression using day of week and hour of day as features

`LinearRegression` works much worse, and the periodic pattern looks odd. The reason for this is that we encoded day of week and time of day using integers, which are interpreted as categorical variables. Therefore, the linear model can only learn a linear function of the time of day—and it learned that later in the day, there are more rentals. However, the patterns are much more complex than that. We can capture this by interpreting the integers as categorical variables, by transforming them using `OneHotEncoder` (see Figure 4-17):

In[59]:

```
enc = OneHotEncoder()  
X_hour_week_onehot = enc.fit_transform(X_hour_week).toarray()
```

In[60]:

```
eval_on_features(X_hour_week_onehot, y, Ridge())
```

Out[60]:

```
Test-set R^2: 0.62
```

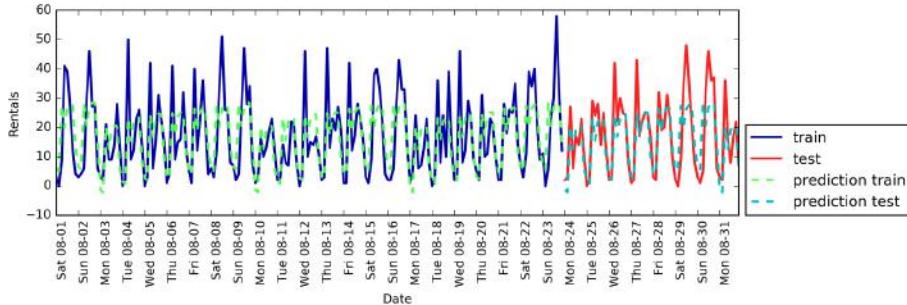


Figure 4-17. Predictions made by linear regression using a one-hot encoding of hour of day and day of week

This gives us a much better match than the continuous feature encoding. Now the linear model learns one coefficient for each day of the week, and one coefficient for each time of the day. That means that the “time of day” pattern is shared over all days of the week, though.

Using interaction features, we can allow the model to learn one coefficient for each combination of day and time of day (see Figure 4-18):

In[61]:

```
poly_transformer = PolynomialFeatures(degree=2, interaction_only=True,
                                      include_bias=False)
X_hour_week_onehot_poly = poly_transformer.fit_transform(X_hour_week_onehot)
lr = Ridge()
eval_on_features(X_hour_week_onehot_poly, y, lr)
```

Out[61]:

```
Test-set R^2: 0.85
```

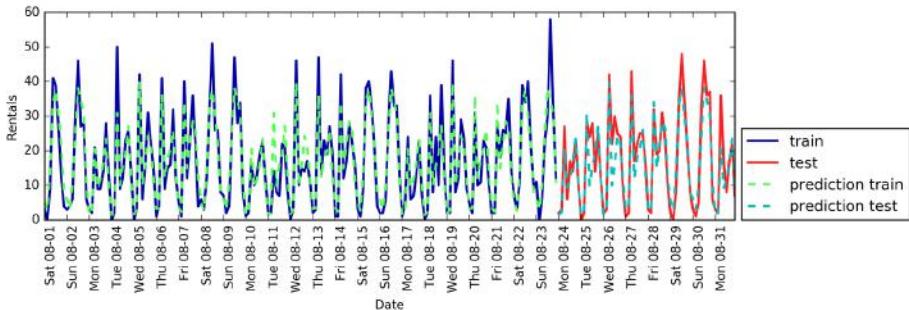


Figure 4-18. Predictions made by linear regression using a product of the day of week and hour of day features

This transformation finally yields a model that performs similarly well to the random forest. A big benefit of this model is that it is very clear what is learned: one coefficient for each day and time. We can simply plot the coefficients learned by the model, something that would not be possible for the random forest.

First, we create feature names for the hour and day features:

In[62]:

```
hour = ["%02d:00" % i for i in range(0, 24, 3)]
day = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
features = day + hour
```

Then we name all the interaction features extracted by `PolynomialFeatures`, using the `get_feature_names` method, and keep only the features with nonzero coefficients:

In[63]:

```
features_poly = poly_transformer.get_feature_names(features)
features_nonzero = np.array(features_poly)[lr.coef_ != 0]
coef_nonzero = lr.coef_[lr.coef_ != 0]
```

Now we can visualize the coefficients learned by the linear model, as seen in Figure 4-19:

In[64]:

```
plt.figure(figsize=(15, 2))
plt.plot(coef_nonzero, 'o')
plt.xticks(np.arange(len(coef_nonzero)), features_nonzero, rotation=90)
plt.xlabel("Feature magnitude")
plt.ylabel("Feature")
```

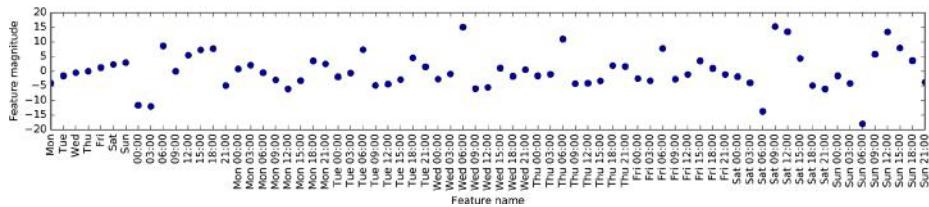


Figure 4-19. Coefficients of the linear regression model using a product of hour and day

Summary and Outlook

In this chapter, we discussed how to deal with different data types (in particular, with categorical variables). We emphasized the importance of representing data in a way that is suitable for the machine learning algorithm—for example, by one-hot-encoding categorical variables. We also discussed the importance of engineering new features, and the possibility of utilizing expert knowledge in creating derived features from your data. In particular, linear models might benefit greatly from generating new features via binning and adding polynomials and interactions, while more complex, nonlinear models like random forests and SVMs might be able to learn more complex tasks without explicitly expanding the feature space. In practice, the features that are used (and the match between features and method) is often the most important piece in making a machine learning approach work well.

Now that you have a good idea of how to represent your data in an appropriate way and which algorithm to use for which task, the next chapter will focus on evaluating the performance of machine learning models and selecting the right parameter settings.

Model Evaluation and Improvement

Having discussed the fundamentals of supervised and unsupervised learning, and having explored a variety of machine learning algorithms, we will now dive more deeply into evaluating models and selecting parameters.

We will focus on the supervised methods, regression and classification, as evaluating and selecting models in unsupervised learning is often a very qualitative process (as we saw in [Chapter 3](#)).

To evaluate our supervised models, so far we have split our dataset into a training set and a test set using the `train_test_split` function, built a model on the training set by calling the `fit` method, and evaluated it on the test set using the `score` method, which for classification computes the fraction of correctly classified samples. Here's an example of that process:

In[2]:

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

# create a synthetic dataset
X, y = make_blobs(random_state=0)
# split data and labels into a training and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
# instantiate a model and fit it to the training set
logreg = LogisticRegression().fit(X_train, y_train)
# evaluate the model on the test set
print("Test set score: {:.2f}".format(logreg.score(X_test, y_test)))
```

Out[2]:

```
Test set score: 0.88
```

Remember, the reason we split our data into training and test sets is that we are interested in measuring how well our model *generalizes* to new, previously unseen data. We are not interested in how well our model fit the training set, but rather in how well it can make predictions for data that was not observed during training.

In this chapter, we will expand on two aspects of this evaluation. We will first introduce cross-validation, a more robust way to assess generalization performance, and discuss methods to evaluate classification and regression performance that go beyond the default measures of accuracy and R^2 provided by the `score` method.

We will also discuss grid search, an effective method for adjusting the parameters in supervised models for the best generalization performance.

Cross-Validation

Cross-validation is a statistical method of evaluating generalization performance that is more stable and thorough than using a split into a training and a test set. In cross-validation, the data is instead split repeatedly and multiple models are trained. The most commonly used version of cross-validation is *k-fold cross-validation*, where k is a user-specified number, usually 5 or 10. When performing five-fold cross-validation, the data is first partitioned into five parts of (approximately) equal size, called *folds*. Next, a sequence of models is trained. The first model is trained using the first fold as the test set, and the remaining folds (2–5) are used as the training set. The model is built using the data in folds 2–5, and then the accuracy is evaluated on fold 1. Then another model is built, this time using fold 2 as the test set and the data in folds 1, 3, 4, and 5 as the training set. This process is repeated using folds 3, 4, and 5 as test sets. For each of these five *splits* of the data into training and test sets, we compute the accuracy. In the end, we have collected five accuracy values. The process is illustrated in [Figure 5-1](#):

In[3]:

```
mglearn.plots.plot_cross_validation()
```

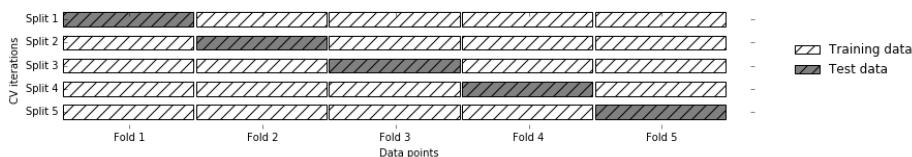


Figure 5-1. Data splitting in five-fold cross-validation

Usually, the first fifth of the data is the first fold, the second fifth of the data is the second fold, and so on.

Cross-Validation in scikit-learn

Cross-validation is implemented in scikit-learn using the `cross_val_score` function from the `model_selection` module. The parameters of the `cross_val_score` function are the model we want to evaluate, the training data, and the ground-truth labels. Let's evaluate `LogisticRegression` on the `iris` dataset:

In[4]:

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression

iris = load_iris()
logreg = LogisticRegression()

scores = cross_val_score(logreg, iris.data, iris.target)
print("Cross-validation scores: {}".format(scores))
```

Out[4]:

```
Cross-validation scores: [ 0.961  0.922  0.958]
```

By default, `cross_val_score` performs three-fold cross-validation, returning three accuracy values. We can change the number of folds used by changing the `cv` parameter:

In[5]:

```
scores = cross_val_score(logreg, iris.data, iris.target, cv=5)
print("Cross-validation scores: {}".format(scores))
```

Out[5]:

```
Cross-validation scores: [ 1.      0.967  0.933  0.9     1.      ]
```

A common way to summarize the cross-validation accuracy is to compute the mean:

In[6]:

```
print("Average cross-validation score: {:.2f}".format(scores.mean()))
```

Out[6]:

```
Average cross-validation score: 0.96
```

Using the mean cross-validation we can conclude that we expect the model to be around 96% accurate on average. Looking at all five scores produced by the five-fold cross-validation, we can also conclude that there is a relatively high variance in the accuracy between folds, ranging from 100% accuracy to 90% accuracy. This could imply that the model is very dependent on the particular folds used for training, but it could also just be a consequence of the small size of the dataset.

Benefits of Cross-Validation

There are several benefits to using cross-validation instead of a single split into a training and a test set. First, remember that `train_test_split` performs a random split of the data. Imagine that we are “lucky” when randomly splitting the data, and all examples that are hard to classify end up in the training set. In that case, the test set will only contain “easy” examples, and our test set accuracy will be unrealistically high. Conversely, if we are “unlucky,” we might have randomly put all the hard-to-classify examples in the test set and consequently obtain an unrealistically low score. However, when using cross-validation, each example will be in the training set exactly once: each example is in one of the folds, and each fold is the test set once. Therefore, the model needs to generalize well to all of the samples in the dataset for all of the cross-validation scores (and their mean) to be high.

Having multiple splits of the data also provides some information about how sensitive our model is to the selection of the training dataset. For the `iris` dataset, we saw accuracies between 90% and 100%. This is quite a range, and it provides us with an idea about how the model might perform in the worst case and best case scenarios when applied to new data.

Another benefit of cross-validation as compared to using a single split of the data is that we use our data more effectively. When using `train_test_split`, we usually use 75% of the data for training and 25% of the data for evaluation. When using five-fold cross-validation, in each iteration we can use four-fifths of the data (80%) to fit the model. When using 10-fold cross-validation, we can use nine-tenths of the data (90%) to fit the model. More data will usually result in more accurate models.

The main disadvantage of cross-validation is increased computational cost. As we are now training k models instead of a single model, cross-validation will be roughly k times slower than doing a single split of the data.



It is important to keep in mind that cross-validation is not a way to build a model that can be applied to new data. Cross-validation does not return a model. When calling `cross_val_score`, multiple models are built internally, but the purpose of cross-validation is only to evaluate how well a given algorithm will generalize when trained on a specific dataset.

Stratified k-Fold Cross-Validation and Other Strategies

Splitting the dataset into k folds by starting with the first one- k -th part of the data, as described in the previous section, might not always be a good idea. For example, let’s have a look at the `iris` dataset:

In[7]:

```
from sklearn.datasets import load_iris  
iris = load_iris()  
print("Iris labels:\n{}".format(iris.target))
```

Out[7]:

As you can see, the first third of the data is the class 0, the second third is the class 1, and the last third is the class 2. Imagine doing three-fold cross-validation on this dataset. The first fold would be only class 0, so in the first split of the data, the test set would be only class 0, and the training set would be only classes 1 and 2. As the classes in training and test sets would be different for all three splits, the three-fold cross-validation accuracy would be zero on this dataset. That is not very helpful, as we can do much better than 0% accuracy on `iris`.

As the simple k -fold strategy fails here, scikit-learn does not use it for classification, but rather uses *stratified k -fold cross-validation*. In stratified cross-validation, we split the data such that the proportions between classes are the same in each fold as they are in the whole dataset, as illustrated in Figure 5-2:

In[8]:

```
mglearn.plots.plot_stratified_cross_validation()
```

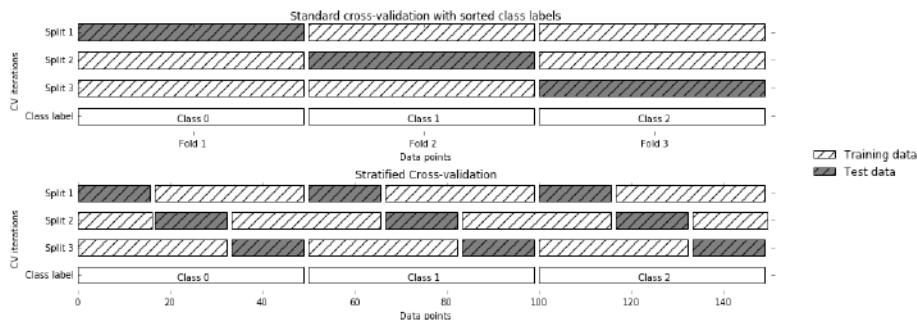


Figure 5-2. Comparison of standard cross-validation and stratified cross-validation when the data is ordered by class label

For example, if 90% of your samples belong to class A and 10% of your samples belong to class B, then stratified cross-validation ensures that in each fold, 90% of samples belong to class A and 10% of samples belong to class B.

It is usually a good idea to use stratified k -fold cross-validation instead of k -fold cross-validation to evaluate a classifier, because it results in more reliable estimates of generalization performance. In the case of only 10% of samples belonging to class B, using standard k -fold cross-validation it might easily happen that one fold only contains samples of class A. Using this fold as a test set would not be very informative about the overall performance of the classifier.

For regression, scikit-learn uses the standard k -fold cross-validation by default. It would be possible to also try to make each fold representative of the different values the regression target has, but this is not a commonly used strategy and would be surprising to most users.

More control over cross-validation

We saw earlier that we can adjust the number of folds that are used in `cross_val_score` using the `cv` parameter. However, scikit-learn allows for much finer control over what happens during the splitting of the data by providing a *cross-validation splitter* as the `cv` parameter. For most use cases, the defaults of k -fold cross-validation for regression and stratified k -fold for classification work well, but there are some cases where you might want to use a different strategy. Say, for example, we want to use the standard k -fold cross-validation on a classification dataset to reproduce someone else's results. To do this, we first have to import the `KFold` splitter class from the `model_selection` module and instantiate it with the number of folds we want to use:

In[9]:

```
from sklearn.model_selection import KFold
kfold = KFold(n_splits=5)
```

Then, we can pass the `kfold` splitter object as the `cv` parameter to `cross_val_score`:

In[10]:

```
print("Cross-validation scores:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

Out[10]:

```
Cross-validation scores:
 [ 1.      0.933  0.433  0.967  0.433]
```

This way, we can verify that it is indeed a really bad idea to use three-fold (nonstratified) cross-validation on the `iris` dataset:

In[11]:

```
kfold = KFold(n_splits=3)
print("Cross-validation scores:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

Out[11]:

```
Cross-validation scores:
[ 0.  0.  0.]
```

Remember: each fold corresponds to one of the classes in the `iris` dataset, and so nothing can be learned. Another way to resolve this problem is to shuffle the data instead of stratifying the folds, to remove the ordering of the samples by label. We can do that by setting the `shuffle` parameter of `KFold` to `True`. If we shuffle the data, we also need to fix the `random_state` to get a reproducible shuffling. Otherwise, each run of `cross_val_score` would yield a different result, as each time a different split would be used (this might not be a problem, but can be surprising). Shuffling the data before splitting it yields a much better result:

In[12]:

```
kfold = KFold(n_splits=3, shuffle=True, random_state=0)
print("Cross-validation scores:\n{}".format(
    cross_val_score(logreg, iris.data, iris.target, cv=kfold)))
```

Out[12]:

```
Cross-validation scores:
[ 0.9  0.96 0.96]
```

Leave-one-out cross-validation

Another frequently used cross-validation method is *leave-one-out*. You can think of leave-one-out cross-validation as k -fold cross-validation where each fold is a single sample. For each split, you pick a single data point to be the test set. This can be very time consuming, particularly for large datasets, but sometimes provides better estimates on small datasets:

In[13]:

```
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()
scores = cross_val_score(logreg, iris.data, iris.target, cv=loo)
print("Number of cv iterations: ", len(scores))
print("Mean accuracy: {:.2f}".format(scores.mean()))
```

Out[13]:

```
Number of cv iterations:  150
Mean accuracy: 0.95
```

Shuffle-split cross-validation

Another, very flexible strategy for cross-validation is *shuffle-split cross-validation*. In shuffle-split cross-validation, each split samples `train_size` many points for the training set and `test_size` many (disjoint) point for the test set. This splitting is repeated `n_iter` times. Figure 5-3 illustrates running four iterations of splitting a dataset consisting of 10 points, with a training set of 5 points and test sets of 2 points each (you can use integers for `train_size` and `test_size` to use absolute sizes for these sets, or floating-point numbers to use fractions of the whole dataset):

In[14]:

```
mlearn.plots.plot_shuffle_split()
```



Figure 5-3. `ShuffleSplit` with 10 points, `train_size`=5, `test_size`=2, and `n_iter`=4

The following code splits the dataset into 50% training set and 50% test set for 10 iterations:

In[15]:

```
from sklearn.model_selection import ShuffleSplit
shuffle_split = ShuffleSplit(test_size=.5, train_size=.5, n_splits=10)
scores = cross_val_score(logreg, iris.data, iris.target, cv=shuffle_split)
print("Cross-validation scores:\n{}".format(scores))
```

Out[15]:

```
Cross-validation scores:
[ 0.96   0.907   0.947   0.96   0.96   0.907   0.893   0.907   0.92   0.973]
```

Shuffle-split cross-validation allows for control over the number of iterations independently of the training and test sizes, which can sometimes be helpful. It also allows for using only part of the data in each iteration, by providing `train_size` and `test_size` settings that don't add up to one. Subsampling the data in this way can be useful for experimenting with large datasets.

There is also a stratified variant of `ShuffleSplit`, aptly named `StratifiedShuffleSplit`, which can provide more reliable results for classification tasks.

Cross-validation with groups

Another very common setting for cross-validation is when there are groups in the data that are highly related. Say you want to build a system to recognize emotions from pictures of faces, and you collect a dataset of pictures of 100 people where each person is captured multiple times, showing various emotions. The goal is to build a classifier that can correctly identify emotions of people not in the dataset. You could use the default stratified cross-validation to measure the performance of a classifier here. However, it is likely that pictures of the same person will be in both the training and the test set. It will be much easier for a classifier to detect emotions in a face that is part of the training set, compared to a completely new face. To accurately evaluate the generalization to new faces, we must therefore ensure that the training and test sets contain images of different people.

To achieve this, we can use `GroupKFold`, which takes an array of `groups` as argument that we can use to indicate which person is in the image. The `groups` array here indicates groups in the data that should not be split when creating the training and test sets, and should not be confused with the class label.

This example of groups in the data is common in medical applications, where you might have multiple samples from the same patient, but are interested in generalizing to new patients. Similarly, in speech recognition, you might have multiple recordings of the same speaker in your dataset, but are interested in recognizing speech of new speakers.

The following is an example of using a synthetic dataset with a grouping given by the `groups` array. The dataset consists of 12 data points, and for each of the data points, `groups` specifies which group (think patient) the point belongs to. The `groups` specify that there are four groups, and the first three samples belong to the first group, the next four samples belong to the second group, and so on:

In[17]:

```
from sklearn.model_selection import GroupKFold
# create synthetic dataset
X, y = make_blobs(n_samples=12, random_state=0)
# assume the first three samples belong to the same group,
# then the next four, etc.
groups = [0, 0, 0, 1, 1, 1, 2, 2, 3, 3, 3]
scores = cross_val_score(logreg, X, y, groups, cv=GroupKFold(n_splits=3))
print("Cross-validation scores:\n{}".format(scores))
```

Out[17]:

```
Cross-validation scores:
[ 0.75  0.8   0.667]
```

The samples don't need to be ordered by group; we just did this for illustration purposes. The splits that are calculated based on these labels are visualized in [Figure 5-4](#).

As you can see, for each split, each group is either entirely in the training set or entirely in the test set:

In[16]:

```
mglearn.plots.plot_label_kfold()
```

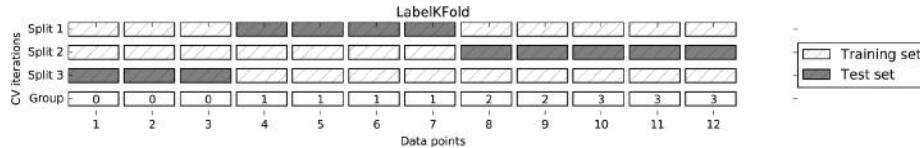


Figure 5-4. Label-dependent splitting with GroupKFold

There are more splitting strategies for cross-validation in scikit-learn, which allow for an even greater variety of use cases (you can find these in the [scikit-learn user guide](#)). However, the standard KFold, StratifiedKFold, and GroupKFold are by far the most commonly used ones.

Grid Search

Now that we know how to evaluate how well a model generalizes, we can take the next step and improve the model's generalization performance by tuning its parameters. We discussed the parameter settings of many of the algorithms in scikit-learn in Chapters 2 and 3, and it is important to understand what the parameters mean before trying to adjust them. Finding the values of the important parameters of a model (the ones that provide the best generalization performance) is a tricky task, but necessary for almost all models and datasets. Because it is such a common task, there are standard methods in scikit-learn to help you with it. The most commonly used method is *grid search*, which basically means trying all possible combinations of the parameters of interest.

Consider the case of a kernel SVM with an RBF (radial basis function) kernel, as implemented in the SVC class. As we discussed in Chapter 2, there are two important parameters: the kernel bandwidth, `gamma`, and the regularization parameter, `C`. Say we want to try the values `0.001`, `0.01`, `0.1`, `1`, `10`, and `100` for the parameter `C`, and the same for `gamma`. Because we have six different settings for `C` and `gamma` that we want to try, we have 36 combinations of parameters in total. Looking at all possible combinations creates a table (or grid) of parameter settings for the SVM, as shown here:

	C = 0.001	C = 0.01	... C = 10
gamma=0.001	SVC(C=0.001, gamma=0.001)	SVC(C=0.01, gamma=0.001)	... SVC(C=10, gamma=0.001)
gamma=0.01	SVC(C=0.001, gamma=0.01)	SVC(C=0.01, gamma=0.01)	... SVC(C=10, gamma=0.01)
...
gamma=100	SVC(C=0.001, gamma=100)	SVC(C=0.01, gamma=100)	... SVC(C=10, gamma=100)

Simple Grid Search

We can implement a simple grid search just as for loops over the two parameters, training and evaluating a classifier for each combination:

In[18]:

```
# naive grid search implementation
from sklearn.svm import SVC
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
print("Size of training set: {}    size of test set: {}".format(
    X_train.shape[0], X_test.shape[0]))

best_score = 0

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # for each combination of parameters, train an SVC
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # evaluate the SVC on the test set
        score = svm.score(X_test, y_test)
        # if we got a better score, store the score and parameters
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}

print("Best score: {:.2f}".format(best_score))
print("Best parameters: {}".format(best_parameters))
```

Out[18]:

```
Size of training set: 112    size of test set: 38
Best score: 0.97
Best parameters: {'C': 100, 'gamma': 0.001}
```

The Danger of Overfitting the Parameters and the Validation Set

Given this result, we might be tempted to report that we found a model that performs with 97% accuracy on our dataset. However, this claim could be overly optimistic (or just wrong), for the following reason: we tried many different parameters and

selected the one with best accuracy on the test set, but this accuracy won't necessarily carry over to new data. Because we used the test data to adjust the parameters, we can no longer use it to assess how good the model is. This is the same reason we needed to split the data into training and test sets in the first place; we need an independent dataset to evaluate, one that was not used to create the model.

One way to resolve this problem is to split the data again, so we have three sets: the training set to build the model, the validation (or development) set to select the parameters of the model, and the test set to evaluate the performance of the selected parameters. [Figure 5-5](#) shows what this looks like:

In[19]:

```
mglearn.plots.plot_threefold_split()
```

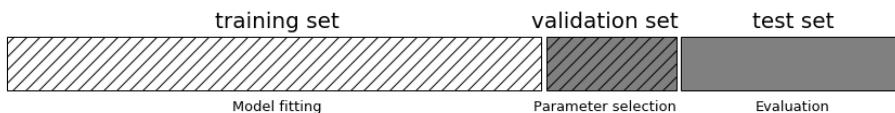


Figure 5-5. A threefold split of data into training set, validation set, and test set

After selecting the best parameters using the validation set, we can rebuild a model using the parameter settings we found, but now training on both the training data and the validation data. This way, we can use as much data as possible to build our model. This leads to the following implementation:

In[20]:

```
from sklearn.svm import SVC
# split data into train+validation set and test set
X_trainval, X_test, y_trainval, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
# split train+validation set into training and validation sets
X_train, X_valid, y_train, y_valid = train_test_split(
    X_trainval, y_trainval, random_state=1)
print("Size of training set: {}    size of validation set: {}    size of test set: {}"
      .format(X_train.shape[0], X_valid.shape[0], X_test.shape[0]))
best_score = 0

for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # for each combination of parameters, train an SVC
        svm = SVC(gamma=gamma, C=C)
        svm.fit(X_train, y_train)
        # evaluate the SVC on the test set
        score = svm.score(X_valid, y_valid)
        # if we got a better score, store the score and parameters
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
```

```
# rebuild a model on the combined training and validation set,
# and evaluate it on the test set
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
test_score = svm.score(X_test, y_test)
print("Best score on validation set: {:.2f}".format(best_score))
print("Best parameters: ", best_parameters)
print("Test set score with best parameters: {:.2f}".format(test_score))
```

Out[20]:

```
Size of training set: 84    size of validation set: 28    size of test set: 38

Best score on validation set: 0.96
Best parameters: {'C': 10, 'gamma': 0.001}
Test set score with best parameters: 0.92
```

The best score on the validation set is 96%: slightly lower than before, probably because we used less data to train the model (`X_train` is smaller now because we split our dataset twice). However, the score on the test set—the score that actually tells us how well we generalize—is even lower, at 92%. So we can only claim to classify new data 92% correctly, not 97% correctly as we thought before!

The distinction between the training set, validation set, and test set is fundamentally important to applying machine learning methods in practice. Any choices made based on the test set accuracy “leak” information from the test set into the model. Therefore, it is important to keep a separate test set, which is only used for the final evaluation. It is good practice to do all exploratory analysis and model selection using the combination of a training and a validation set, and reserve the test set for a final evaluation—this is even true for exploratory visualization. Strictly speaking, evaluating more than one model on the test set and choosing the better of the two will result in an overly optimistic estimate of how accurate the model is.

Grid Search with Cross-Validation

While the method of splitting the data into a training, a validation, and a test set that we just saw is workable, and relatively commonly used, it is quite sensitive to how exactly the data is split. From the output of the previous code snippet we can see that `GridSearchCV` selects '`'C': 10, 'gamma': 0.001` as the best parameters, while the output of the code in the previous section selects '`'C': 100, 'gamma': 0.001` as the best parameters. For a better estimate of the generalization performance, instead of using a single split into a training and a validation set, we can use cross-validation to evaluate the performance of each parameter combination. This method can be coded up as follows:

In[21]:

```
for gamma in [0.001, 0.01, 0.1, 1, 10, 100]:
    for C in [0.001, 0.01, 0.1, 1, 10, 100]:
        # for each combination of parameters,
        # train an SVC
        svm = SVC(gamma=gamma, C=C)
        # perform cross-validation
        scores = cross_val_score(svm, X_trainval, y_trainval, cv=5)
        # compute mean cross-validation accuracy
        score = np.mean(scores)
        # if we got a better score, store the score and parameters
        if score > best_score:
            best_score = score
            best_parameters = {'C': C, 'gamma': gamma}
# rebuild a model on the combined training and validation set
svm = SVC(**best_parameters)
svm.fit(X_trainval, y_trainval)
```

To evaluate the accuracy of the SVM using a particular setting of C and γ using five-fold cross-validation, we need to train $36 * 5 = 180$ models. As you can imagine, the main downside of the use of cross-validation is the time it takes to train all these models.

The following visualization (Figure 5-6) illustrates how the best parameter setting is selected in the preceding code:

In[22]:

```
mglearn.plots.plot_cross_val_selection()
```

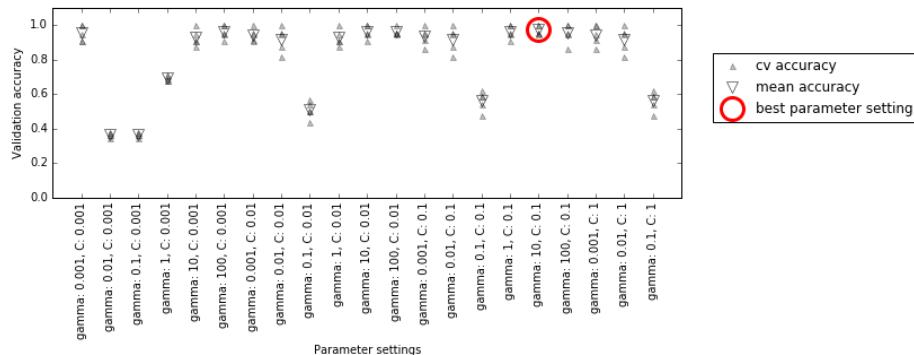


Figure 5-6. Results of grid search with cross-validation

For each parameter setting (only a subset is shown), five accuracy values are computed, one for each split in the cross-validation. Then the mean validation accuracy is computed for each parameter setting. The parameters with the highest mean validation accuracy are chosen, marked by the circle.



As we said earlier, cross-validation is a way to evaluate a given algorithm on a specific dataset. However, it is often used in conjunction with parameter search methods like grid search. For this reason, many people use the term *cross-validation* colloquially to refer to grid search with cross-validation.

The overall process of splitting the data, running the grid search, and evaluating the final parameters is illustrated in [Figure 5-7](#):

In[23]:

```
mglearn.plots.plot_grid_search_overview()
```

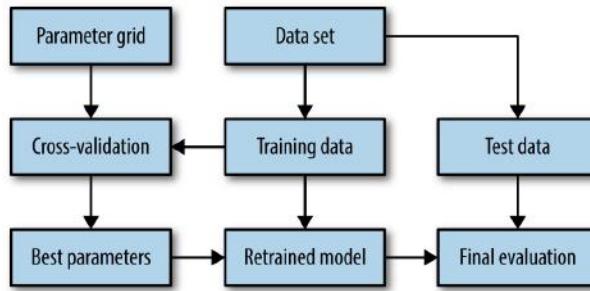


Figure 5-7. Overview of the process of parameter selection and model evaluation with `GridSearchCV`

Because grid search with cross-validation is such a commonly used method to adjust parameters, `scikit-learn` provides the `GridSearchCV` class, which implements it in the form of an estimator. To use the `GridSearchCV` class, you first need to specify the parameters you want to search over using a dictionary. `GridSearchCV` will then perform all the necessary model fits. The keys of the dictionary are the names of parameters we want to adjust (as given when constructing the model—in this case, `C` and `gamma`), and the values are the parameter settings we want to try out. Trying the values `0.001`, `0.01`, `0.1`, `1`, `10`, and `100` for `C` and `gamma` translates to the following dictionary:

In[24]:

```
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],  
             'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}  
print("Parameter grid:\n{}".format(param_grid))
```

Out[24]:

```
Parameter grid:  
{'C': [0.001, 0.01, 0.1, 1, 10, 100], 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

We can now instantiate the `GridSearchCV` class with the model (`SVC`), the parameter grid to search (`param_grid`), and the cross-validation strategy we want to use (say, five-fold stratified cross-validation):

In[25]:

```
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
```

`GridSearchCV` will use cross-validation in place of the split into a training and validation set that we used before. However, we still need to split the data into a training and a test set, to avoid overfitting the parameters:

In[26]:

```
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, random_state=0)
```

The `grid_search` object that we created behaves just like a classifier; we can call the standard methods `fit`, `predict`, and `score` on it.¹ However, when we call `fit`, it will run cross-validation for each combination of parameters we specified in `param_grid`:

In[27]:

```
grid_search.fit(X_train, y_train)
```

Fitting the `GridSearchCV` object not only searches for the best parameters, but also automatically fits a new model on the whole training dataset with the parameters that yielded the best cross-validation performance. What happens in `fit` is therefore equivalent to the result of the In[21] code we saw at the beginning of this section. The `GridSearchCV` class provides a very convenient interface to access the retrained model using the `predict` and `score` methods. To evaluate how well the best found parameters generalize, we can call `score` on the test set:

In[28]:

```
print("Test set score: {:.2f}".format(grid_search.score(X_test, y_test)))
```

Out[28]:

```
Test set score: 0.97
```

Choosing the parameters using cross-validation, we actually found a model that achieves 97% accuracy on the test set. The important thing here is that we *did not use the test set* to choose the parameters. The parameters that were found are scored in the

¹ A scikit-learn estimator that is created using another estimator is called a *meta-estimator*. `GridSearchCV` is the most commonly used meta-estimator, but we will see more later.

`best_params_` attribute, and the best cross-validation accuracy (the mean accuracy over the different splits for this parameter setting) is stored in `best_score_`:

In[29]:

```
print("Best parameters: {}".format(grid_search.best_params_))
print("Best cross-validation score: {:.2f}".format(grid_search.best_score_))
```

Out[29]:

```
Best parameters: {'C': 100, 'gamma': 0.01}
Best cross-validation score: 0.97
```



Again, be careful not to confuse `best_score_` with the generalization performance of the model as computed by the `score` method on the test set. Using the `score` method (or evaluating the output of the `predict` method) employs a model *trained on the whole training set*. The `best_score_` attribute stores the mean cross-validation accuracy, with *cross-validation performed on the training set*.

Sometimes it is helpful to have access to the actual model that was found—for example, to look at coefficients or feature importances. You can access the model with the best parameters trained on the whole training set using the `best_estimator_` attribute:

In[30]:

```
print("Best estimator:\n{}".format(grid_search.best_estimator_))
```

Out[30]:

```
Best estimator:
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
      decision_function_shape=None, degree=3, gamma=0.01, kernel='rbf',
      max_iter=-1, probability=False, random_state=None, shrinking=True,
      tol=0.001, verbose=False)
```

Because `grid_search` itself has `predict` and `score` methods, using `best_estimator_` is not needed to make predictions or evaluate the model.

Analyzing the result of cross-validation

It is often helpful to visualize the results of cross-validation, to understand how the model generalization depends on the parameters we are searching. As grid searches are quite computationally expensive to run, often it is a good idea to start with a relatively coarse and small grid. We can then inspect the results of the cross-validated grid search, and possibly expand our search. The results of a grid search can be found in the `cv_results_` attribute, which is a dictionary storing all aspects of the search. It

contains a lot of details, as you can see in the following output, and is best looked at after converting it to a pandas DataFrame:

In[31]:

```
import pandas as pd
# convert to DataFrame
results = pd.DataFrame(grid_search.cv_results_)
# show the first 5 rows
display(results.head())
```

Out[31]:

	param_C	param_gamma	params	mean_test_score
0	0.001	0.001	{'C': 0.001, 'gamma': 0.001}	0.366
1	0.001	0.01	{'C': 0.001, 'gamma': 0.01}	0.366
2	0.001	0.1	{'C': 0.001, 'gamma': 0.1}	0.366
3	0.001	1	{'C': 0.001, 'gamma': 1}	0.366
4	0.001	10	{'C': 0.001, 'gamma': 10}	0.366

	rank_test_score	split0_test_score	split1_test_score	split2_test_score
0	22	0.375	0.347	0.363
1	22	0.375	0.347	0.363
2	22	0.375	0.347	0.363
3	22	0.375	0.347	0.363
4	22	0.375	0.347	0.363

	split3_test_score	split4_test_score	std_test_score
0	0.363	0.380	0.011
1	0.363	0.380	0.011
2	0.363	0.380	0.011
3	0.363	0.380	0.011
4	0.363	0.380	0.011

Each row in `results` corresponds to one particular parameter setting. For each setting, the results of all cross-validation splits are recorded, as well as the mean and standard deviation over all splits. As we were searching a two-dimensional grid of parameters (`C` and `gamma`), this is best visualized as a heat map (Figure 5-8). First we extract the mean validation scores, then we reshape the scores so that the axes correspond to `C` and `gamma`:

In[32]:

```
scores = np.array(results.mean_test_score).reshape(6, 6)

# plot the mean cross-validation scores
mglearn.tools.heatmap(scores, xlabel='gamma', xticklabels=param_grid['gamma'],
                      ylabel='C', yticklabels=param_grid['C'], cmap="viridis")
```



Figure 5-8. Heat map of mean cross-validation score as a function of C and γ

Each point in the heat map corresponds to one run of cross-validation, with a particular parameter setting. The color encodes the cross-validation accuracy, with light colors meaning high accuracy and dark colors meaning low accuracy. You can see that SVC is very sensitive to the setting of the parameters. For many of the parameter settings, the accuracy is around 40%, which is quite bad; for other settings the accuracy is around 96%. We can take away from this plot several things. First, the parameters we adjusted are *very important* for obtaining good performance. Both parameters (C and γ) matter a lot, as adjusting them can change the accuracy from 40% to 96%. Additionally, the ranges we picked for the parameters are ranges in which we see significant changes in the outcome. It's also important to note that the ranges for the parameters are large enough: the optimum values for each parameter are not on the edges of the plot.

Now let's look at some plots (shown in Figure 5-9) where the result is less ideal, because the search ranges were not chosen properly:



Figure 5-9. Heat map visualizations of misspecified search grids

In[33]:

```
fig, axes = plt.subplots(1, 3, figsize=(13, 5))

param_grid_linear = {'C': np.linspace(1, 2, 6),
                     'gamma': np.linspace(1, 2, 6)}

param_grid_one_log = {'C': np.linspace(1, 2, 6),
                      'gamma': np.logspace(-3, 2, 6)}

param_grid_range = {'C': np.logspace(-3, 2, 6),
                    'gamma': np.logspace(-7, -2, 6)}

for param_grid, ax in zip([param_grid_linear, param_grid_one_log,
                           param_grid_range], axes):
    grid_search = GridSearchCV(SVC(), param_grid, cv=5)
    grid_search.fit(X_train, y_train)
    scores = grid_search.cv_results_['mean_test_score'].reshape(6, 6)

    # plot the mean cross-validation scores
    scores_image = mglearn.tools.heatmap(
        scores, xlabel='gamma', ylabel='C', xticklabels=param_grid['gamma'],
        yticklabels=param_grid['C'], cmap="viridis", ax=ax)

    plt.colorbar(scores_image, ax=axes.tolist())
```

The first panel shows no changes at all, with a constant color over the whole parameter grid. In this case, this is caused by improper scaling and range of the parameters `C` and `gamma`. However, if no change in accuracy is visible over the different parameter settings, it could also be that a parameter is just not important at all. It is usually good to try very extreme values first, to see if there are any changes in the accuracy as a result of changing a parameter.

The second panel shows a vertical stripe pattern. This indicates that only the setting of the `gamma` parameter makes any difference. This could mean that the `gamma` parameter is searching over interesting values but the `C` parameter is not—or it could mean the `C` parameter is not important.

The third panel shows changes in both `C` and `gamma`. However, we can see that in the entire bottom left of the plot, nothing interesting is happening. We can probably exclude the very small values from future grid searches. The optimum parameter setting is at the top right. As the optimum is in the border of the plot, we can expect that there might be even better values beyond this border, and we might want to change our search range to include more parameters in this region.

Tuning the parameter grid based on the cross-validation scores is perfectly fine, and a good way to explore the importance of different parameters. However, you should not test different parameter ranges on the final test set—as we discussed earlier, eval-

uation of the test set should happen only once we know exactly what model we want to use.

Search over spaces that are not grids

In some cases, trying all possible combinations of all parameters as `GridSearchCV` usually does, is not a good idea. For example, `SVC` has a `kernel` parameter, and depending on which kernel is chosen, other parameters will be relevant. If `kernel='linear'`, the model is linear, and only the `C` parameter is used. If `kernel='rbf'`, both the `C` and `gamma` parameters are used (but not other parameters like `degree`). In this case, searching over all possible combinations of `C`, `gamma`, and `kernel` wouldn't make sense: if `kernel='linear'`, `gamma` is not used, and trying different values for `gamma` would be a waste of time. To deal with these kinds of "conditional" parameters, `GridSearchCV` allows the `param_grid` to be a *list of dictionaries*. Each dictionary in the list is expanded into an independent grid. A possible grid search involving `kernel` and `parameters` could look like this:

In[34]:

```
param_grid = [{'kernel': ['rbf'],
              'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}],
              {'kernel': ['linear'],
              'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
print("List of grids:\n{}".format(param_grid))
```

Out[34]:

```
List of grids:
[{'kernel': ['rbf'], 'C': [0.001, 0.01, 0.1, 1, 10, 100],
 'gamma': [0.001, 0.01, 0.1, 1, 10, 100]},
 {'kernel': ['linear'], 'C': [0.001, 0.01, 0.1, 1, 10, 100]}]
```

In the first grid, the `kernel` parameter is always set to `'rbf'` (not that the entry for `kernel` is a list of length one), and both the `C` and `gamma` parameters are varied. In the second grid, the `kernel` parameter is always set to `linear`, and only `C` is varied. Now let's apply this more complex parameter search:

In[35]:

```
grid_search = GridSearchCV(SVC(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
print("Best parameters: {}".format(grid_search.best_params_))
print("Best cross-validation score: {:.2f}".format(grid_search.best_score_))
```

Out[35]:

```
Best parameters: {'C': 100, 'kernel': 'rbf', 'gamma': 0.01}
Best cross-validation score: 0.97
```

Let's look at the `cv_results_` again. As expected, if `kernel` is 'linear', then only C is varied:

In[36]:

```
results = pd.DataFrame(grid_search.cv_results_)
# we display the transposed table so that it better fits on the page:
display(results.T)
```

Out[36]:

	0	1	2	3	...	38	39	40	41
param_C	0.001	0.001	0.001	0.001	...	0.1	1	10	100
param_gamma	0.001	0.01	0.1	1	...	NaN	NaN	NaN	NaN
param_kernel	rbf	rbf	rbf	rbf	...	linear	linear	linear	linear
params	{C: 0.001, kernel: rbf, gamma: 0.001}	{C: 0.001, kernel: rbf, gamma: 0.01}	{C: 0.001, kernel: rbf, gamma: 0.1}	{C: 0.001, kernel: rbf, gamma: 1}	...	{C: 0.1, kernel: linear}	{C: 1, kernel: linear}	{C: 10, kernel: linear}	{C: 100, kernel: linear}
mean_test_score	0.37	0.37	0.37	0.37	...	0.95	0.97	0.96	0.96
rank_test_score	27	27	27	27	...	11	1	3	3
split0_test_score	0.38	0.38	0.38	0.38	...	0.96	1	0.96	0.96
split1_test_score	0.35	0.35	0.35	0.35	...	0.91	0.96	1	1
split2_test_score	0.36	0.36	0.36	0.36	...	1	1	1	1
split3_test_score	0.36	0.36	0.36	0.36	...	0.91	0.95	0.91	0.91
split4_test_score	0.38	0.38	0.38	0.38	...	0.95	0.95	0.95	0.95
std_test_score	0.011	0.011	0.011	0.011	...	0.033	0.022	0.034	0.034

12 rows × 42 columns

Using different cross-validation strategies with grid search

Similarly to `cross_val_score`, `GridSearchCV` uses stratified k -fold cross-validation by default for classification, and k -fold cross-validation for regression. However, you can also pass any cross-validation splitter, as described in “[More control over cross-validation](#)” on page 256, as the `cv` parameter in `GridSearchCV`. In particular, to get only a single split into a training and a validation set, you can use `ShuffleSplit` or `StratifiedShuffleSplit` with `n_iter=1`. This might be helpful for very large datasets, or very slow models.

Nested cross-validation

In the preceding examples, we went from using a single split of the data into training, validation, and test sets to splitting the data into training and test sets and then performing cross-validation on the training set. But when using `GridSearchCV` as

described earlier, we still have a single split of the data into training and test sets, which might make our results unstable and make us depend too much on this single split of the data. We can go a step further, and instead of splitting the original data into training and test sets once, use multiple splits of cross-validation. This will result in what is called *nested cross-validation*. In nested cross-validation, there is an outer loop over splits of the data into training and test sets. For each of them, a grid search is run (which might result in different best parameters for each split in the outer loop). Then, for each outer split, the test set score using the best settings is reported.

The result of this procedure is a list of scores—not a model, and not a parameter setting. The scores tell us how well a model generalizes, given the best parameters found by the grid. As it doesn't provide a model that can be used on new data, nested cross-validation is rarely used when looking for a predictive model to apply to future data. However, it can be useful for evaluating how well a given model works on a particular dataset.

Implementing nested cross-validation in `scikit-learn` is straightforward. We call `cross_val_score` with an instance of `GridSearchCV` as the model:

In[34]:

```
scores = cross_val_score(GridSearchCV(SVC(), param_grid, cv=5),
                        iris.data, iris.target, cv=5)
print("Cross-validation scores: ", scores)
print("Mean cross-validation score: ", scores.mean())
```

Out[34]:

```
Cross-validation scores: [ 0.967  1.      0.967  0.967  1.      ]
Mean cross-validation score:  0.98
```

The result of our nested cross-validation can be summarized as “SVC can achieve 98% mean cross-validation accuracy on the `iris` dataset”—nothing more and nothing less.

Here, we used stratified five-fold cross-validation in both the inner and the outer loop. As our `param_grid` contains 36 combinations of parameters, this results in a whopping $36 * 5 * 5 = 900$ models being built, making nested cross-validation a very expensive procedure. Here, we used the same cross-validation splitter in the inner and the outer loop; however, this is not necessary and you can use any combination of cross-validation strategies in the inner and outer loops. It can be a bit tricky to understand what is happening in the single line given above, and it can be helpful to visualize it as `for` loops, as done in the following simplified implementation:

In[35]:

```
def nested_cv(X, y, inner_cv, outer_cv, Classifier, parameter_grid):
    outer_scores = []
    # for each split of the data in the outer cross-validation
    # (split method returns indices)
    for training_samples, test_samples in outer_cv.split(X, y):
        # find best parameter using inner cross-validation
        best_params = {}
        best_score = -np.inf
        # iterate over parameters
        for parameters in parameter_grid:
            # accumulate score over inner splits
            cv_scores = []
            # iterate over inner cross-validation
            for inner_train, inner_test in inner_cv.split(
                X[training_samples], y[training_samples]):
                # build classifier given parameters and training data
                clf = Classifier(**parameters)
                clf.fit(X[inner_train], y[inner_train])
                # evaluate on inner test set
                score = clf.score(X[inner_test], y[inner_test])
                cv_scores.append(score)
            # compute mean score over inner folds
            mean_score = np.mean(cv_scores)
            if mean_score > best_score:
                # if better than so far, remember parameters
                best_score = mean_score
                best_params = parameters
        # build classifier on best parameters using outer training set
        clf = Classifier(**best_params)
        clf.fit(X[training_samples], y[training_samples])
        # evaluate
        outer_scores.append(clf.score(X[test_samples], y[test_samples]))
    return np.array(outer_scores)
```

Now, let's run this function on the iris dataset:

In[36]:

```
from sklearn.model_selection import ParameterGrid, StratifiedKFold
scores = nested_cv(iris.data, iris.target, StratifiedKFold(5),
                   StratifiedKFold(5), SVC, ParameterGrid(param_grid))
print("Cross-validation scores: {}".format(scores))
```

Out[36]:

```
Cross-validation scores: [ 0.967  1.      0.967  0.967  1.      ]
```

Parallelizing cross-validation and grid search

While running a grid search over many parameters and on large datasets can be computationally challenging, it is also *embarrassingly parallel*. This means that building a

model using a particular parameter setting on a particular cross-validation split can be done completely independently from the other parameter settings and models. This makes grid search and cross-validation ideal candidates for parallelization over multiple CPU cores or over a cluster. You can make use of multiple cores in `GridSearchCV` and `cross_val_score` by setting the `n_jobs` parameter to the number of CPU cores you want to use. You can set `n_jobs=-1` to use all available cores.

You should be aware that `scikit-learn` *does not allow nesting of parallel operations*. So, if you are using the `n_jobs` option on your model (for example, a random forest), you cannot use it in `GridSearchCV` to search over this model. If your dataset and model are very large, it might be that using many cores uses up too much memory, and you should monitor your memory usage when building large models in parallel.

It is also possible to parallelize grid search and cross-validation over multiple machines in a cluster, although at the time of writing this is not supported within `scikit-learn`. It is, however, possible to use the IPython parallel framework for parallel grid searches, if you don't mind writing the `for` loop over parameters as we did in “[Simple Grid Search](#)” on page 261.

For Spark users, there is also the recently developed `spark-sklearn` package, which allows running a grid search over an already established Spark cluster.

Evaluation Metrics and Scoring

So far, we have evaluated classification performance using accuracy (the fraction of correctly classified samples) and regression performance using R^2 . However, these are only two of the many possible ways to summarize how well a supervised model performs on a given dataset. In practice, these evaluation metrics might not be appropriate for your application, and it is important to choose the right metric when selecting between models and adjusting parameters.

Keep the End Goal in Mind

When selecting a metric, you should always have the end goal of the machine learning application in mind. In practice, we are usually interested not just in making accurate predictions, but in using these predictions as part of a larger decision-making process. Before picking a machine learning metric, you should think about the high-level goal of the application, often called the *business metric*. The consequences of choosing a particular algorithm for a machine learning application are

called the *business impact*.² Maybe the high-level goal is avoiding traffic accidents, or decreasing the number of hospital admissions. It could also be getting more users for your website, or having users spend more money in your shop. When choosing a model or adjusting parameters, you should pick the model or parameter values that have the most positive influence on the business metric. Often this is hard, as assessing the business impact of a particular model might require putting it in production in a real-life system.

In the early stages of development, and for adjusting parameters, it is often infeasible to put models into production just for testing purposes, because of the high business or personal risks that can be involved. Imagine evaluating the pedestrian avoidance capabilities of a self-driving car by just letting it drive around, without verifying it first; if your model is bad, pedestrians will be in trouble! Therefore we often need to find some surrogate evaluation procedure, using an evaluation metric that is easier to compute. For example, we could test classifying images of pedestrians against non-pedestrians and measure accuracy. Keep in mind that this is only a surrogate, and it pays off to find the closest metric to the original business goal that is feasible to evaluate. This closest metric should be used whenever possible for model evaluation and selection. The result of this evaluation might not be a single number—the consequence of your algorithm could be that you have 10% more customers, but each customer will spend 15% less—but it should capture the expected business impact of choosing one model over another.

In this section, we will first discuss metrics for the important special case of binary classification, then turn to multiclass classification and finally regression.

Metrics for Binary Classification

Binary classification is arguably the most common and conceptually simple application of machine learning in practice. However, there are still a number of caveats in evaluating even this simple task. Before we dive into alternative metrics, let's have a look at the ways in which measuring accuracy might be misleading. Remember that for binary classification, we often speak of a *positive* class and a *negative* class, with the understanding that the positive class is the one we are looking for.

Kinds of errors

Often, accuracy is not a good measure of predictive performance, as the number of mistakes we make does not contain all the information we are interested in. Imagine an application to screen for the early detection of cancer using an automated test. If

² We ask scientifically minded readers to excuse the commercial language in this section. Not losing track of the end goal is equally important in science, though the authors are not aware of a similar phrase to “business impact” being used in that realm.

the test is negative, the patient will be assumed healthy, while if the test is positive, the patient will undergo additional screening. Here, we would call a positive test (an indication of cancer) the positive class, and a negative test the negative class. We can't assume that our model will always work perfectly, and it will make mistakes. For any application, we need to ask ourselves what the consequences of these mistakes might be in the real world.

One possible mistake is that a healthy patient will be classified as positive, leading to additional testing. This leads to some costs and an inconvenience for the patient (and possibly some mental distress). An incorrect positive prediction is called a *false positive*. The other possible mistake is that a sick patient will be classified as negative, and will not receive further tests and treatment. The undiagnosed cancer might lead to serious health issues, and could even be fatal. A mistake of this kind—an incorrect negative prediction—is called a *false negative*. In statistics, a false positive is also known as *type I error*, and a false negative as *type II error*. We will stick to “false negative” and “false positive,” as they are more explicit and easier to remember. In the cancer diagnosis example, it is clear that we want to avoid false negatives as much as possible, while false positives can be viewed as more of a minor nuisance.

While this is a particularly drastic example, the consequence of false positives and false negatives are rarely the same. In commercial applications, it might be possible to assign dollar values to both kinds of mistakes, which would allow measuring the error of a particular prediction in dollars, instead of accuracy. This might be much more meaningful for making business decisions on which model to use.

Imbalanced datasets

Types of errors play an important role when one of two classes is much more frequent than the other one. This is very common in practice; a good example is click-through prediction, where each data point represents an “impression,” an item that was shown to a user. This item might be an ad, or a related story, or a related person to follow on a social media site. The goal is to predict whether, if shown a particular item, a user will click on it (indicating they are interested). Most things users are shown on the Internet (in particular, ads) will not result in a click. You might need to show a user 100 ads or articles before they find something interesting enough to click on. This results in a dataset where for each 99 “no click” data points, there is 1 “clicked” data point; in other words, 99% of the samples belong to the “no click” class. Datasets in which one class is much more frequent than the other are often called *imbalanced datasets*, or *datasets with imbalanced classes*. In reality, imbalanced data is the norm, and it is rare that the events of interest have equal or even similar frequency in the data.

Now let's say you build a classifier that is 99% accurate on the click prediction task. What does that tell you? 99% accuracy sounds impressive, but this doesn't take the

class imbalance into account. You can achieve 99% accuracy without building a machine learning model, by always predicting “no click.” On the other hand, even with imbalanced data, a 99% accurate model could in fact be quite good. However, accuracy doesn’t allow us to distinguish the constant “no click” model from a potentially good model.

To illustrate, we’ll create a 9:1 imbalanced dataset from the `digits` dataset, by classifying the digit 9 against the nine other classes:

In[37]:

```
from sklearn.datasets import load_digits

digits = load_digits()
y = digits.target == 9

X_train, X_test, y_train, y_test = train_test_split(
    digits.data, y, random_state=0)
```

We can use the `DummyClassifier` to always predict the majority class (here “not nine”) to see how uninformative accuracy can be:

In[38]:

```
from sklearn.dummy import DummyClassifier
dummy_majority = DummyClassifier(strategy='most_frequent').fit(X_train, y_train)
pred_most_frequent = dummy_majority.predict(X_test)
print("Unique predicted labels: {}".format(np.unique(pred_most_frequent)))
print("Test score: {:.2f}".format(dummy_majority.score(X_test, y_test)))
```

Out[38]:

```
Unique predicted labels: [False]
Test score: 0.90
```

We obtained close to 90% accuracy without learning anything. This might seem striking, but think about it for a minute. Imagine someone telling you their model is 90% accurate. You might think they did a very good job. But depending on the problem, that might be possible by just predicting one class! Let’s compare this against using an actual classifier:

In[39]:

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(max_depth=2).fit(X_train, y_train)
pred_tree = tree.predict(X_test)
print("Test score: {:.2f}".format(tree.score(X_test, y_test)))
```

Out[39]:

```
Test score: 0.92
```

According to accuracy, the `DecisionTreeClassifier` is only slightly better than the constant predictor. This could indicate either that something is wrong with how we used `DecisionTreeClassifier`, or that accuracy is in fact not a good measure here.

For comparison purposes, let's evaluate two more classifiers, `LogisticRegression` and the default `DummyClassifier`, which makes random predictions but produces classes with the same proportions as in the training set:

In[40]:

```
from sklearn.linear_model import LogisticRegression

dummy = DummyClassifier().fit(X_train, y_train)
pred_dummy = dummy.predict(X_test)
print("dummy score: {:.2f}".format(dummy.score(X_test, y_test)))

logreg = LogisticRegression(C=0.1).fit(X_train, y_train)
pred_logreg = logreg.predict(X_test)
print("logreg score: {:.2f}".format(logreg.score(X_test, y_test)))
```

Out[40]:

```
dummy score: 0.80
logreg score: 0.98
```

The dummy classifier that produces random output is clearly the worst of the lot (according to accuracy), while `LogisticRegression` produces very good results. However, even the random classifier yields over 80% accuracy. This makes it very hard to judge which of these results is actually helpful. The problem here is that accuracy is an inadequate measure for quantifying predictive performance in this imbalanced setting. For the rest of this chapter, we will explore alternative metrics that provide better guidance in selecting models. In particular, we would like to have metrics that tell us how much better a model is than making “most frequent” predictions or random predictions, as they are computed in `pred_most_frequent` and `pred_dummy`. If we use a metric to assess our models, it should definitely be able to weed out these nonsense predictions.

Confusion matrices

One of the most comprehensive ways to represent the result of evaluating binary classification is using confusion matrices. Let's inspect the predictions of `LogisticRegression` from the previous section using the `confusion_matrix` function. We already stored the predictions on the test set in `pred_logreg`:

In[41]:

```
from sklearn.metrics import confusion_matrix

confusion = confusion_matrix(y_test, pred_logreg)
print("Confusion matrix:\n{}".format(confusion))
```

Out[41]:

```
Confusion matrix:
[[401  2]
 [ 8  39]]
```

The output of `confusion_matrix` is a two-by-two array, where the rows correspond to the true classes and the columns correspond to the predicted classes. Each entry counts how often a sample that belongs to the class corresponding to the row (here, “not nine” and “nine”) was classified as the class corresponding to the column. The following plot (Figure 5-10) illustrates this meaning:

In[42]:

```
mlearn.plots.plot_confusion_matrix_illustration()
```

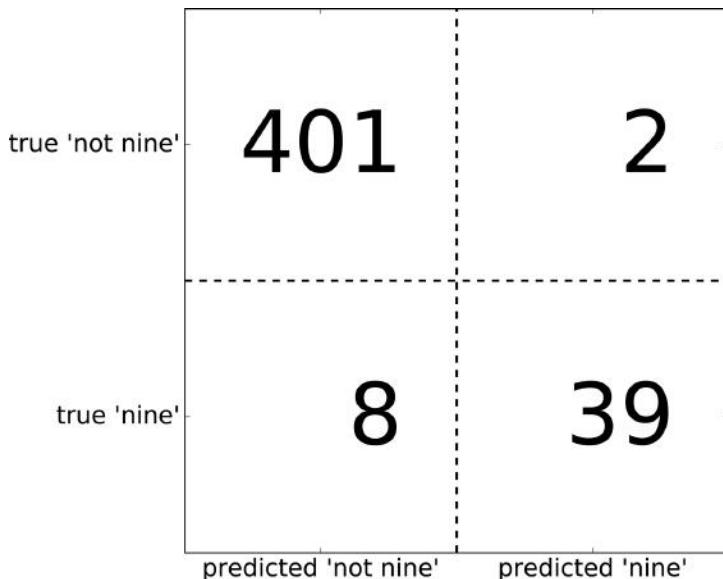


Figure 5-10. Confusion matrix of the “nine vs. rest” classification task

Entries on the main diagonal³ of the confusion matrix correspond to correct classifications, while other entries tell us how many samples of one class got mistakenly classified as another class.

If we declare “a nine” the positive class, we can relate the entries of the confusion matrix with the terms *false positive* and *false negative* that we introduced earlier. To complete the picture, we call correctly classified samples belonging to the positive class *true positives* and correctly classified samples belonging to the negative class *true negatives*. These terms are usually abbreviated FP, FN, TP, and TN and lead to the following interpretation for the confusion matrix (Figure 5-11):

In[43]:

```
mglearn.plots.plot_binary_confusion_matrix()
```

	TN	FP
negative class		
	FN	TP
positive class		
	predicted negative	predicted positive

Figure 5-11. Confusion matrix for binary classification

Now let’s use the confusion matrix to compare the models we fitted earlier (the two dummy models, the decision tree, and the logistic regression):

In[44]:

```
print("Most frequent class:")
print(confusion_matrix(y_test, pred_most_frequent))
print("\nDummy model:")
print(confusion_matrix(y_test, pred_dummy))
print("\nDecision tree:")
print(confusion_matrix(y_test, pred_tree))
print("\nLogistic Regression")
print(confusion_matrix(y_test, pred_logreg))
```

³ The main diagonal of a two-dimensional array or matrix A is A[i, i].

Out[44]:

Most frequent class:

```
[[403  0]
 [ 47  0]]
```

Dummy model:

```
[[361 42]
 [ 43 43]]
```

Decision tree:

```
[[390 13]
 [ 24 23]]
```

Logistic Regression

```
[[401 2]
 [ 8 39]]
```

Looking at the confusion matrix, it is quite clear that something is wrong with `pred_most_frequent`, because it always predicts the same class. `pred_dummy`, on the other hand, has a very small number of true positives (4), particularly compared to the number of false negatives and false positives—there are many more false positives than true positives! The predictions made by the decision tree make much more sense than the dummy predictions, even though the accuracy was nearly the same. Finally, we can see that logistic regression does better than `pred_tree` in all aspects: it has more true positives and true negatives while having fewer false positives and false negatives. From this comparison, it is clear that only the decision tree and the logistic regression give reasonable results, and that the logistic regression works better than the tree on all accounts. However, inspecting the full confusion matrix is a bit cumbersome, and while we gained a lot of insight from looking at all aspects of the matrix, the process was very manual and qualitative. There are several ways to summarize the information in the confusion matrix, which we will discuss next.

Relation to accuracy. We already saw one way to summarize the result in the confusion matrix—by computing accuracy, which can be expressed as:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

In other words, accuracy is the number of correct predictions (TP and TN) divided by the number of all samples (all entries of the confusion matrix summed up).

Precision, recall, and f-score. There are several other ways to summarize the confusion matrix, with the most common ones being precision and recall. *Precision* measures how many of the samples predicted as positive are actually positive:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Precision is used as a performance metric when the goal is to limit the number of false positives. As an example, imagine a model for predicting whether a new drug will be effective in treating a disease in clinical trials. Clinical trials are notoriously expensive, and a pharmaceutical company will only want to run an experiment if it is very sure that the drug will actually work. Therefore, it is important that the model does not produce many false positives—in other words, that it has a high precision. Precision is also known as *positive predictive value* (PPV).

Recall, on the other hand, measures how many of the positive samples are captured by the positive predictions:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Recall is used as performance metric when we need to identify all positive samples; that is, when it is important to avoid false negatives. The cancer diagnosis example from earlier in this chapter is a good example for this: it is important to find all people that are sick, possibly including healthy patients in the prediction. Other names for recall are *sensitivity*, *hit rate*, or *true positive rate* (TPR).

There is a trade-off between optimizing recall and optimizing precision. You can trivially obtain a perfect recall if you predict all samples to belong to the positive class—there will be no false negatives, and no true negatives either. However, predicting all samples as positive will result in many false positives, and therefore the precision will be very low. On the other hand, if you find a model that predicts only the single data point it is most sure about as positive and the rest as negative, then precision will be perfect (assuming this data point is in fact positive), but recall will be very bad.



Precision and recall are only two of many classification measures derived from TP, FP, TN, and FN. You can find a great summary of all the measures [on Wikipedia](#). In the machine learning community, precision and recall are arguably the most commonly used measures for binary classification, but other communities might use other related metrics.

So, while precision and recall are very important measures, looking at only one of them will not provide you with the full picture. One way to summarize them is the *f-score* or *f-measure*, which is with the harmonic mean of precision and recall:

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

This particular variant is also known as the f_1 -score. As it takes precision and recall into account, it can be a better measure than accuracy on imbalanced binary classification datasets. Let's run it on the predictions for the “nine vs. rest” dataset that we computed earlier. Here, we will assume that the “nine” class is the positive class (it is labeled as `True` while the rest is labeled as `False`), so the positive class is the minority class:

In[45]:

```
from sklearn.metrics import f1_score
print("f1 score most frequent: {:.2f}".format(
    f1_score(y_test, pred_most_frequent)))
print("f1 score dummy: {:.2f}".format(f1_score(y_test, pred_dummy)))
print("f1 score tree: {:.2f}".format(f1_score(y_test, pred_tree)))
print("f1 score logistic regression: {:.2f}".format(
    f1_score(y_test, pred_logreg)))
```

Out[45]:

```
f1 score most frequent: 0.00
f1 score dummy: 0.10
f1 score tree: 0.55
f1 score logistic regression: 0.89
```

We can note two things here. First, we get an error message for the `most_frequent` prediction, as there were no predictions of the positive class (which makes the denominator in the f -score zero). Also, we can see a pretty strong distinction between the dummy predictions and the tree predictions, which wasn't clear when looking at accuracy alone. Using the f -score for evaluation, we summarized the predictive performance again in one number. However, the f -score seems to capture our intuition of what makes a good model much better than accuracy did. A disadvantage of the f -score, however, is that it is harder to interpret and explain than accuracy.

If we want a more comprehensive summary of precision, recall, and f_1 -score, we can use the `classification_report` convenience function to compute all three at once, and print them in a nice format:

In[46]:

```
from sklearn.metrics import classification_report
print(classification_report(y_test, pred_most_frequent,
                            target_names=["not nine", "nine"]))
```

Out[46]:

	precision	recall	f1-score	support
not nine	0.90	1.00	0.94	403
nine	0.00	0.00	0.00	47
avg / total	0.80	0.90	0.85	450

The `classification_report` function produces one line per class (here, `True` and `False`) and reports precision, recall, and f -score with this class as the positive class. Before, we assumed the minority “nine” class was the positive class. If we change the positive class to “not nine,” we can see from the output of `classification_report` that we obtain an f -score of 0.94 with the `most_frequent` model. Furthermore, for the “not nine” class we have a recall of 1, as we classified all samples as “not nine.” The last column next to the f -score provides the *support* of each class, which simply means the number of samples in this class according to the ground truth.

The last row in the classification report shows a weighted (by the number of samples in the class) average of the numbers for each class. Here are two more reports, one for the dummy classifier and one for the logistic regression:

In[47]:

```
print(classification_report(y_test, pred_dummy,
                            target_names=["not nine", "nine"]))
```

Out[47]:

	precision	recall	f1-score	support
not nine	0.90	0.92	0.91	403
nine	0.11	0.09	0.10	47
avg / total	0.81	0.83	0.82	450

In[48]:

```
print(classification_report(y_test, pred_logreg,
                            target_names=["not nine", "nine"]))
```

Out[48]:

	precision	recall	f1-score	support
not nine	0.98	1.00	0.99	403
nine	0.95	0.83	0.89	47
avg / total	0.98	0.98	0.98	450

As you may notice when looking at the reports, the differences between the dummy models and a very good model are not as clear any more. Picking which class is declared the positive class has a big impact on the metrics. While the f -score for the dummy classification is 0.13 (vs. 0.89 for the logistic regression) on the “nine” class, for the “not nine” class it is 0.90 vs. 0.99, which both seem like reasonable results. Looking at all the numbers together paints a pretty accurate picture, though, and we can clearly see the superiority of the logistic regression model.

Taking uncertainty into account

The confusion matrix and the classification report provide a very detailed analysis of a particular set of predictions. However, the predictions themselves already threw away a lot of information that is contained in the model. As we discussed in [Chapter 2](#), most classifiers provide a `decision_function` or a `predict_proba` method to assess degrees of certainty about predictions. Making predictions can be seen as thresholding the output of `decision_function` or `predict_proba` at a certain fixed point—in binary classification we use 0 for the decision function and 0.5 for `predict_proba`.

The following is an example of an imbalanced binary classification task, with 400 points in the negative class classified against 50 points in the positive class. The training data is shown on the left in [Figure 5-12](#). We train a kernel SVM model on this data, and the plots to the right of the training data illustrate the values of the decision function as a heat map. You can see a black circle in the plot in the top center, which denotes the threshold of the `decision_function` being exactly zero. Points inside this circle will be classified as the positive class, and points outside as the negative class:

In[49]:

```
from mglearn.datasets import make_blobs
X, y = make_blobs(n_samples=(400, 50), centers=2, cluster_std=[7.0, 2],
                  random_state=22)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
```

In[50]:

```
mglearn.plots.plot_decision_threshold()
```

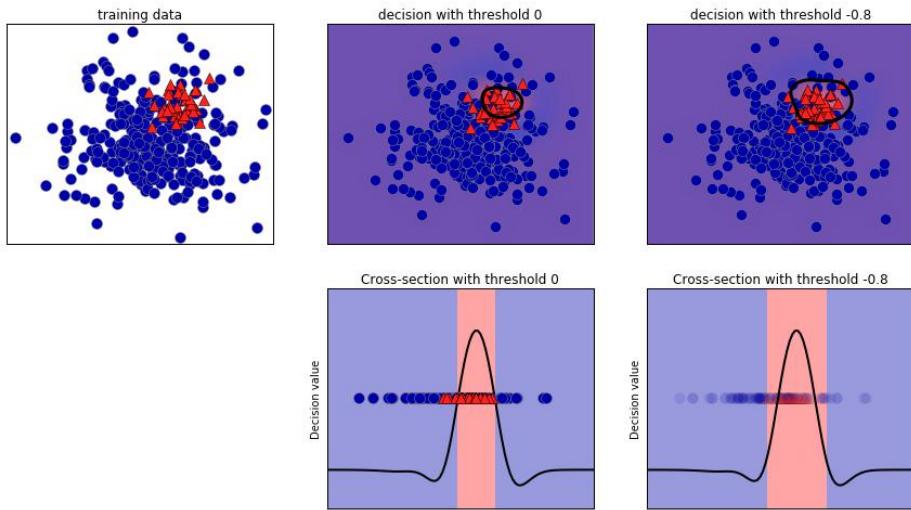


Figure 5-12. Heatmap of the decision function and the impact of changing the decision threshold

We can use the `classification_report` function to evaluate precision and recall for both classes:

In[51]:

```
print(classification_report(y_test, svc.predict(X_test)))
```

Out[51]:

	precision	recall	f1-score	support
0	0.97	0.89	0.93	104
1	0.35	0.67	0.46	9
avg / total	0.92	0.88	0.89	113

For class 1, we get a fairly small recall, and precision is mixed. Because class 0 is so much larger, the classifier focuses on getting class 0 right, and not the smaller class 1.

Let's assume in our application it is more important to have a high recall for class 1, as in the cancer screening example earlier. This means we are willing to risk more false positives (false class 1) in exchange for more true positives (which will increase the recall). The predictions generated by `svc.predict` really do not fulfill this requirement, but we can adjust the predictions to focus on a higher recall of class 1 by changing the decision threshold away from 0. By default, points with a `decision_function` value greater than 0 will be classified as class 1. We want *more* points to be classified as class 1, so we need to *decrease* the threshold:

In[52]:

```
y_pred_lower_threshold = svc.decision_function(X_test) > -8
```

Let's look at the classification report for this prediction:

In[53]:

```
print(classification_report(y_test, y_pred_lower_threshold))
```

Out[53]:

	precision	recall	f1-score	support
0	1.00	0.82	0.90	104
1	0.32	1.00	0.49	9
avg / total	0.95	0.83	0.87	113

As expected, the recall of class 1 went up, and the precision went down. We are now classifying a larger region of space as class 1, as illustrated in the top-right panel of [Figure 5-12](#). If you value precision over recall or the other way around, or your data is heavily imbalanced, changing the decision threshold is the easiest way to obtain better results. As the `decision_function` can have arbitrary ranges, it is hard to provide a rule of thumb regarding how to pick a threshold.



If you do set a threshold, you need to be careful not to do so using the test set. As with any other parameter, setting a decision threshold on the test set is likely to yield overly optimistic results. Use a validation set or cross-validation instead.

Picking a threshold for models that implement the `predict_proba` method can be easier, as the output of `predict_proba` is on a fixed 0 to 1 scale, and models probabilities. By default, the threshold of 0.5 means that if the model is more than 50% “sure” that a point is of the positive class, it will be classified as such. Increasing the threshold means that the model needs to be more confident to make a positive decision (and less confident to make a negative decision). While working with probabilities may be more intuitive than working with arbitrary thresholds, not all models provide realistic models of uncertainty (a `DecisionTree` that is grown to its full depth is always 100% sure of its decisions, even though it might often be wrong). This relates to the concept of *calibration*: a calibrated model is a model that provides an accurate measure of its uncertainty. Discussing calibration in detail is beyond the scope of this book, but you can find more details in the paper [“Predicting Good Probabilities with Supervised Learning”](#) by Alexandru Niculescu-Mizil and Rich Caruana.

Precision-recall curves and ROC curves

As we just discussed, changing the threshold that is used to make a classification decision in a model is a way to adjust the trade-off of precision and recall for a given classifier. Maybe you want to miss less than 10% of positive samples, meaning a desired recall of 90%. This decision depends on the application, and it should be driven by business goals. Once a particular goal is set—say, a particular recall or precision value for a class—a threshold can be set appropriately. It is always possible to set a threshold to fulfill a particular target, like 90% recall. The hard part is to develop a model that still has reasonable precision with this threshold—if you classify everything as positive, you will have 100% recall, but your model will be useless.

Setting a requirement on a classifier like 90% recall is often called setting the *operating point*. Fixing an operating point is often helpful in business settings to make performance guarantees to customers or other groups inside your organization.

Often, when developing a new model, it is not entirely clear what the operating point will be. For this reason, and to understand a modeling problem better, it is instructive to look at all possible thresholds, or all possible trade-offs of precision and recalls *at once*. This is possible using a tool called the *precision-recall curve*. You can find the function to compute the precision-recall curve in the `sklearn.metrics` module. It needs the ground truth labeling and predicted uncertainties, created via either `decision_function` or `predict_proba`:

In[54]:

```
from sklearn.metrics import precision_recall_curve
precision, recall, thresholds = precision_recall_curve(
    y_test, svc.decision_function(X_test))
```

The `precision_recall_curve` function returns a list of precision and recall values for all possible thresholds (all values that appear in the decision function) in sorted order, so we can plot a curve, as seen in [Figure 5-13](#):

In[55]:

```
# Use more data points for a smoother curve
X, y = make_blobs(n_samples=(4000, 500), centers=2, cluster_std=[7.0, 2],
                   random_state=22)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
svc = SVC(gamma=.05).fit(X_train, y_train)
precision, recall, thresholds = precision_recall_curve(
    y_test, svc.decision_function(X_test))
# find threshold closest to zero
close_zero = np.argmin(np.abs(thresholds))
plt.plot(precision[close_zero], recall[close_zero], 'o', markersize=10,
          label="threshold zero", fillstyle="none", c='k', mew=2)

plt.plot(precision, recall, label="precision recall curve")
plt.xlabel("Precision")
plt.ylabel("Recall")
```

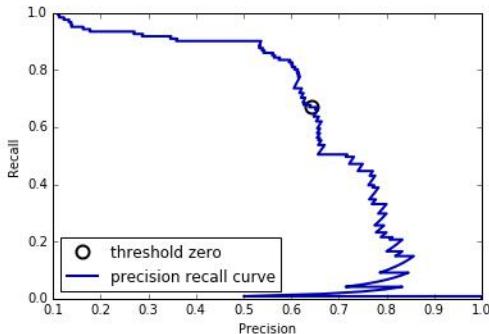


Figure 5-13. Precision recall curve for SVC($\text{gamma}=0.05$)

Each point along the curve in Figure 5-13 corresponds to a possible threshold of the `decision_function`. We can see, for example, that we can achieve a recall of 0.4 at a precision of about 0.75. The black circle marks the point that corresponds to a threshold of 0, the default threshold for `decision_function`. This point is the trade-off that is chosen when calling the `predict` method.

The closer a curve stays to the upper-right corner, the better the classifier. A point at the upper right means high precision *and* high recall for the same threshold. The curve starts at the top-left corner, corresponding to a very low threshold, classifying everything as the positive class. Raising the threshold moves the curve toward higher precision, but also lower recall. Raising the threshold more and more, we get to a situation where most of the points classified as being positive are true positives, leading to a very high precision but lower recall. The more the model keeps recall high as precision goes up, the better.

Looking at this particular curve a bit more, we can see that with this model it is possible to get a precision of up to around 0.5 with very high recall. If we want a much higher precision, we have to sacrifice a lot of recall. In other words, on the left the curve is relatively flat, meaning that recall does not go down a lot when we require increased precision. For precision greater than 0.5, each gain in precision costs us a lot of recall.

Different classifiers can work well in different parts of the curve—that is, at different operating points. Let's compare the SVM we trained to a random forest trained on the same dataset. The `RandomForestClassifier` doesn't have a `decision_function`, only `predict_proba`. The `precision_recall_curve` function expects as its second argument a certainty measure for the positive class (class 1), so we pass the probability of a sample being class 1—that is, `rf.predict_proba(X_test)[:, 1]`. The default threshold for `predict_proba` in binary classification is 0.5, so this is the point we marked on the curve (see Figure 5-14):

In[56]:

```
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(n_estimators=100, random_state=0, max_features=2)
rf.fit(X_train, y_train)

# RandomForestClassifier has predict_proba, but not decision_function
precision_rf, recall_rf, thresholds_rf = precision_recall_curve(
    y_test, rf.predict_proba(X_test)[:, 1])

plt.plot(precision, recall, label="svc")

plt.plot(precision[close_zero], recall[close_zero], 'o', markersize=10,
         label="threshold zero svc", fillstyle="none", c='k', mew=2)

plt.plot(precision_rf, recall_rf, label="rf")

close_default_rf = np.argmin(np.abs(thresholds_rf - 0.5))
plt.plot(precision_rf[close_default_rf], recall_rf[close_default_rf], '^', c='k',
         markersize=10, label="threshold 0.5 rf", fillstyle="none", mew=2)

plt.xlabel("Precision")
plt.ylabel("Recall")
plt.legend(loc="best")
```

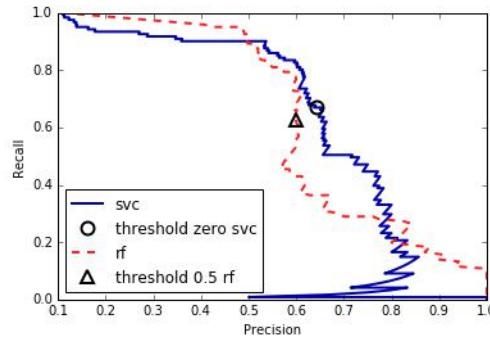


Figure 5-14. Comparing precision recall curves of SVM and random forest

From the comparison plot we can see that the random forest performs better at the extremes, for very high recall or very high precision requirements. Around the middle (approximately precision=0.7), the SVM performs better. If we only looked at the f_1 -score to compare overall performance, we would have missed these subtleties. The f_1 -score only captures one point on the precision-recall curve, the one given by the default threshold:

In[57]:

```
print("f1_score of random forest: {:.3f}".format(
    f1_score(y_test, rf.predict(X_test))))
print("f1_score of svc: {:.3f}".format(f1_score(y_test, svc.predict(X_test))))
```

Out[57]:

```
f1_score of random forest: 0.610
f1_score of svc: 0.656
```

Comparing two precision-recall curves provides a lot of detailed insight, but is a fairly manual process. For automatic model comparison, we might want to summarize the information contained in the curve, without limiting ourselves to a particular threshold or operating point. One particular way to summarize the precision-recall curve is by computing the integral or area under the curve of the precision-recall curve, also known as the *average precision*.⁴ You can use the `average_precision_score` function to compute the average precision. Because we need to compute the ROC curve and consider multiple thresholds, the result of `decision_function` or `predict_proba` needs to be passed to `average_precision_score`, not the result of `predict`:

In[58]:

```
from sklearn.metrics import average_precision_score
ap_rf = average_precision_score(y_test, rf.predict_proba(X_test)[:, 1])
ap_svc = average_precision_score(y_test, svc.decision_function(X_test))
print("Average precision of random forest: {:.3f}".format(ap_rf))
print("Average precision of svc: {:.3f}".format(ap_svc))
```

Out[58]:

```
Average precision of random forest: 0.666
Average precision of svc: 0.663
```

When averaging over all possible thresholds, we see that the random forest and SVC perform similarly well, with the random forest even slightly ahead. This is quite different from the result we got from `f1_score` earlier. Because average precision is the area under a curve that goes from 0 to 1, average precision always returns a value between 0 (worst) and 1 (best). The average precision of a classifier that assigns `decision_function` at random is the fraction of positive samples in the dataset.

Receiver operating characteristics (ROC) and AUC

There is another tool that is commonly used to analyze the behavior of classifiers at different thresholds: the *receiver operating characteristics curve*, or *ROC curve* for short. Similar to the precision-recall curve, the ROC curve considers all possible

⁴ There are some minor technical differences between the area under the precision-recall curve and average precision. However, this explanation conveys the general idea.

thresholds for a given classifier, but instead of reporting precision and recall, it shows the *false positive rate* (FPR) against the *true positive rate* (TPR). Recall that the true positive rate is simply another name for recall, while the false positive rate is the fraction of false positives out of all negative samples:

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

The ROC curve can be computed using the `roc_curve` function (see Figure 5-15):

In[59]:

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_test, svc.decision_function(X_test))

plt.plot(fpr, tpr, label="ROC Curve")
plt.xlabel("FPR")
plt.ylabel("TPR (recall)")
# find threshold closest to zero
close_zero = np.argmin(np.abs(thresholds))
plt.plot(fpr[close_zero], tpr[close_zero], 'o', markersize=10,
         label="threshold zero", fillstyle="none", c='k', mew=2)
plt.legend(loc=4)
```

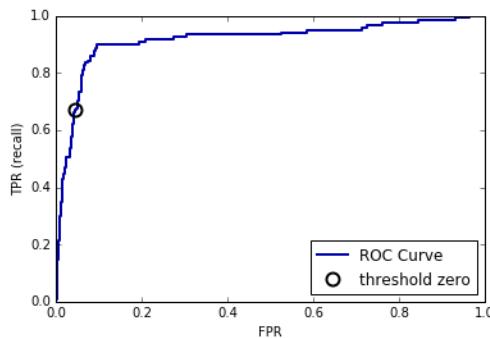


Figure 5-15. ROC curve for SVM

For the ROC curve, the ideal curve is close to the top left: you want a classifier that produces a *high recall* while keeping a *low false positive rate*. Compared to the default threshold of 0, the curve shows that we can achieve a significantly higher recall (around 0.9) while only increasing the FPR slightly. The point closest to the top left might be a better operating point than the one chosen by default. Again, be aware that choosing a threshold should not be done on the test set, but on a separate validation set.

You can find a comparison of the random forest and the SVM using ROC curves in Figure 5-16:

In[60]:

```
from sklearn.metrics import roc_curve
fpr_rf, tpr_rf, thresholds_rf = roc_curve(y_test, rf.predict_proba(X_test)[:, 1])

plt.plot(fpr, tpr, label="ROC Curve SVC")
plt.plot(fpr_rf, tpr_rf, label="ROC Curve RF")

plt.xlabel("FPR")
plt.ylabel("TPR (recall)")
plt.plot(fpr[close_zero], tpr[close_zero], 'o', markersize=10,
         label="threshold zero SVC", fillstyle="none", c='k', mew=2)
close_default_rf = np.argmin(np.abs(thresholds_rf - 0.5))
plt.plot(fpr_rf[close_default_rf], tpr_rf[close_default_rf], '^', markersize=10,
         label="threshold 0.5 RF", fillstyle="none", c='k', mew=2)

plt.legend(loc=4)
```

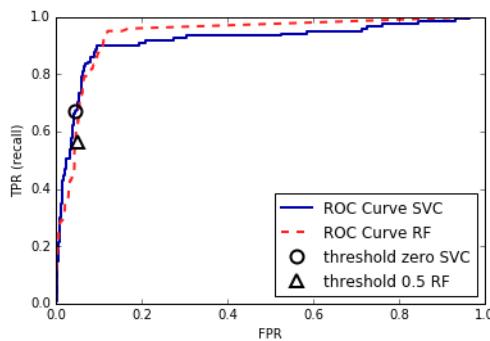


Figure 5-16. Comparing ROC curves for SVM and random forest

As for the precision-recall curve, we often want to summarize the ROC curve using a single number, the area under the curve (this is commonly just referred to as the AUC, and it is understood that the curve in question is the ROC curve). We can compute the area under the ROC curve using the `roc_auc_score` function:

In[61]:

```
from sklearn.metrics import roc_auc_score
rf_auc = roc_auc_score(y_test, rf.predict_proba(X_test)[:, 1])
svc_auc = roc_auc_score(y_test, svc.decision_function(X_test))
print("AUC for Random Forest: {:.3f}".format(rf_auc))
print("AUC for SVC: {:.3f}".format(svc_auc))
```

Out[61]:

```
AUC for Random Forest: 0.937
AUC for SVC: 0.916
```

Comparing the random forest and SVM using the AUC score, we find that the random forest performs quite a bit better than the SVM. Recall that because average precision is the area under a curve that goes from 0 to 1, average precision always returns a value between 0 (worst) and 1 (best). Predicting randomly always produces an AUC of 0.5, no matter how imbalanced the classes in a dataset are. This makes AUC a much better metric for imbalanced classification problems than accuracy. The AUC can be interpreted as evaluating the *ranking* of positive samples. It's equivalent to the probability that a randomly picked point of the positive class will have a higher score according to the classifier than a randomly picked point from the negative class. So, a perfect AUC of 1 means that all positive points have a higher score than all negative points. For classification problems with imbalanced classes, using AUC for model selection is often much more meaningful than using accuracy.

Let's go back to the problem we studied earlier of classifying all nines in the digits dataset versus all other digits. We will classify the dataset with an SVM with three different settings of the kernel bandwidth, `gamma` (see Figure 5-17):

In[62]:

```
y = digits.target == 9

X_train, X_test, y_train, y_test = train_test_split(
    digits.data, y, random_state=0)

plt.figure()

for gamma in [1, 0.05, 0.01]:
    svc = SVC(gamma=gamma).fit(X_train, y_train)
    accuracy = svc.score(X_test, y_test)
    auc = roc_auc_score(y_test, svc.decision_function(X_test))
    fpr, tpr, _ = roc_curve(y_test, svc.decision_function(X_test))
    print("gamma = {:.2f} accuracy = {:.2f} AUC = {:.2f}".format(
        gamma, accuracy, auc))
    plt.plot(fpr, tpr, label="gamma={:.3f}".format(gamma))
plt.xlabel("FPR")
plt.ylabel("TPR")
plt.xlim(-0.01, 1)
plt.ylim(0, 1.02)
plt.legend(loc="best")
```

Out[62]:

```
gamma = 1.00  accuracy = 0.90  AUC = 0.50
gamma = 0.05  accuracy = 0.90  AUC = 0.90
gamma = 0.01  accuracy = 0.90  AUC = 1.00
```



Figure 5-17. Comparing ROC curves of SVMs with different settings of gamma

The accuracy of all three settings of `gamma` is the same, 90%. This might be the same as chance performance, or it might not. Looking at the AUC and the corresponding curve, however, we see a clear distinction between the three models. With `gamma=1.0`, the AUC is actually at chance level, meaning that the output of the `decision_func`tion is as good as random. With `gamma=0.05`, performance drastically improves to an AUC of 0.5. Finally, with `gamma=0.01`, we get a perfect AUC of 1.0. That means that all positive points are ranked higher than all negative points according to the decision function. In other words, with the right threshold, this model can classify the data perfectly!¹⁵ Knowing this, we can adjust the threshold on this model and obtain great predictions. If we had only used accuracy, we would never have discovered this.

For this reason, we highly recommend using AUC when evaluating models on imbalanced data. Keep in mind that AUC does not make use of the default threshold, though, so adjusting the decision threshold might be necessary to obtain useful classification results from a model with a high AUC.

Metrics for Multiclass Classification

Now that we have discussed evaluation of binary classification tasks in depth, let's move on to metrics to evaluate multiclass classification. Basically, all metrics for multiclass classification are derived from binary classification metrics, but averaged

⁵ Looking at the curve for `gamma=0.01` in detail, you can see a small kink close to the top left. That means that at least one point was not ranked correctly. The AUC of 1.0 is a consequence of rounding to the second decimal point.

over all classes. Accuracy for multiclass classification is again defined as the fraction of correctly classified examples. And again, when classes are imbalanced, accuracy is not a great evaluation measure. Imagine a three-class classification problem with 85% of points belonging to class A, 10% belonging to class B, and 5% belonging to class C. What does being 85% accurate mean on this dataset? In general, multiclass classification results are harder to understand than binary classification results. Apart from accuracy, common tools are the confusion matrix and the classification report we saw in the binary case in the previous section. Let's apply these two detailed evaluation methods on the task of classifying the 10 different handwritten digits in the `digits` dataset:

In[63]:

```
from sklearn.metrics import accuracy_score
X_train, X_test, y_train, y_test = train_test_split(
    digits.data, digits.target, random_state=0)
lr = LogisticRegression().fit(X_train, y_train)
pred = lr.predict(X_test)
print("Accuracy: {:.3f}".format(accuracy_score(y_test, pred)))
print("Confusion matrix:\n{}".format(confusion_matrix(y_test, pred)))
```

Out[63]:

```
Accuracy: 0.953
Confusion matrix:
[[37  0  0  0  0  0  0  0  0  0]
 [ 0 39  0  0  0  0  2  0  2  0]
 [ 0  0 41  3  0  0  0  0  0  0]
 [ 0  0  1 43  0  0  0  0  0  1]
 [ 0  0  0  0 38  0  0  0  0  0]
 [ 0  1  0  0  0 47  0  0  0  0]
 [ 0  0  0  0  0  0 52  0  0  0]
 [ 0  1  0  1  1  0  0 45  0  0]
 [ 0  3  1  0  0  0  0  0 43  1]
 [ 0  0  0  1  0  1  0  0  1 44]]
```

The model has an accuracy of 95.3%, which already tells us that we are doing pretty well. The confusion matrix provides us with some more detail. As for the binary case, each row corresponds to a true label, and each column corresponds to a predicted label. You can find a visually more appealing plot in [Figure 5-18](#):

In[64]:

```
scores_image = mglearn.tools.heatmap(
    confusion_matrix(y_test, pred), xlabel='Predicted label',
    ylabel='True label', xticklabels=digits.target_names,
    yticklabels=digits.target_names, cmap=plt.cm.gray_r, fmt="%d")
plt.title("Confusion matrix")
plt.gca().invert_yaxis()
```



Figure 5-18. Confusion matrix for the 10-digit classification task

For the first class, the digit 0, there are 37 samples in the class, and all of these samples were classified as class 0 (there are no false negatives for class 0). We can see that because all other entries in the first row of the confusion matrix are 0. We can also see that no other digits were mistakenly classified as 0, because all other entries in the first column of the confusion matrix are 0 (there are no false positives for class 0). Some digits were confused with others, though—for example, the digit 2 (third row), three of which were classified as the digit 3 (fourth column). There was also one digit 3 that was classified as 2 (third column, fourth row) and one digit 8 that was classified as 2 (third column, fourth row).

With the `classification_report` function, we can compute the precision, recall, and f -score for each class:

In[65]:

```
print(classification_report(y_test, pred))
```

Out[65]:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	37
1	0.89	0.91	0.90	43
2	0.95	0.93	0.94	44
3	0.90	0.96	0.92	45
4	0.97	1.00	0.99	38
5	0.98	0.98	0.98	48
6	0.96	1.00	0.98	52
7	1.00	0.94	0.97	48
8	0.93	0.90	0.91	48
9	0.96	0.94	0.95	47
avg / total	0.95	0.95	0.95	450

Unsurprisingly, precision and recall are a perfect 1 for class 0, as there are no confusions with this class. For class 7, on the other hand, precision is 1 because no other class was mistakenly classified as 7, while for class 6, there are no false negatives, so the recall is 1. We can also see that the model has particular difficulties with classes 8 and 3.

The most commonly used metric for imbalanced datasets in the multiclass setting is the multiclass version of the f -score. The idea behind the multiclass f -score is to compute one binary f -score per class, with that class being the positive class and the other classes making up the negative classes. Then, these per-class f -scores are averaged using one of the following strategies:

- "macro" averaging computes the unweighted per-class f -scores. This gives equal weight to all classes, no matter what their size is.
- "weighted" averaging computes the mean of the per-class f -scores, weighted by their support. This is what is reported in the classification report.
- "micro" averaging computes the total number of false positives, false negatives, and true positives over all classes, and then computes precision, recall, and f -score using these counts.

If you care about each *sample* equally much, it is recommended to use the "micro" average f_1 -score; if you care about each *class* equally much, it is recommended to use the "macro" average f_1 -score:

In[66]:

```
print("Micro average f1 score: {:.3f}".format
      (f1_score(y_test, pred, average="micro")))
print("Macro average f1 score: {:.3f}".format
      (f1_score(y_test, pred, average="macro")))
```

Out[66]:

```
Micro average f1 score: 0.953
Macro average f1 score: 0.954
```

Regression Metrics

Evaluation for regression can be done in similar detail as we did for classification—for example, by analyzing overpredicting the target versus underpredicting the target. However, in most applications we've seen, using the default R^2 used in the `score` method of all regressors is enough. Sometimes business decisions are made on the basis of mean squared error or mean absolute error, which might give incentive to tune models using these metrics. In general, though, we have found R^2 to be a more intuitive metric to evaluate regression models.

Using Evaluation Metrics in Model Selection

We have discussed many evaluation methods in detail, and how to apply them given the ground truth and a model. However, we often want to use metrics like AUC in model selection using `GridSearchCV` or `cross_val_score`. Luckily `scikit-learn` provides a very simple way to achieve this, via the `scoring` argument that can be used in both `GridSearchCV` and `cross_val_score`. You can simply provide a string describing the evaluation metric you want to use. Say, for example, we want to evaluate the SVM classifier on the “nine vs. rest” task on the `digits` dataset, using the AUC score. Changing the score from the default (accuracy) to AUC can be done by providing “`roc_auc`” as the `scoring` parameter:

In[67]:

```
# default scoring for classification is accuracy
print("Default scoring: {}".format(
    cross_val_score(SVC(), digits.data, digits.target == 9)))
# providing scoring="accuracy" doesn't change the results
explicit_accuracy = cross_val_score(SVC(), digits.data, digits.target == 9,
                                      scoring="accuracy")
print("Explicit accuracy scoring: {}".format(explicit_accuracy))
roc_auc = cross_val_score(SVC(), digits.data, digits.target == 9,
                           scoring="roc_auc")
print("AUC scoring: {}".format(roc_auc))
```

Out[67]:

```
Default scoring: [ 0.9  0.9  0.9]
Explicit accuracy scoring: [ 0.9  0.9  0.9]
AUC scoring: [ 0.994  0.99   0.996]
```

Similarly, we can change the metric used to pick the best parameters in `GridSearchCV`:

In[68]:

```
X_train, X_test, y_train, y_test = train_test_split(
    digits.data, digits.target == 9, random_state=0)

# we provide a somewhat bad grid to illustrate the point:
param_grid = {'gamma': [0.0001, 0.01, 0.1, 1, 10]}
# using the default scoring of accuracy:
grid = GridSearchCV(SVC(), param_grid=param_grid)
grid.fit(X_train, y_train)
print("Grid-Search with accuracy")
print("Best parameters:", grid.best_params_)
print("Best cross-validation score (accuracy): {:.3f}".format(grid.best_score_))
print("Test set AUC: {:.3f}".format(
    roc_auc_score(y_test, grid.decision_function(X_test))))
print("Test set accuracy: {:.3f}".format(grid.score(X_test, y_test)))
```

Out[68]:

```
Grid-Search with accuracy
Best parameters: {'gamma': 0.0001}
Best cross-validation score (accuracy)): 0.970
Test set AUC: 0.992
Test set accuracy: 0.973
```

In[69]:

```
# using AUC scoring instead:
grid = GridSearchCV(SVC(), param_grid=param_grid, scoring="roc_auc")
grid.fit(X_train, y_train)
print("\nGrid-Search with AUC")
print("Best parameters:", grid.best_params_)
print("Best cross-validation score (AUC): {:.3f}".format(grid.best_score_))
print("Test set AUC: {:.3f}".format(
    roc_auc_score(y_test, grid.decision_function(X_test))))
print("Test set accuracy: {:.3f}".format(grid.score(X_test, y_test)))
```

Out[69]:

```
Grid-Search with AUC
Best parameters: {'gamma': 0.01}
Best cross-validation score (AUC): 0.997
Test set AUC: 1.000
Test set accuracy: 1.000
```

When using accuracy, the parameter `gamma=0.0001` is selected, while `gamma=0.01` is selected when using AUC. The cross-validation accuracy is consistent with the test set accuracy in both cases. However, using AUC found a better parameter setting in terms of AUC and even in terms of accuracy.⁶

The most important values for the `scoring` parameter for classification are `accuracy` (the default); `roc_auc` for the area under the ROC curve; `average_precision` for the area under the precision-recall curve; `f1`, `f1_macro`, `f1_micro`, and `f1_weighted` for the binary f_1 -score and the different weighted variants. For regression, the most commonly used values are `r2` for the R^2 score, `mean_squared_error` for mean squared error, and `mean_absolute_error` for mean absolute error. You can find a full list of supported arguments in the [documentation](#) or by looking at the `SCORER` dictionary defined in the `metrics.scorer` module:

⁶ Finding a higher-accuracy solution using AUC is likely a consequence of accuracy being a bad measure of model performance on imbalanced data.

In[70]:

```
from sklearn.metrics.scorer import SCORERS
print("Available scorers:\n{}".format(sorted(SCORERS.keys()))))
```

Out[70]:

```
Available scorers:
['accuracy', 'adjusted_rand_score', 'average_precision', 'f1', 'f1_macro',
 'f1_micro', 'f1_samples', 'f1_weighted', 'log_loss', 'mean_absolute_error',
 'mean_squared_error', 'median_absolute_error', 'precision', 'precision_macro',
 'precision_micro', 'precision_samples', 'precision_weighted', 'r2', 'recall',
 'recall_macro', 'recall_micro', 'recall_samples', 'recall_weighted', 'roc_auc']
```

Summary and Outlook

In this chapter we discussed cross-validation, grid search, and evaluation metrics, the cornerstones of evaluating and improving machine learning algorithms. The tools described in this chapter, together with the algorithms described in Chapters 2 and 3, are the bread and butter of every machine learning practitioner.

There are two particular points that we made in this chapter that warrant repeating, because they are often overlooked by new practitioners. The first has to do with cross-validation. Cross-validation or the use of a test set allow us to evaluate a machine learning model as it will perform in the future. However, if we use the test set or cross-validation to select a model or select model parameters, we “use up” the test data, and using the same data to evaluate how well our model will do in the future will lead to overly optimistic estimates. We therefore need to resort to a split into training data for model building, validation data for model and parameter selection, and test data for model evaluation. Instead of a simple split, we can replace each of these splits with cross-validation. The most commonly used form (as described earlier) is a training/test split for evaluation, and using cross-validation on the training set for model and parameter selection.

The second point has to do with the importance of the evaluation metric or scoring function used for model selection and model evaluation. The theory of how to make business decisions from the predictions of a machine learning model is somewhat beyond the scope of this book.⁷ However, it is rarely the case that the end goal of a machine learning task is building a model with a high accuracy. Make sure that the metric you choose to evaluate and select a model for is a good stand-in for what the model will actually be used for. In reality, classification problems rarely have balanced classes, and often false positives and false negatives have very different consequences.

⁷ We highly recommend Foster Provost and Tom Fawcett’s book *Data Science for Business* (O’Reilly) for more information on this topic.

Make sure you understand what these consequences are, and pick an evaluation metric accordingly.

The model evaluation and selection techniques we have described so far are the most important tools in a data scientist's toolbox. Grid search and cross-validation as we've described them in this chapter can only be applied to a single supervised model. We have seen before, however, that many models require preprocessing, and that in some applications, like the face recognition example in [Chapter 3](#), extracting a different representation of the data can be useful. In the next chapter, we will introduce the `Pipeline` class, which allows us to use grid search and cross-validation on these complex chains of algorithms.

Algorithm Chains and Pipelines

For many machine learning algorithms, the particular representation of the data that you provide is very important, as we discussed in [Chapter 4](#). This starts with scaling the data and combining features by hand and goes all the way to learning features using unsupervised machine learning, as we saw in [Chapter 3](#). Consequently, most machine learning applications require not only the application of a single algorithm, but the chaining together of many different processing steps and machine learning models. In this chapter, we will cover how to use the `Pipeline` class to simplify the process of building chains of transformations and models. In particular, we will see how we can combine `Pipeline` and `GridSearchCV` to search over parameters for all processing steps at once.

As an example of the importance of chaining models, we noticed that we can greatly improve the performance of a kernel SVM on the `cancer` dataset by using the `MinMaxScaler` for preprocessing. Here's code for splitting the data, computing the minimum and maximum, scaling the data, and training the SVM:

In[1]:

```
from sklearn.svm import SVC
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# load and split the data
cancer = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)

# compute minimum and maximum on the training data
scaler = MinMaxScaler().fit(X_train)
```

In[2]:

```
# rescale the training data
X_train_scaled = scaler.transform(X_train)

svm = SVC()
# learn an SVM on the scaled training data
svm.fit(X_train_scaled, y_train)
# scale the test data and score the scaled data
X_test_scaled = scaler.transform(X_test)
print("Test score: {:.2f}".format(svm.score(X_test_scaled, y_test)))
```

Out[2]:

```
Test score: 0.95
```

Parameter Selection with Preprocessing

Now let's say we want to find better parameters for SVC using GridSearchCV, as discussed in [Chapter 5](#). How should we go about doing this? A naive approach might look like this:

In[3]:

```
from sklearn.model_selection import GridSearchCV
# for illustration purposes only, don't use this code!
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
              'gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
grid = GridSearchCV(SVC(), param_grid=param_grid, cv=5)
grid.fit(X_train_scaled, y_train)
print("Best cross-validation accuracy: {:.2f}".format(grid.best_score_))
print("Best set score: {:.2f}".format(grid.score(X_test_scaled, y_test)))
print("Best parameters: ", grid.best_params_)
```

Out[3]:

```
Best cross-validation accuracy: 0.98
Best set score: 0.97
Best parameters: {'gamma': 1, 'C': 1}
```

Here, we ran the grid search over the parameters of SVC using the scaled data. However, there is a subtle catch in what we just did. When scaling the data, we used *all the data in the training set* to find out how to train it. We then use the *scaled training data* to run our grid search using cross-validation. For each split in the cross-validation, some part of the original training set will be declared the training part of the split, and some the test part of the split. The test part is used to measure what new data will look like to a model trained on the training part. However, we already used the information contained in the test part of the split, when scaling the data. Remember that the test part in each split in the cross-validation is part of the training set, and we used the information from the entire training set to find the right scaling of the data.

This is fundamentally different from how new data looks to the model. If we observe new data (say, in form of our test set), this data will not have been used to scale the training data, and it might have a different minimum and maximum than the training data. The following example (Figure 6-1) shows how the data processing during cross-validation and the final evaluation differ:

In[4]:

```
mglearn.plots.plot_improper_processing()
```



Figure 6-1. Data usage when preprocessing outside the cross-validation loop

So, the splits in the cross-validation no longer correctly mirror how new data will look to the modeling process. We already leaked information from these parts of the data into our modeling process. This will lead to overly optimistic results during cross-validation, and possibly the selection of suboptimal parameters.

To get around this problem, the splitting of the dataset during cross-validation should be done *before doing any preprocessing*. Any process that extracts knowledge from the dataset should only ever be applied to the training portion of the dataset, so any cross-validation should be the “outermost loop” in your processing.

To achieve this in scikit-learn with the `cross_val_score` function and the `GridSearchCV` function, we can use the `Pipeline` class. The `Pipeline` class is a class that allows “gluing” together multiple processing steps into a single scikit-learn estima-

tor. The `Pipeline` class itself has `fit`, `predict`, and `score` methods and behaves just like any other model in `scikit-learn`. The most common use case of the `Pipeline` class is in chaining preprocessing steps (like scaling of the data) together with a supervised model like a classifier.

Building Pipelines

Let's look at how we can use the `Pipeline` class to express the workflow for training an SVM after scaling the data with `MinMaxScaler` (for now without the grid search). First, we build a pipeline object by providing it with a list of steps. Each step is a tuple containing a name (any string of your choosing¹) and an instance of an estimator:

In[5]:

```
from sklearn.pipeline import Pipeline
pipe = Pipeline([('scaler', MinMaxScaler()), ('svm', SVC())])
```

Here, we created two steps: the first, called "scaler", is an instance of `MinMaxScaler`, and the second, called "svm", is an instance of `SVC`. Now, we can fit the pipeline, like any other `scikit-learn` estimator:

In[6]:

```
pipe.fit(X_train, y_train)
```

Here, `pipe.fit` first calls `fit` on the first step (the scaler), then transforms the training data using the scaler, and finally fits the SVM with the scaled data. To evaluate on the test data, we simply call `pipe.score`:

In[7]:

```
print("Test score: {:.2f}".format(pipe.score(X_test, y_test)))
```

Out[7]:

```
Test score: 0.95
```

Calling the `score` method on the pipeline first transforms the test data using the scaler, and then calls the `score` method on the SVM using the scaled test data. As you can see, the result is identical to the one we got from the code at the beginning of the chapter, when doing the transformations by hand. Using the pipeline, we reduced the code needed for our “preprocessing + classification” process. The main benefit of using the pipeline, however, is that we can now use this single estimator in `cross_val_score` or `GridSearchCV`.

¹ With one exception: the name can't contain a double underscore, `__`.

Using Pipelines in Grid Searches

Using a pipeline in a grid search works the same way as using any other estimator. We define a parameter grid to search over, and construct a `GridSearchCV` from the pipeline and the parameter grid. When specifying the parameter grid, there is a slight change, though. We need to specify for each parameter which step of the pipeline it belongs to. Both parameters that we want to adjust, `C` and `gamma`, are parameters of `SVC`, the second step. We gave this step the name "`svm`". The syntax to define a parameter grid for a pipeline is to specify for each parameter the step name, followed by `__` (a double underscore), followed by the parameter name. To search over the `C` parameter of `SVC` we therefore have to use "`svm__C`" as the key in the parameter grid dictionary, and similarly for `gamma`:

In[8]:

```
param_grid = {'svm__C': [0.001, 0.01, 0.1, 1, 10, 100],  
             'svm__gamma': [0.001, 0.01, 0.1, 1, 10, 100]}
```

With this parameter grid we can use `GridSearchCV` as usual:

In[9]:

```
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5)  
grid.fit(X_train, y_train)  
print("Best cross-validation accuracy: {:.2f}".format(grid.best_score_))  
print("Test set score: {:.2f}".format(grid.score(X_test, y_test)))  
print("Best parameters: {}".format(grid.best_params_))
```

Out[9]:

```
Best cross-validation accuracy: 0.98  
Test set score: 0.97  
Best parameters: {'svm__C': 1, 'svm__gamma': 1}
```

In contrast to the grid search we did before, now for each split in the cross-validation, the `MinMaxScaler` is refit with only the training splits and no information is leaked from the test split into the parameter search. Compare this (Figure 6-2) with Figure 6-1 earlier in this chapter:

In[10]:

```
mglearn.plots.plot_proper_processing()
```



Figure 6-2. Data usage when preprocessing inside the cross-validation loop with a pipeline

The impact of leaking information in the cross-validation varies depending on the nature of the preprocessing step. Estimating the scale of the data using the test fold usually doesn't have a terrible impact, while using the test fold in feature extraction and feature selection can lead to substantial differences in outcomes.

Illustrating Information Leakage

A great example of leaking information in cross-validation is given in Hastie, Tibshirani, and Friedman's book *The Elements of Statistical Learning*, and we reproduce an adapted version here. Let's consider a synthetic regression task with 100 samples and 1,000 features that are sampled independently from a Gaussian distribution. We also sample the response from a Gaussian distribution:

In[11]:

```
rnd = np.random.RandomState(seed=0)
X = rnd.normal(size=(100, 10000))
y = rnd.normal(size=(100,))
```

Given the way we created the dataset, there is no relation between the data, X , and the target, y (they are independent), so it should not be possible to learn anything from this dataset. We will now do the following. First, select the most informative of the 10 features using `SelectPercentile` feature selection, and then we evaluate a `Ridge` regressor using cross-validation:

In[12]:

```
from sklearn.feature_selection import SelectPercentile, f_regression

select = SelectPercentile(score_func=f_regression, percentile=5).fit(X, y)
X_selected = select.transform(X)
print("X_selected.shape: {}".format(X_selected.shape))
```

Out[12]:

```
X_selected.shape: (100, 500)
```

In[13]:

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import Ridge
print("Cross-validation accuracy (cv only on ridge): {:.2f}".format(
    np.mean(cross_val_score(Ridge(), X_selected, y, cv=5))))
```

Out[13]:

```
Cross-validation accuracy (cv only on ridge): 0.91
```

The mean R^2 computed by cross-validation is 0.91, indicating a very good model. This clearly cannot be right, as our data is entirely random. What happened here is that our feature selection picked out some features among the 10,000 random features that are (by chance) very well correlated with the target. Because we fit the feature selection *outside* of the cross-validation, it could find features that are correlated both on the training and the test folds. The information we leaked from the test folds was very informative, leading to highly unrealistic results. Let's compare this to a proper cross-validation using a pipeline:

In[14]:

```
pipe = Pipeline([('select', SelectPercentile(score_func=f_regression,
                                              percentile=5)),
                 ('ridge', Ridge())])
print("Cross-validation accuracy (pipeline): {:.2f}".format(
    np.mean(cross_val_score(pipe, X, y, cv=5))))
```

Out[14]:

```
Cross-validation accuracy (pipeline): -0.25
```

This time, we get a *negative* R^2 score, indicating a very poor model. Using the pipeline, the feature selection is now *inside* the cross-validation loop. This means features can only be selected using the training folds of the data, not the test fold. The feature selection finds features that are correlated with the target on the training set, but because the data is entirely random, these features are not correlated with the target on the test set. In this example, rectifying the data leakage issue in the feature selection makes the difference between concluding that a model works very well and concluding that a model works not at all.

The General Pipeline Interface

The `Pipeline` class is not restricted to preprocessing and classification, but can in fact join any number of estimators together. For example, you could build a pipeline containing feature extraction, feature selection, scaling, and classification, for a total of four steps. Similarly, the last step could be regression or clustering instead of classification.

The only requirement for estimators in a pipeline is that all but the last step need to have a `transform` method, so they can produce a new representation of the data that can be used in the next step.

Internally, during the call to `Pipeline.fit`, the pipeline calls `fit` and then `transform` on each step in turn,² with the input given by the output of the `transform` method of the previous step. For the last step in the pipeline, just `fit` is called.

Brushing over some finer details, this is implemented as follows. Remember that `pipeline.steps` is a list of tuples, so `pipeline.steps[0][1]` is the first estimator, `pipeline.steps[1][1]` is the second estimator, and so on:

In[15]:

```
def fit(self, X, y):
    X_transformed = X
    for name, estimator in self.steps[:-1]:
        # iterate over all but the final step
        # fit and transform the data
        X_transformed = estimator.fit_transform(X_transformed, y)
    # fit the last step
    self.steps[-1][1].fit(X_transformed, y)
    return self
```

When predicting using `Pipeline`, we similarly `transform` the data using all but the last step, and then call `predict` on the last step:

In[16]:

```
def predict(self, X):
    X_transformed = X
    for step in self.steps[:-1]:
        # iterate over all but the final step
        # transform the data
        X_transformed = step[1].transform(X_transformed)
    # fit the last step
    return self.steps[-1][1].predict(X_transformed)
```

² Or just `fit_transform`.

The process is illustrated in [Figure 6-3](#) for two transformers, T1 and T2, and a classifier (called Classifier).

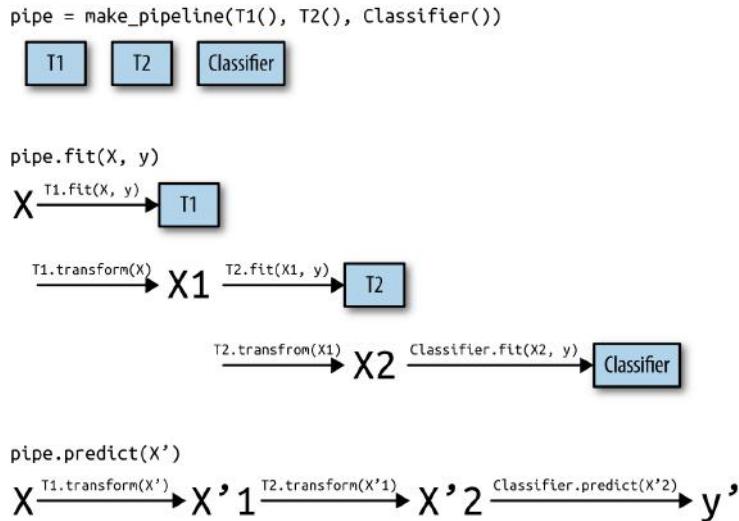


Figure 6-3. Overview of the pipeline training and prediction process

The pipeline is actually even more general than this. There is no requirement for the last step in a pipeline to have a `predict` function, and we could create a pipeline just containing, for example, a scaler and PCA. Then, because the last step (PCA) has a `transform` method, we could call `transform` on the pipeline to get the output of `PCA.transform` applied to the data that was processed by the previous step. The last step of a pipeline is only required to have a `fit` method.

Convenient Pipeline Creation with `make_pipeline`

Creating a pipeline using the syntax described earlier is sometimes a bit cumbersome, and we often don't need user-specified names for each step. There is a convenience function, `make_pipeline`, that will create a pipeline for us and automatically name each step based on its class. The syntax for `make_pipeline` is as follows:

In[17]:

```
from sklearn.pipeline import make_pipeline
# standard syntax
pipe_long = Pipeline([('scaler', MinMaxScaler()), ('svm', SVC(C=100))])
# abbreviated syntax
pipe_short = make_pipeline(MinMaxScaler(), SVC(C=100))
```

The pipeline objects `pipe_long` and `pipe_short` do exactly the same thing, but `pipe_short` has steps that were automatically named. We can see the names of the steps by looking at the `steps` attribute:

In[18]:

```
print("Pipeline steps:\n{}".format(pipe_short.steps))
```

Out[18]:

```
Pipeline steps:  
[('minmaxscaler', MinMaxScaler(copy=True, feature_range=(0, 1))),  
 ('svc', SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,  
             decision_function_shape=None, degree=3, gamma='auto',  
             kernel='rbf', max_iter=-1, probability=False,  
             random_state=None, shrinking=True, tol=0.001,  
             verbose=False)])
```

The steps are named `minmaxscaler` and `svc`. In general, the step names are just lowercase versions of the class names. If multiple steps have the same class, a number is appended:

In[19]:

```
from sklearn.preprocessing import StandardScaler  
from sklearn.decomposition import PCA  
  
pipe = make_pipeline(StandardScaler(), PCA(n_components=2), StandardScaler())  
print("Pipeline steps:\n{}".format(pipe.steps))
```

Out[19]:

```
Pipeline steps:  
[('standardscaler-1', StandardScaler(copy=True, with_mean=True, with_std=True)),  
 ('pca', PCA(copy=True, iterated_power=4, n_components=2, random_state=None,  
             svd_solver='auto', tol=0.0, whiten=False)),  
 ('standardscaler-2', StandardScaler(copy=True, with_mean=True, with_std=True))]
```

As you can see, the first `StandardScaler` step was named `standardscaler-1` and the second `standardscaler-2`. However, in such settings it might be better to use the `Pipeline` construction with explicit names, to give more semantic names to each step.

Accessing Step Attributes

Often you will want to inspect attributes of one of the steps of the pipeline—say, the coefficients of a linear model or the components extracted by PCA. The easiest way to access the steps in a pipeline is via the `named_steps` attribute, which is a dictionary from the step names to the estimators:

In[20]:

```
# fit the pipeline defined before to the cancer dataset
pipe.fit(cancer.data)
# extract the first two principal components from the "pca" step
components = pipe.named_steps["pca"].components_
print("components.shape: {}".format(components.shape))
```

Out[20]:

```
components.shape: (2, 30)
```

Accessing Attributes in a Grid-Searched Pipeline

As we discussed earlier in this chapter, one of the main reasons to use pipelines is for doing grid searches. A common task is to access some of the steps of a pipeline inside a grid search. Let's grid search a `LogisticRegression` classifier on the `cancer` dataset, using `Pipeline` and `StandardScaler` to scale the data before passing it to the `LogisticRegression` classifier. First we create a pipeline using the `make_pipeline` function:

In[21]:

```
from sklearn.linear_model import LogisticRegression

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

Next, we create a parameter grid. As explained in [Chapter 2](#), the regularization parameter to tune for `LogisticRegression` is the parameter `C`. We use a logarithmic grid for this parameter, searching between 0.01 and 100. Because we used the `make_pipeline` function, the name of the `LogisticRegression` step in the pipeline is the lowercased class name, `logisticregression`. To tune the parameter `C`, we therefore have to specify a parameter grid for `logisticregression__C`:

In[22]:

```
param_grid = {'logisticregression__C': [0.01, 0.1, 1, 10, 100]}
```

As usual, we split the `cancer` dataset into training and test sets, and fit a grid search:

In[23]:

```
X_train, X_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=4)
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
```

So how do we access the coefficients of the best `LogisticRegression` model that was found by `GridSearchCV`? From [Chapter 5](#) we know that the best model found by `GridSearchCV`, trained on all the training data, is stored in `grid.best_estimator_`:

In[24]:

```
print("Best estimator:\n{}".format(grid.best_estimator_))
```

Out[24]:

```
Best estimator:  
Pipeline(steps=[  
    ('standardscaler', StandardScaler(copy=True, with_mean=True, with_std=True)),  
    ('logisticregression', LogisticRegression(C=0.1, class_weight=None,  
        dual=False, fit_intercept=True, intercept_scaling=1, max_iter=100,  
        multi_class='ovr', n_jobs=1, penalty='l2', random_state=None,  
        solver='liblinear', tol=0.0001, verbose=0, warm_start=False))])
```

This `best_estimator_` in our case is a pipeline with two steps, `standardscaler` and `logisticregression`. To access the `logisticregression` step, we can use the `named_steps` attribute of the pipeline, as explained earlier:

In[25]:

```
print("Logistic regression step:\n{}".format(  
    grid.best_estimator_.named_steps["logisticregression"]))
```

Out[25]:

```
Logistic regression step:  
LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,  
    intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,  
    penalty='l2', random_state=None, solver='liblinear', tol=0.0001,  
    verbose=0, warm_start=False)
```

Now that we have the trained `LogisticRegression` instance, we can access the coefficients (weights) associated with each input feature:

In[26]:

```
print("Logistic regression coefficients:\n{}".format(  
    grid.best_estimator_.named_steps["logisticregression"].coef_))
```

Out[26]:

```
Logistic regression coefficients:  
[[ -0.389 -0.375 -0.376 -0.396 -0.115  0.017 -0.355 -0.39 -0.058  0.209  
  -0.495 -0.004 -0.371 -0.383 -0.045  0.198  0.004 -0.049  0.21  0.224  
  -0.547 -0.525 -0.499 -0.515 -0.393 -0.123 -0.388 -0.417 -0.325 -0.139]]
```

This might be a somewhat lengthy expression, but often it comes in handy in understanding your models.

Grid-Searching Preprocessing Steps and Model Parameters

Using pipelines, we can encapsulate all the processing steps in our machine learning workflow in a single scikit-learn estimator. Another benefit of doing this is that we can now *adjust the parameters of the preprocessing* using the outcome of a supervised task like regression or classification. In previous chapters, we used polynomial features on the boston dataset before applying the ridge regressor. Let's model that using a pipeline instead. The pipeline contains three steps—scaling the data, computing polynomial features, and ridge regression:

In[27]:

```
from sklearn.datasets import load_boston
boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split(boston.data, boston.target,
                                                    random_state=0)

from sklearn.preprocessing import PolynomialFeatures
pipe = make_pipeline(
    StandardScaler(),
    PolynomialFeatures(),
    Ridge())
```

How do we know which degrees of polynomials to choose, or whether to choose any polynomials or interactions at all? Ideally we want to select the `degree` parameter based on the outcome of the classification. Using our pipeline, we can search over the `degree` parameter together with the parameter `alpha` of `Ridge`. To do this, we define a `param_grid` that contains both, appropriately prefixed by the step names:

In[28]:

```
param_grid = {'polynomialfeatures_degree': [1, 2, 3],
              'ridge_alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
```

Now we can run our grid search again:

In[29]:

```
grid = GridSearchCV(pipe, param_grid=param_grid, cv=5, n_jobs=-1)
grid.fit(X_train, y_train)
```

We can visualize the outcome of the cross-validation using a heat map (Figure 6-4), as we did in [Chapter 5](#):

In[30]:

```
plt.matshow(grid.cv_results_['mean_test_score'].reshape(3, -1),
            vmin=0, cmap="viridis")
plt.xlabel("ridge_alpha")
plt.ylabel("polynomialfeatures_degree")
```

```

plt.xticks(range(len(param_grid['ridge_alpha'])), param_grid['ridge_alpha'])
plt.yticks(range(len(param_grid['polynomialfeatures_degree'])),
           param_grid['polynomialfeatures_degree'])

plt.colorbar()

```

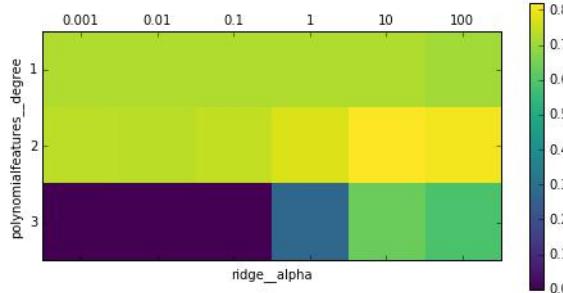


Figure 6-4. Heat map of mean cross-validation score as a function of the degree of the polynomial features and alpha parameter of Ridge

Looking at the results produced by the cross-validation, we can see that using polynomials of degree two helps, but that degree-three polynomials are much worse than either degree one or two. This is reflected in the best parameters that were found:

In[31]:

```
print("Best parameters: {}".format(grid.best_params_))
```

Out[31]:

```
Best parameters: {'polynomialfeatures_degree': 2, 'ridge_alpha': 10}
```

Which lead to the following score:

In[32]:

```
print("Test-set score: {:.2f}".format(grid.score(X_test, y_test)))
```

Out[32]:

```
Test-set score: 0.77
```

Let's run a grid search without polynomial features for comparison:

In[33]:

```

param_grid = {'ridge_alpha': [0.001, 0.01, 0.1, 1, 10, 100]}
pipe = make_pipeline(StandardScaler(), Ridge())
grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(X_train, y_train)
print("Score without poly features: {:.2f}".format(grid.score(X_test, y_test)))

```

Out[33]:

```
Score without poly features: 0.63
```

As we would expect looking at the grid search results visualized in [Figure 6-4](#), using no polynomial features leads to decidedly worse results.

Searching over preprocessing parameters together with model parameters is a very powerful strategy. However, keep in mind that `GridSearchCV` tries *all possible combinations* of the specified parameters. Therefore, adding more parameters to your grid exponentially increases the number of models that need to be built.

Grid-Searching Which Model To Use

You can even go further in combining `GridSearchCV` and `Pipeline`: it is also possible to search over the actual steps being performed in the pipeline (say whether to use `StandardScaler` or `MinMaxScaler`). This leads to an even bigger search space and should be considered carefully. Trying all possible solutions is usually not a viable machine learning strategy. However, here is an example comparing a `RandomForestClassifier` and an `SVC` on the `iris` dataset. We know that the `SVC` might need the data to be scaled, so we also search over whether to use `StandardScaler` or no preprocessing. For the `RandomForestClassifier`, we know that no preprocessing is necessary. We start by defining the pipeline. Here, we explicitly name the steps. We want two steps, one for the preprocessing and then a classifier. We can instantiate this using `SVC` and `StandardScaler`:

In[34]:

```
pipe = Pipeline([('preprocessing', StandardScaler()), ('classifier', SVC())])
```

Now we can define the `parameter_grid` to search over. We want the `classifier` to be either `RandomForestClassifier` or `SVC`. Because they have different parameters to tune, and need different preprocessing, we can make use of the list of search grids we discussed in “[Search over spaces that are not grids](#)” on page 271. To assign an estimator to a step, we use the name of the step as the parameter name. When we wanted to skip a step in the pipeline (for example, because we don’t need preprocessing for the `RandomForest`), we can set that step to `None`:

In[35]:

```
from sklearn.ensemble import RandomForestClassifier

param_grid = [
    {'classifier': [SVC()], 'preprocessing': [StandardScaler(), None],
     'classifier__gamma': [0.001, 0.01, 0.1, 1, 10, 100],
     'classifier__C': [0.001, 0.01, 0.1, 1, 10, 100]},
    {'classifier': [RandomForestClassifier(n_estimators=100)],
     'preprocessing': [None], 'classifier__max_features': [1, 2, 3]}]
```

Now we can instantiate and run the grid search as usual, here on the `cancer` dataset:

In[36]:

```
X_train, X_test, y_train, y_test = train_test_split(  
    cancer.data, cancer.target, random_state=0)  
  
grid = GridSearchCV(pipe, param_grid, cv=5)  
grid.fit(X_train, y_train)  
  
print("Best params: \n{}\n".format(grid.best_params_))  
print("Best cross-validation score: {:.2f}\n".format(grid.best_score_))  
print("Test-set score: {:.2f}\n".format(grid.score(X_test, y_test)))
```

Out[36]:

```
Best params:  
{'classifier':  
 SVC(C=10, cache_size=200, class_weight=None, coef0=0.0,  
     decision_function_shape=None, degree=3, gamma=0.01, kernel='rbf',  
     max_iter=-1, probability=False, random_state=None, shrinking=True,  
     tol=0.001, verbose=False),  
 'preprocessing':  
 StandardScaler(copy=True, with_mean=True, with_std=True),  
 'classifier__C': 10, 'classifier__gamma': 0.01}  
  
Best cross-validation score: 0.99  
Test-set score: 0.98
```

The outcome of the grid search is that SVC with `StandardScaler` preprocessing, `C=10`, and `gamma=0.01` gave the best result.

Summary and Outlook

In this chapter we introduced the `Pipeline` class, a general-purpose tool to chain together multiple processing steps in a machine learning workflow. Real-world applications of machine learning rarely involve an isolated use of a model, and instead are a sequence of processing steps. Using pipelines allows us to encapsulate multiple steps into a single Python object that adheres to the familiar `scikit-learn` interface of `fit`, `predict`, and `transform`. In particular when doing model evaluation using cross-validation and parameter selection using grid search, using the `Pipeline` class to capture all the processing steps is essential for proper evaluation. The `Pipeline` class also allows writing more succinct code, and reduces the likelihood of mistakes that can happen when building processing chains without the `pipeline` class (like forgetting to apply all transformers on the test set, or not applying them in the right order). Choosing the right combination of feature extraction, preprocessing, and models is somewhat of an art, and often requires some trial and error. However, using pipelines, this “trying out” of many different processing steps is quite simple. When

experimenting, be careful not to overcomplicate your processes, and make sure to evaluate whether every component you are including in your model is necessary.

With this chapter, we have completed our survey of general-purpose tools and algorithms provided by `scikit-learn`. You now possess all the required skills and know the necessary mechanisms to apply machine learning in practice. In the next chapter, we will dive in more detail into one particular type of data that is commonly seen in practice, and that requires some special expertise to handle correctly: text data.

Working with Text Data

In [Chapter 4](#), we talked about two kinds of features that can represent properties of the data: continuous features that describe a quantity, and categorical features that are items from a fixed list. There is a third kind of feature that can be found in many applications, which is text. For example, if we want to classify an email message as either a legitimate email or spam, the content of the email will certainly contain important information for this classification task. Or maybe we want to learn about the opinion of a politician on the topic of immigration. Here, that individual's speeches or tweets might provide useful information. In customer service, we often want to find out if a message is a complaint or an inquiry. We can use the subject line and content of a message to automatically determine the customer's intent, which allows us to send the message to the appropriate department, or even send a fully automatic reply.

Text data is usually represented as strings, made up of characters. In any of the examples just given, the length of the text data will vary. This feature is clearly very different from the numeric features that we've discussed so far, and we will need to process the data before we can apply our machine learning algorithms to it.

Types of Data Represented as Strings

Before we dive into the processing steps that go into representing text data for machine learning, we want to briefly discuss different kinds of text data that you might encounter. Text is usually just a string in your dataset, but not all string features should be treated as text. A string feature can sometimes represent categorical variables, as we discussed in [Chapter 5](#). There is no way to know how to treat a string feature before looking at the data.

There are four kinds of string data you might see:

- Categorical data
- Free strings that can be semantically mapped to categories
- Structured string data
- Text data

Categorical data is data that comes from a fixed list. Say you collect data via a survey where you ask people their favorite color, with a drop-down menu that allows them to select from “red,” “green,” “blue,” “yellow,” “black,” “white,” “purple,” and “pink.” This will result in a dataset with exactly eight different possible values, which clearly encode a categorical variable. You can check whether this is the case for your data by eyeballing it (if you see very many different strings it is unlikely that this is a categorical variable) and confirm it by computing the unique values over the dataset, and possibly a histogram over how often each appears. You also might want to check whether each variable actually corresponds to a category that makes sense for your application. Maybe halfway through the existence of your survey, someone found that “black” was misspelled as “blak” and subsequently fixed the survey. As a result, your dataset contains both “blak” and “black,” which correspond to the same semantic meaning and should be consolidated.

Now imagine instead of providing a drop-down menu, you provide a text field for the users to provide their own favorite colors. Many people might respond with a color name like “black” or “blue.” Others might make typographical errors, use different spellings like “gray” and “grey,” or use more evocative and specific names like “midnight blue.” You will also have some very strange entries. Some good examples come from [the xkcd Color Survey](#), where people had to name colors and came up with names like “velociraptor cloaka” and “my dentist’s office orange. I still remember his dandruff slowly wafting into my gaping yaw,” which are hard to map to colors automatically (or at all). The responses you can obtain from a text field belong to the second category in the list, *free strings that can be semantically mapped to categories*. It will probably be best to encode this data as a categorical variable, where you can select the categories either by using the most common entries, or by defining categories that will capture responses in a way that makes sense for your application. You might then have some categories for standard colors, maybe a category “multicolored” for people that gave answers like “green and red stripes,” and an “other” category for things that cannot be encoded otherwise. This kind of preprocessing of strings can take a lot of manual effort and is not easily automated. If you are in a position where you can influence data collection, we highly recommend avoiding manually entered values for concepts that are better captured using categorical variables.

Often, manually entered values do not correspond to fixed categories, but still have some underlying *structure*, like addresses, names of places or people, dates, telephone

numbers, or other identifiers. These kinds of strings are often very hard to parse, and their treatment is highly dependent on context and domain. A systematic treatment of these cases is beyond the scope of this book.

The final category of string data is freeform *text data* that consists of phrases or sentences. Examples include tweets, chat logs, and hotel reviews, as well as the collected works of Shakespeare, the content of Wikipedia, or the Project Gutenberg collection of 50,000 ebooks. All of these collections contain information mostly as sentences composed of words.¹ For simplicity's sake, let's assume all our documents are in one language, English.² In the context of text analysis, the dataset is often called the *corpus*, and each data point, represented as a single text, is called a *document*. These terms come from the *information retrieval* (IR) and *natural language processing* (NLP) community, which both deal mostly in text data.

Example Application: Sentiment Analysis of Movie Reviews

As a running example in this chapter, we will use a dataset of movie reviews from the IMDb (Internet Movie Database) website collected by Stanford researcher Andrew Maas.³ This dataset contains the text of the reviews, together with a label that indicates whether a review is “positive” or “negative.” The IMDb website itself contains ratings from 1 to 10. To simplify the modeling, this annotation is summarized as a two-class classification dataset where reviews with a score of 6 or higher are labeled as positive, and the rest as negative. We will leave the question of whether this is a good representation of the data open, and simply use the data as provided by Andrew Maas.

After unpacking the data, the dataset is provided as text files in two separate folders, one for the training data and one for the test data. Each of these in turn has two sub-folders, one called *pos* and one called *neg*:

¹ Arguably, the content of websites linked to in tweets contains more information than the text of the tweets themselves.

² Most of what we will talk about in the rest of the chapter also applies to other languages that use the Roman alphabet, and partially to other languages with word boundary delimiters. Chinese, for example, does not delimit word boundaries, and has other challenges that make applying the techniques in this chapter difficult.

³ The dataset is available at <http://ai.stanford.edu/~amaas/data/sentiment/>.

In[2]:

```
!tree -L 2 data/aclImdb
```

Out[2]:

```
data/aclImdb
└── test
    ├── neg
    └── pos
└── train
    ├── neg
    └── pos
```

6 directories, 0 files

The *pos* folder contains all the positive reviews, each as a separate text file, and similarly for the *neg* folder. There is a helper function in `scikit-learn` to load files stored in such a folder structure, where each subfolder corresponds to a label, called `load_files`. We apply the `load_files` function first to the training data:

In[3]:

```
from sklearn.datasets import load_files

reviews_train = load_files("data/aclImdb/train/")
# load_files returns a bunch, containing training texts and training labels
text_train, y_train = reviews_train.data, reviews_train.target
print("type of text_train: {}".format(type(text_train)))
print("length of text_train: {}".format(len(text_train)))
print("text_train[1]:\n{}".format(text_train[1]))
```

Out[3]:

```
type of text_train: <class 'list'>
length of text_train: 25000
text_train[1]:
b'Words can\'t describe how bad this movie is. I can\'t explain it by writing
only. You have too see it for yourself to get at grip of how horrible a movie
really can be. Not that I recommend you to do that. There are so many
clich\u00e9s, mistakes (and all other negative things you can imagine) here
that will just make you cry. To start with the technical first, there are a
LOT of mistakes regarding the airplane. I won\'t list them here, but just
mention the coloring of the plane. They didn\'t even manage to show an
airliner in the colors of a fictional airline, but instead used a 747
painted in the original Boeing livery. Very bad. The plot is stupid and has
been done many times before, only much, much better. There are so many
ridiculous moments here that i lost count of it really early. Also, I was on
the bad guys\' side all the time in the movie, because the good guys were so
stupid. "Executive Decision" should without a doubt be you\'re choice over
this one, even the "Turbulence"-movies are better. In fact, every other
movie in the world is better than this one.'
```

You can see that `text_train` is a list of length 25,000, where each entry is a string containing a review. We printed the review with index 1. You can also see that the review contains some HTML line breaks (`
`). While these are unlikely to have a

large impact on our machine learning models, it is better to clean the data and remove this formatting before we proceed:

In[4]:

```
text_train = [doc.replace(b"<br />", b" ") for doc in text_train]
```

The type of the entries of `text_train` will depend on your Python version. In Python 3, they will be of type `bytes` which represents a binary encoding of the string data. In Python 2, `text_train` contains strings. We won't go into the details of the different string types in Python here, but we recommend that you read [the Python 2](#) and/or [Python 3 documentation](#) regarding strings and Unicode.

The dataset was collected such that the positive class and the negative class balanced, so that there are as many positive as negative strings:

In[5]:

```
print("Samples per class (training): {}".format(np.bincount(y_train)))
```

Out[5]:

```
Samples per class (training): [12500 12500]
```

We load the test dataset in the same manner:

In[6]:

```
reviews_test = load_files("data/aclImdb/test/")
text_test, y_test = reviews_test.data, reviews_test.target
print("Number of documents in test data: {}".format(len(text_test)))
print("Samples per class (test): {}".format(np.bincount(y_test)))
text_test = [doc.replace(b"<br />", b" ") for doc in text_test]
```

Out[6]:

```
Number of documents in test data: 25000
Samples per class (test): [12500 12500]
```

The task we want to solve is as follows: given a review, we want to assign the label “positive” or “negative” based on the text content of the review. This is a standard binary classification task. However, the text data is not in a format that a machine learning model can handle. We need to convert the string representation of the text into a numeric representation that we can apply our machine learning algorithms to.

Representing Text Data as a Bag of Words

One of the most simple but effective and commonly used ways to represent text for machine learning is using the *bag-of-words* representation. When using this representation, we discard most of the structure of the input text, like chapters, paragraphs, sentences, and formatting, and only count *how often each word appears in each text* in

the corpus. Discarding the structure and counting only word occurrences leads to the mental image of representing text as a “bag.”

Computing the bag-of-words representation for a corpus of documents consists of the following three steps:

1. *Tokenization.* Split each document into the words that appear in it (called *tokens*), for example by splitting them on whitespace and punctuation.
2. *Vocabulary building.* Collect a vocabulary of all words that appear in any of the documents, and number them (say, in alphabetical order).
3. *Encoding.* For each document, count how often each of the words in the vocabulary appear in this document.

There are some subtleties involved in step 1 and step 2, which we will discuss in more detail later in this chapter. For now, let’s look at how we can apply the bag-of-words processing using `scikit-learn`. Figure 7-1 illustrates the process on the string "This is how you get ants.". The output is one vector of word counts for each document. For each word in the vocabulary, we have a count of how often it appears in each document. That means our numeric representation has one feature for each unique word in the whole dataset. Note how the order of the words in the original string is completely irrelevant to the bag-of-words feature representation.

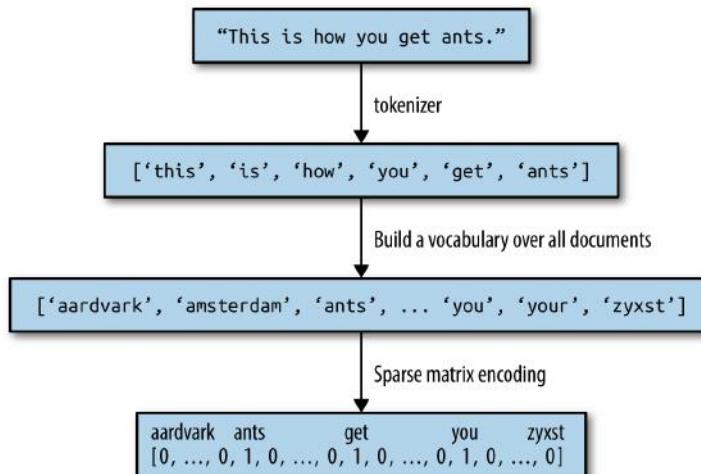


Figure 7-1. Bag-of-words processing

Applying Bag-of-Words to a Toy Dataset

The bag-of-words representation is implemented in `CountVectorizer`, which is a transformer. Let's first apply it to a toy dataset, consisting of two samples, to see it working:

In[7]:

```
bards_words = ["The fool doth think he is wise,",  
               "but the wise man knows himself to be a fool"]
```

We import and instantiate the `CountVectorizer` and `fit` it to our toy data as follows:

In[8]:

```
from sklearn.feature_extraction.text import CountVectorizer  
vect = CountVectorizer()  
vect.fit(bards_words)
```

Fitting the `CountVectorizer` consists of the tokenization of the training data and building of the vocabulary, which we can access as the `vocabulary_` attribute:

In[9]:

```
print("Vocabulary size: {}".format(len(vect.vocabulary_)))  
print("Vocabulary content:\n {}".format(vect.vocabulary_))
```

Out[9]:

```
Vocabulary size: 13  
Vocabulary content:  
{'the': 9, 'himself': 5, 'wise': 12, 'he': 4, 'doth': 2, 'to': 11, 'knows': 7,  
'man': 8, 'fool': 3, 'is': 6, 'be': 0, 'think': 10, 'but': 1}
```

The vocabulary consists of 13 words, from "be" to "wise".

To create the bag-of-words representation for the training data, we call the `transform` method:

In[10]:

```
bag_of_words = vect.transform(bards_words)  
print("bag_of_words: {}".format(repr(bag_of_words)))
```

Out[10]:

```
bag_of_words: <2x13 sparse matrix of type '<class 'numpy.int64'>'  
with 16 stored elements in Compressed Sparse Row format>
```

The bag-of-words representation is stored in a SciPy sparse matrix that only stores the entries that are nonzero (see [Chapter 1](#)). The matrix is of shape 2×13 , with one row for each of the two data points and one feature for each of the words in the vocabulary. A sparse matrix is used as most documents only contain a small subset of the words in the vocabulary, meaning most entries in the feature array are 0. Think

about how many different words might appear in a movie review compared to all the words in the English language (which is what the vocabulary models). Storing all those zeros would be prohibitive, and a waste of memory. To look at the actual content of the sparse matrix, we can convert it to a “dense” NumPy array (that also stores all the 0 entries) using the `toarray` method:⁴

In[11]:

```
print("Dense representation of bag_of_words:\n{}".format(
    bag_of_words.toarray()))
```

Out[11]:

```
Dense representation of bag_of_words:
[[0 0 1 1 0 1 0 0 1 1 0 1]
 [1 1 0 1 0 1 0 1 1 1 0 1]]
```

We can see that the word counts for each word are either 0 or 1; neither of the two strings in `bag_of_words` contains a word twice. Let’s take a look at how to read these feature vectors. The first string (“The fool doth think he is wise,”) is represented as the first row in, and it contains the first word in the vocabulary, “be”, zero times. It also contains the second word in the vocabulary, “but”, zero times. It contains the third word, “doth”, once, and so on. Looking at both rows, we can see that the fourth word, “fool”, the tenth word, “the”, and the thirteenth word, “wise”, appear in both strings.

Bag-of-Words for Movie Reviews

Now that we’ve gone through the bag-of-words process in detail, let’s apply it to our task of sentiment analysis for movie reviews. Earlier, we loaded our training and test data from the IMDb reviews into lists of strings (`text_train` and `text_test`), which we will now process:

In[12]:

```
vect = CountVectorizer().fit(text_train)
X_train = vect.transform(text_train)
print("X_train:\n{}".format(repr(X_train)))
```

Out[12]:

```
X_train:
<25000x74849 sparse matrix of type '<class 'numpy.int64'>'>
with 3431196 stored elements in Compressed Sparse Row format>
```

⁴ This is possible because we are using a small toy dataset that contains only 13 words. For any real dataset, this would result in a `MemoryError`.

The shape of `X_train`, the bag-of-words representation of the training data, is $25,000 \times 74,849$, indicating that the vocabulary contains 74,849 entries. Again, the data is stored as a SciPy sparse matrix. Let's look at the vocabulary in a bit more detail. Another way to access the vocabulary is using the `get_feature_name` method of the vectorizer, which returns a convenient list where each entry corresponds to one feature:

In[13]:

```
feature_names = vect.get_feature_names()
print("Number of features: {}".format(len(feature_names)))
print("First 20 features:\n{}".format(feature_names[:20]))
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("Every 2000th feature:\n{}".format(feature_names[::-2000]))
```

Out[13]:

```
Number of features: 74849
First 20 features:
['00', '000', '000000000001', '00001', '00015', '000s', '001', '003830',
 '006', '007', '0079', '0080', '0083', '0093638', '00am', '00pm', '00s',
 '01', '01pm', '02']
Features 20010 to 20030:
['dratted', 'draub', 'draught', 'draughts', 'draughtswoman', 'draw', 'drawback',
 'drawbacks', 'drawer', 'drawers', 'drawing', 'drawings', 'drawl',
 'drawled', 'drawling', 'drawn', 'draws', 'draza', 'dre', 'drea']
Every 2000th feature:
['00', 'aesir', 'aquarian', 'barking', 'blustering', 'bête', 'chicanery',
 'condensing', 'cunning', 'detox', 'draper', 'enshrined', 'favorit', 'freezer',
 'goldman', 'hasan', 'huitieme', 'intelligible', 'kantrowitz', 'lawful',
 'maars', 'megalunged', 'mostey', 'norrland', 'padilla', 'pincher',
 'promisingly', 'receptionist', 'rivals', 'schnaas', 'shunning', 'sparse',
 'subset', 'temptations', 'treatises', 'unproven', 'walkman', 'xylophonist']
```

As you can see, possibly a bit surprisingly, the first 10 entries in the vocabulary are all numbers. All these numbers appear somewhere in the reviews, and are therefore extracted as words. Most of these numbers don't have any immediate semantic meaning—apart from "007", which in the particular context of movies is likely to refer to the James Bond character.⁵ Weeding out the meaningful from the nonmeaningful “words” is sometimes tricky. Looking further along in the vocabulary, we find a collection of English words starting with “dra”. You might notice that for “draught”, “drawback”, and “drawer” both the singular and plural forms are contained in the vocabulary as distinct words. These words have very closely related semantic meanings, and counting them as different words, corresponding to different features, might not be ideal.

⁵ A quick analysis of the data confirms that this is indeed the case. Try confirming it yourself.

Before we try to improve our feature extraction, let's obtain a quantitative measure of performance by actually building a classifier. We have the training labels stored in `y_train` and the bag-of-words representation of the training data in `X_train`, so we can train a classifier on this data. For high-dimensional, sparse data like this, linear models like `LogisticRegression` often work best.

Let's start by evaluating `LogisticRegression` using cross-validation:⁶

In[14]:

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
scores = cross_val_score(LogisticRegression(), X_train, y_train, cv=5)
print("Mean cross-validation accuracy: {:.2f}".format(np.mean(scores)))
```

Out[14]:

```
Mean cross-validation accuracy: 0.88
```

We obtain a mean cross-validation score of 88%, which indicates reasonable performance for a balanced binary classification task. We know that `LogisticRegression` has a regularization parameter, `C`, which we can tune via cross-validation:

In[15]:

```
from sklearn.model_selection import GridSearchCV
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
print("Best parameters: ", grid.best_params_)
```

Out[15]:

```
Best cross-validation score: 0.89
Best parameters: {'C': 0.1}
```

We obtain a cross-validation score of 89% using `C=0.1`. We can now assess the generalization performance of this parameter setting on the test set:

In[16]:

```
X_test = vect.transform(text_test)
print("{:.2f}".format(grid.score(X_test, y_test)))
```

Out[16]:

```
0.88
```

⁶ The attentive reader might notice that we violate our lesson from [Chapter 6](#) on cross-validation with preprocessing here. Using the default settings of `CountVectorizer`, it actually does not collect any statistics, so our results are valid. Using `Pipeline` from the start would be a better choice for applications, but we defer it for ease of exposure.

Now, let's see if we can improve the extraction of words. The CountVectorizer extracts tokens using a regular expression. By default, the regular expression that is used is "`\b\w\w+\b`". If you are not familiar with regular expressions, this means it finds all sequences of characters that consist of at least two letters or numbers (`\w`) and that are separated by word boundaries (`\b`). It does not find single-letter words, and it splits up contractions like "doesn't" or "bit.ly", but it matches "h8ter" as a single word. The CountVectorizer then converts all words to lowercase characters, so that "soon", "Soon", and "sOon" all correspond to the same token (and therefore feature). This simple mechanism works quite well in practice, but as we saw earlier, we get many uninformative features (like the numbers). One way to cut back on these is to only use tokens that appear in at least two documents (or at least five documents, and so on). A token that appears only in a single document is unlikely to appear in the test set and is therefore not helpful. We can set the minimum number of documents a token needs to appear in with the `min_df` parameter:

In[17]:

```
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train with min_df: {}".format(repr(X_train)))
```

Out[17]:

```
X_train with min_df: <25000x27271 sparse matrix of type '<class 'numpy.int64'>'  
with 3354014 stored elements in Compressed Sparse Row format>
```

By requiring at least five appearances of each token, we can bring down the number of features to 27,271, as seen in the preceding output—only about a third of the original features. Let's look at some tokens again:

In[18]:

```
feature_names = vect.get_feature_names()

print("First 50 features:\n{}".format(feature_names[:50]))
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("Every 700th feature:\n{}".format(feature_names[::700]))
```

Out[18]:

```
First 50 features:  
['00', '000', '007', '00s', '01', '02', '03', '04', '05', '06', '07', '08',  
 '09', '10', '100', '1000', '100th', '101', '102', '103', '104', '105', '107',  
 '108', '10s', '10th', '11', '110', '112', '116', '117', '11th', '12', '120',  
 '12th', '13', '135', '13th', '14', '140', '14th', '15', '150', '15th', '16',  
 '160', '1600', '16mm', '16s', '16th']  
Features 20010 to 20030:  
['repentance', 'repercussions', 'repertoire', 'repetition', 'repetitions',  
 'repetitious', 'repetitive', 'rephrase', 'replace', 'replaced', 'replacement',  
 'replaces', 'replacing', 'replay', 'replayable', 'replayed', 'replaying',  
 'replays', 'replete', 'replica']
```

Every 700th feature:

```
['00', 'affections', 'appropriately', 'barbra', 'blurbs', 'butchered',
 'cheese', 'commitment', 'courts', 'deconstructed', 'disgraceful', 'dvds',
 'eschews', 'fell', 'freezer', 'goriest', 'hauser', 'hungary', 'insinuate',
 'juggle', 'leering', 'maelstrom', 'messiah', 'music', 'occasional', 'parking',
 'pleasantville', 'pronunciation', 'recipient', 'reviews', 'sas', 'shea',
 'sneers', 'steiger', 'swastika', 'thrusting', 'tvs', 'vampyre', 'westerns']
```

There are clearly many fewer numbers, and some of the more obscure words or misspellings seem to have vanished. Let's see how well our model performs by doing a grid search again:

In[19]:

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
```

Out[19]:

```
Best cross-validation score: 0.89
```

The best validation accuracy of the grid search is still 89%, unchanged from before. We didn't improve our model, but having fewer features to deal with speeds up processing and throwing away useless features might make the model more interpretable.



If the `transform` method of `CountVectorizer` is called on a document that contains words that were not contained in the training data, these words will be ignored as they are not part of the dictionary. This is not really an issue for classification, as it's not possible to learn anything about words that are not in the training data. For some applications, like spam detection, it might be helpful to add a feature that encodes how many so-called “out of vocabulary” words there are in a particular document, though. For this to work, you need to set `min_df`; otherwise, this feature will never be active during training.

Stopwords

Another way that we can get rid of uninformative words is by discarding words that are too frequent to be informative. There are two main approaches: using a language-specific list of stopwords, or discarding words that appear too frequently. `scikit-learn` has a built-in list of English stopwords in the `feature_extraction.text` module:

In[20]:

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
print("Number of stop words: {}".format(len(ENGLISH_STOP_WORDS)))
print("Every 10th stopword:\n{}".format(list(ENGLISH_STOP_WORDS)[::10]))
```

Out[20]:

```
Number of stop words: 318
Every 10th stopword:
['above', 'elsewhere', 'into', 'well', 'rather', 'fifteen', 'had', 'enough',
 'herein', 'should', 'third', 'although', 'more', 'this', 'none', 'seemed',
 'nobody', 'seems', 'he', 'also', 'fill', 'anyone', 'anything', 'me', 'the',
 'yet', 'go', 'seeming', 'front', 'beforehand', 'forty', 'i']
```

Clearly, removing the stopwords in the list can only decrease the number of features by the length of the list—here, 318—but it might lead to an improvement in performance. Let’s give it a try:

In[21]:

```
# Specifying stop_words="english" uses the built-in list.
# We could also augment it and pass our own.
vect = CountVectorizer(min_df=5, stop_words="english").fit(text_train)
X_train = vect.transform(text_train)
print("X_train with stop words:\n{}".format(repr(X_train)))
```

Out[21]:

```
X_train with stop words:
<25000x26966 sparse matrix of type '<class 'numpy.int64'>'>
    with 2149958 stored elements in Compressed Sparse Row format>
```

There are now 305 (27,271–26,966) fewer features in the dataset, which means that most, but not all, of the stopwords appeared. Let’s run the grid search again:

In[22]:

```
grid = GridSearchCV(LogisticRegression(), param_grid, cv=5)
grid.fit(X_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
```

Out[22]:

```
Best cross-validation score: 0.88
```

The grid search performance decreased slightly using the stopwords—not enough to worry about, but given that excluding 305 features out of over 27,000 is unlikely to change performance or interpretability a lot, it doesn’t seem worth using this list. Fixed lists are mostly helpful for small datasets, which might not contain enough information for the model to determine which words are stopwords from the data itself. As an exercise, you can try out the other approach, discarding frequently

appearing words, by setting the `max_df` option of `CountVectorizer` and see how it influences the number of features and the performance.

Rescaling the Data with tf–idf

Instead of dropping features that are deemed unimportant, another approach is to rescale features by how informative we expect them to be. One of the most common ways to do this is using the *term frequency-inverse document frequency* (tf–idf) method. The intuition of this method is to give high weight to any term that appears often in a particular document, but not in many documents in the corpus. If a word appears often in a particular document, but not in very many documents, it is likely to be very descriptive of the content of that document. `scikit-learn` implements the tf–idf method in two classes: `TfidfTransformer`, which takes in the sparse matrix output produced by `CountVectorizer` and transforms it, and `TfidfVectorizer`, which takes in the text data and does both the bag-of-words feature extraction and the tf–idf transformation. There are several variants of the tf–idf rescaling scheme, which you can [read about on Wikipedia](#). The tf–idf score for word w in document d as implemented in both the `TfidfTransformer` and `TfidfVectorizer` classes is given by:⁷

$$\text{tfidf}(w, d) = \text{tf} \log \left(\frac{N + 1}{N_w + 1} \right) + 1$$

where N is the number of documents in the training set, N_w is the number of documents in the training set that the word w appears in, and tf (the term frequency) is the number of times that the word w appears in the query document d (the document you want to transform or encode). Both classes also apply L2 normalization after computing the tf–idf representation; in other words, they rescale the representation of each document to have Euclidean norm 1. Rescaling in this way means that the length of a document (the number of words) does not change the vectorized representation.

Because tf–idf actually makes use of the statistical properties of the training data, we will use a pipeline, as described in [Chapter 6](#), to ensure the results of our grid search are valid. This leads to the following code:

⁷ We provide this formula here mostly for completeness; you don't need to remember it to use the tf–idf encoding.

In[23]:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
pipe = make_pipeline(TfidfVectorizer(min_df=5, norm=None),
                      LogisticRegression())
param_grid = {'logisticregression__C': [0.001, 0.01, 0.1, 1, 10]}

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
```

Out[23]:

```
Best cross-validation score: 0.89
```

As you can see, there is some improvement when using tf-idf instead of just word counts. We can also inspect which words tf-idf found most important. Keep in mind that the tf-idf scaling is meant to find words that distinguish documents, but it is a purely unsupervised technique. So, “important” here does not necessarily relate to the “positive review” and “negative review” labels we are interested in. First, we extract the `TfidfVectorizer` from the pipeline:

In[24]:

```
vectorizer = grid.best_estimator_.named_steps["tfidfvectorizer"]
# transform the training dataset
X_train = vectorizer.transform(text_train)
# find maximum value for each of the features over the dataset
max_value = X_train.max(axis=0).toarray().ravel()
sorted_by_tfidf = max_value.argsort()
# get feature names
feature_names = np.array(vectorizer.get_feature_names())

print("Features with lowest tfidf:\n{}".format(
    feature_names[sorted_by_tfidf[:20]]))

print("Features with highest tfidf: \n{}".format(
    feature_names[sorted_by_tfidf[-20]]))
```

Out[24]:

```
Features with lowest tfidf:
['poignant' 'disagree' 'instantly' 'importantly' 'lacked' 'occurred'
 'currently' 'altogether' 'nearby' 'undoubtedly' 'directs' 'fond' 'stinker'
 'avoided' 'emphasis' 'commented' 'disappoint' 'realizing' 'downhill'
 'inane']

Features with highest tfidf:
['coop' 'homer' 'dillinger' 'hackenstein' 'gadget' 'taker' 'macarthur'
 'vargas' 'jesse' 'basket' 'dominick' 'the' 'victor' 'bridget' 'victoria'
 'khouri' 'zizek' 'rob' 'timon' 'titanic']
```

Features with low tf-idf are those that either are very commonly used across documents or are only used sparingly, and only in very long documents. Interestingly, many of the high-tf-idf features actually identify certain shows or movies. These terms only appear in reviews for this particular show or franchise, but tend to appear very often in these particular reviews. This is very clear, for example, for "pokemon", "smallville", and "doodlebops", but "scanners" here actually also refers to a movie title. These words are unlikely to help us in our sentiment classification task (unless maybe some franchises are universally reviewed positively or negatively) but certainly contain a lot of specific information about the reviews.

We can also find the words that have low inverse document frequency—that is, those that appear frequently and are therefore deemed less important. The inverse document frequency values found on the training set are stored in the `idf_` attribute:

In[25]:

```
sorted_by_idf = np.argsort(vectorizer.idf_)
print("Features with lowest idf:\n{}".format(
    feature_names[sorted_by_idf[:100]]))
```

Out[25]:

```
Features with lowest idf:
['the' 'and' 'of' 'to' 'this' 'is' 'it' 'in' 'that' 'but' 'for' 'with'
 'was' 'as' 'on' 'movie' 'not' 'have' 'one' 'be' 'film' 'are' 'you' 'all'
 'at' 'an' 'by' 'so' 'from' 'like' 'who' 'they' 'there' 'if' 'his' 'out'
 'just' 'about' 'he' 'or' 'has' 'what' 'some' 'good' 'can' 'more' 'when'
 'time' 'up' 'very' 'even' 'only' 'no' 'would' 'my' 'see' 'really' 'story'
 'which' 'well' 'had' 'me' 'than' 'much' 'their' 'get' 'were' 'other'
 'been' 'do' 'most' 'don' 'her' 'also' 'into' 'first' 'made' 'how' 'great'
 'because' 'will' 'people' 'make' 'way' 'could' 'we' 'bad' 'after' 'any'
 'too' 'then' 'them' 'she' 'watch' 'think' 'acting' 'movies' 'seen' 'its'
 'him']
```

As expected, these are mostly English stopwords like "the" and "no". But some are clearly domain-specific to the movie reviews, like "movie", "film", "time", "story", and so on. Interestingly, "good", "great", and "bad" are also among the most frequent and therefore "least relevant" words according to the tf-idf measure, even though we might expect these to be very important for our sentiment analysis task.

Investigating Model Coefficients

Finally, let's look in a bit more detail into what our logistic regression model actually learned from the data. Because there are so many features—27,271 after removing the infrequent ones—we clearly cannot look at all of the coefficients at the same time. However, we can look at the largest coefficients, and see which words these correspond to. We will use the last model that we trained, based on the tf-idf features.

The following bar chart (Figure 7-2) shows the 25 largest and 25 smallest coefficients of the logistic regression model, with the bars showing the size of each coefficient:

In[26]:

```
mlearn.tools.visualize_coefficients(  
    grid.best_estimator_.named_steps["logisticregression"].coef_,  
    feature_names, n_top_features=40)
```

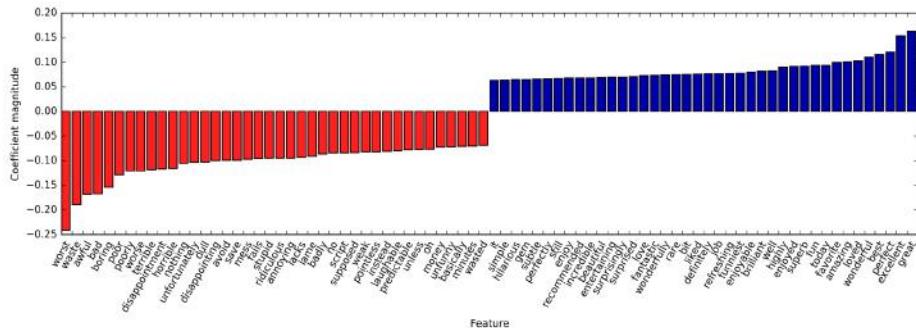


Figure 7-2. Largest and smallest coefficients of logistic regression trained on tf-idf features

The negative coefficients on the left belong to words that according to the model are indicative of negative reviews, while the positive coefficients on the right belong to words that according to the model indicate positive reviews. Most of the terms are quite intuitive, like "worst", "waste", "disappointment", and "laughable" indicating bad movie reviews, while "excellent", "wonderful", "enjoyable", and "refreshing" indicate positive movie reviews. Some words are slightly less clear, like "bit", "job", and "today", but these might be part of phrases like "good job" or "best today."

Bag-of-Words with More Than One Word (n-Grams)

One of the main disadvantages of using a bag-of-words representation is that word order is completely discarded. Therefore, the two strings "it's bad, not good at all" and "it's good, not bad at all" have exactly the same representation, even though the meanings are inverted. Putting "not" in front of a word is only one example (if an extreme one) of how context matters. Fortunately, there is a way of capturing context when using a bag-of-words representation, by not only considering the counts of single tokens, but also the counts of pairs or triplets of tokens that appear next to each other. Pairs of tokens are known as *bigrams*, triplets of tokens are known as *trigrams*, and more generally sequences of tokens are known as *n-grams*. We can change the range of tokens that are considered as features by changing the `ngram_range` parameter of `CountVectorizer` or `TfidfVectorizer`. The `ngram_range` parameter is a tuple, con-

sisting of the minimum length and the maximum length of the sequences of tokens that are considered. Here is an example on the toy data we used earlier:

In[27]:

```
print("bards_words:\n{}".format(bards_words))
```

Out[27]:

```
bards_words:  
['The fool doth think he is wise,',  
 'but the wise man knows himself to be a fool']
```

The default is to create one feature per sequence of tokens that is at least one token long and at most one token long, or in other words exactly one token long (single tokens are also called *unigrams*):

In[28]:

```
cv = CountVectorizer(ngram_range=(1, 1)).fit(bards_words)  
print("Vocabulary size: {}".format(len(cv.vocabulary_)))  
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

Out[28]:

```
Vocabulary size: 13  
Vocabulary:  
['be', 'but', 'doth', 'fool', 'he', 'himself', 'is', 'knows', 'man', 'the',  
 'think', 'to', 'wise']
```

To look only at bigrams—that is, only at sequences of two tokens following each other—we can set `ngram_range` to (2, 2):

In[29]:

```
cv = CountVectorizer(ngram_range=(2, 2)).fit(bards_words)  
print("Vocabulary size: {}".format(len(cv.vocabulary_)))  
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

Out[29]:

```
Vocabulary size: 14  
Vocabulary:  
['be fool', 'but the', 'doth think', 'fool doth', 'he is', 'himself to',  
 'is wise', 'knows himself', 'man knows', 'the fool', 'the wise',  
 'think he', 'to be', 'wise man']
```

Using longer sequences of tokens usually results in many more features, and in more specific features. There is no common bigram between the two phrases in `bard_words`:

In[30]:

```
print("Transformed data (dense):\n{}".format(cv.transform(bards_words).toarray()))
```

Out[30]:

```
Transformed data (dense):
[[0 0 1 1 1 0 1 0 0 1 0 1 0 0]
 [1 1 0 0 0 1 0 1 1 0 1 0 1 1]]
```

For most applications, the minimum number of tokens should be one, as single words often capture a lot of meaning. Adding bigrams helps in most cases. Adding longer sequences—up to 5-grams—might help too, but this will lead to an explosion of the number of features and might lead to overfitting, as there will be many very specific features. In principle, the number of bigrams could be the number of unigrams squared and the number of trigrams could be the number of unigrams to the power of three, leading to very large feature spaces. In practice, the number of higher n -grams that actually appear in the data is much smaller, because of the structure of the (English) language, though it is still large.

Here is what using unigrams, bigrams, and trigrams on `bards_words` looks like:

In[31]:

```
cv = CountVectorizer(ngram_range=(1, 3)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names_()))
```

Out[31]:

```
Vocabulary size: 39
Vocabulary:
['be', 'be fool', 'but', 'but the', 'but the wise', 'doth', 'doth think',
 'doth think he', 'fool', 'fool doth', 'fool doth think', 'he', 'he is',
 'he is wise', 'himself', 'himself to', 'himself to be', 'is', 'is wise',
 'knows', 'knows himself', 'knows himself to', 'man', 'man knows',
 'man knows himself', 'the', 'the fool', 'the fool doth', 'the wise',
 'the wise man', 'think', 'think he', 'think he is', 'to', 'to be',
 'to be fool', 'wise', 'wise man', 'wise man knows']
```

Let's try out the `TfidfVectorizer` on the IMDb movie review data and find the best setting of n -gram range using a grid search:

In[32]:

```
pipe = make_pipeline(TfidfVectorizer(min_df=5), LogisticRegression())
# running the grid search takes a long time because of the
# relatively large grid and the inclusion of trigrams
param_grid = {"logisticregression__C": [0.001, 0.01, 0.1, 1, 10, 100],
              "tfidfvectorizer__ngram_range": [(1, 1), (1, 2), (1, 3)]}

grid = GridSearchCV(pipe, param_grid, cv=5)
grid.fit(text_train, y_train)
print("Best cross-validation score: {:.2f}".format(grid.best_score_))
print("Best parameters:\n{}".format(grid.best_params_))
```

Out[32]:

```
Best cross-validation score: 0.91
Best parameters:
{'tfidfvectorizer__ngram_range': (1, 3), 'logisticregression__C': 100}
```

As you can see from the results, we improved performance by a bit more than a percent by adding bigram and trigram features. We can visualize the cross-validation accuracy as a function of the `ngram_range` and `C` parameter as a heat map, as we did in Chapter 5 (see Figure 7-3):

In[33]:

```
# extract scores from grid_search
scores = grid.cv_results_['mean_test_score'].reshape(-1, 3).T
# visualize heat map
heatmap = mglearn.tools.heatmap(
    scores, xlabel="C", ylabel="ngram_range", cmap="viridis", fmt=".3f",
    xticklabels=param_grid['logisticregression__C'],
    yticklabels=param_grid['tfidfvectorizer__ngram_range'])
plt.colorbar(heatmap)
```

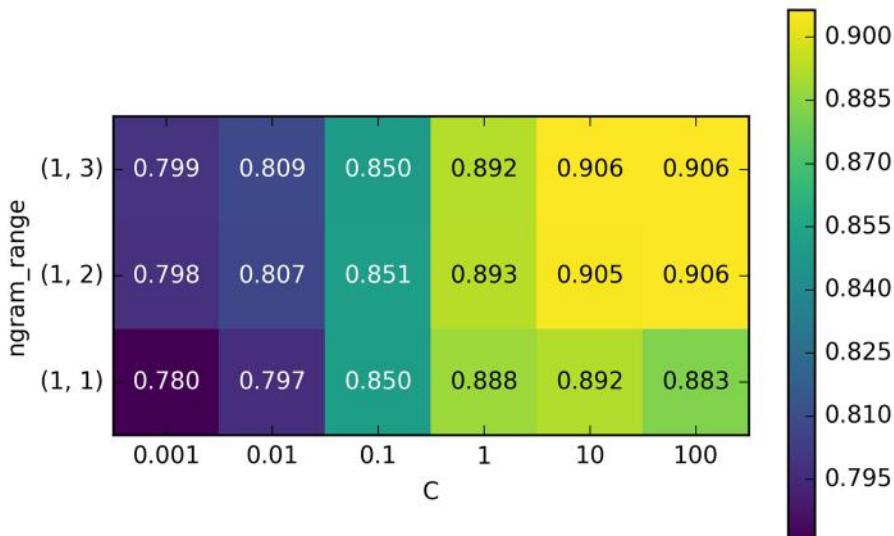


Figure 7-3. Heat map visualization of mean cross-validation accuracy as a function of the parameters `ngram_range` and `C`

From the heat map we can see that using bigrams increases performance quite a bit, while adding trigrams only provides a very small benefit in terms of accuracy. To understand better how the model improved, we can visualize the important coeffi-

cient for the best model, which includes unigrams, bigrams, and trigrams (see Figure 7-4):

In[34]:

```
# extract feature names and coefficients
vect = grid.best_estimator_.named_steps['tfidfvectorizer']
feature_names = np.array(vect.get_feature_names())
coef = grid.best_estimator_.named_steps['logisticregression'].coef_
mglearn.tools.visualize_coefficients(coef, feature_names, n_top_features=40)
```

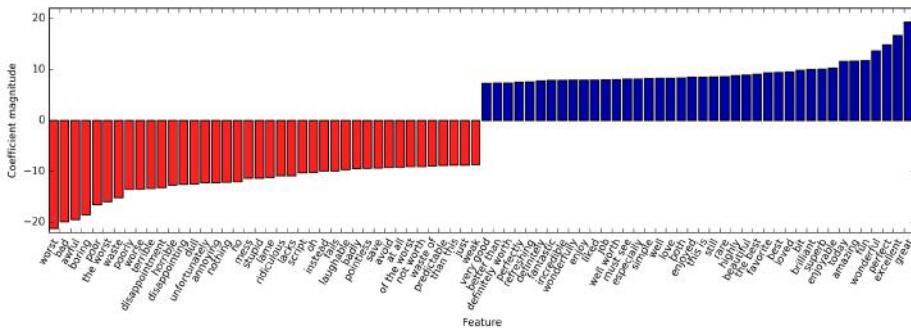


Figure 7-4. Most important features when using unigrams, bigrams, and trigrams with tf-idf rescaling

There are particularly interesting features containing the word “worth” that were not present in the unigram model: “not worth” is indicative of a negative review, while “definitely worth” and “well worth” are indicative of a positive review. This is a prime example of context influencing the meaning of the word “worth.”

Next, we’ll visualize only trigrams, to provide further insight into why these features are helpful. Many of the useful bigrams and trigrams consist of common words that would not be informative on their own, as in the phrases “none of the”, “the only good”, “on and on”, “this is one”, “of the most”, and so on. However, the impact of these features is quite limited compared to the importance of the unigram features, as you can see in Figure 7-5:

In[35]:

```
# find 3-gram features
mask = np.array([len(feature.split(" ")) for feature in feature_names]) == 3
# visualize only 3-gram features
mglearn.tools.visualize_coefficients(coef.ravel()[mask],
                                      feature_names[mask], n_top_features=40)
```

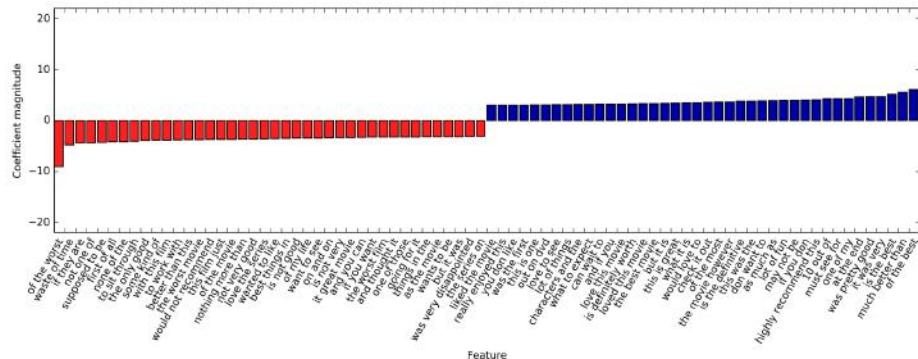


Figure 7-5. Visualization of only the important trigram features of the model

Advanced Tokenization, Stemming, and Lemmatization

As mentioned previously, the feature extraction in the `CountVectorizer` and `TfidfVectorizer` is relatively simple, and much more elaborate methods are possible. One particular step that is often improved in more sophisticated text-processing applications is the first step in the bag-of-words model: tokenization. This step defines what constitutes a word for the purpose of feature extraction.

We saw earlier that the vocabulary often contains singular and plural versions of some words, as in "drawback" and "drawbacks", "drawer" and "drawers", and "drawing" and "drawings". For the purposes of a bag-of-words model, the semantics of "drawback" and "drawbacks" are so close that distinguishing them will only increase overfitting, and not allow the model to fully exploit the training data. Similarly, we found the vocabulary includes words like "replace", "replaced", "replacement", "replaces", and "replacing", which are different verb forms and a noun relating to the verb "to replace." Similarly to having singular and plural forms of a noun, treating different verb forms and related words as distinct tokens is disadvantageous for building a model that generalizes well.

This problem can be overcome by representing each word using its *word stem*, which involves identifying (or *conflating*) all the words that have the same word stem. If this is done by using a rule-based heuristic, like dropping common suffixes, it is usually referred to as *stemming*. If instead a dictionary of known word forms is used (an explicit and human-verified system), and the role of the word in the sentence is taken into account, the process is referred to as *lemmatization* and the standardized form of the word is referred to as the *lemma*. Both processing methods, lemmatization and stemming, are forms of *normalization* that try to extract some normal form of a word. Another interesting case of normalization is spelling correction, which can be helpful in practice but is outside of the scope of this book.

To get a better understanding of normalization, let's compare a method for stemming—the Porter stemmer, a widely used collection of heuristics (here imported from the `nltk` package)—to lemmatization as implemented in the `spacy` package:⁸

In[36]:

```
import spacy
import nltk

# load spacy's English-language models
en_nlp = spacy.load('en')
# instantiate nltk's Porter stemmer
stemmer = nltk.stem.PorterStemmer()

# define function to compare lemmatization in spacy with stemming in nltk
def compare_normalization(doc):
    # tokenize document in spacy
    doc_spacy = en_nlp(doc)
    # print lemmas found by spacy
    print("Lemmatization:")
    print([token.lemma_ for token in doc_spacy])
    # print tokens found by Porter stemmer
    print("Stemming:")
    print([stemmer.stem(token.norm_.lower()) for token in doc_spacy])
```

We will compare lemmatization and the Porter stemmer on a sentence designed to show some of the differences:

In[37]:

```
compare_normalization(u"Our meeting today was worse than yesterday, "
                      "I'm scared of meeting the clients tomorrow.")
```

Out[37]:

```
Lemmatization:
['our', 'meeting', 'today', 'be', 'bad', 'than', 'yesterday', ',', 'i', 'be',
 'scared', 'of', 'meet', 'the', 'client', 'tomorrow', '.']
Stemming:
['our', 'meet', 'today', 'wa', 'wors', 'than', 'yesterday', ',', 'i', 'm',
 'scare', 'of', 'meet', 'the', 'client', 'tomorrow', '.']
```

Stemming is always restricted to trimming the word to a stem, so "was" becomes "wa", while lemmatization can retrieve the correct base verb form, "be". Similarly, lemmatization can normalize "worse" to "bad", while stemming produces "wors". Another major difference is that stemming reduces both occurrences of "meeting" to "meet". Using lemmatization, the first occurrence of "meeting" is recognized as a

⁸ For details of the interface, consult the `nltk` and `spacy` documentation. We are more interested in the general principles here.

noun and left as is, while the second occurrence is recognized as a verb and reduced to "meet". In general, lemmatization is a much more involved process than stemming, but it usually produces better results than stemming when used for normalizing tokens for machine learning.

While scikit-learn implements neither form of normalization, CountVectorizer allows specifying your own tokenizer to convert each document into a list of tokens using the `tokenizer` parameter. We can use the lemmatization from spacy to create a callable that will take a string and produce a list of lemmas:

In[38]:

```
# Technicality: we want to use the regexp-based tokenizer
# that is used by CountVectorizer and only use the lemmatization
# from spacy. To this end, we replace en_nlp.tokenizer (the spacy tokenizer)
# with the regexp-based tokenization.
import re
# regexp used in CountVectorizer
regexp = re.compile('(?u)\\b\\w\\w+\\b')

# load spacy language model and save old tokenizer
en_nlp = spacy.load('en')
old_tokenizer = en_nlp.tokenizer
# replace the tokenizer with the preceding regexp
en_nlp.tokenizer = lambda string: old_tokenizer.tokens_from_list(
    regexp.findall(string))

# create a custom tokenizer using the spacy document processing pipeline
# (now using our own tokenizer)
def custom_tokenizer(document):
    doc_spacy = en_nlp(document, entity=False, parse=False)
    return [token.lemma_ for token in doc_spacy]

# define a count vectorizer with the custom tokenizer
lemma_vect = CountVectorizer(tokenizer=custom_tokenizer, min_df=5)
```

Let's transform the data and inspect the vocabulary size:

In[39]:

```
# transform text_train using CountVectorizer with lemmatization
X_train_lemma = lemma_vect.fit_transform(text_train)
print("X_train_lemma.shape: {}".format(X_train_lemma.shape))

# standard CountVectorizer for reference
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train.shape: {}".format(X_train.shape))
```

Out[39]:

```
X_train_lemma.shape: (25000, 21596)
X_train.shape: (25000, 27271)
```

As you can see from the output, lemmatization reduced the number of features from 27,271 (with the standard `CountVectorizer` processing) to 21,596. Lemmatization can be seen as a kind of regularization, as it conflates certain features. Therefore, we expect lemmatization to improve performance most when the dataset is small. To illustrate how lemmatization can help, we will use `StratifiedShuffleSplit` for cross-validation, using only 1% of the data as training data and the rest as test data:

In[40]:

```
# build a grid search using only 1% of the data as the training set
from sklearn.model_selection import StratifiedShuffleSplit

param_grid = {'C': [0.001, 0.01, 0.1, 1, 10]}
cv = StratifiedShuffleSplit(n_iter=5, test_size=0.99,
                           train_size=0.01, random_state=0)
grid = GridSearchCV(LogisticRegression(), param_grid, cv=cv)
# perform grid search with standard CountVectorizer
grid.fit(X_train, y_train)
print("Best cross-validation score "
      "(standard CountVectorizer): {:.3f}".format(grid.best_score_))
# perform grid search with lemmatization
grid.fit(X_train_lemma, y_train)
print("Best cross-validation score "
      "(lemmatization): {:.3f}".format(grid.best_score_))
```

Out[40]:

```
Best cross-validation score (standard CountVectorizer): 0.721
Best cross-validation score (lemmatization): 0.731
```

In this case, lemmatization provided a modest improvement in performance. As with many of the different feature extraction techniques, the result varies depending on the dataset. Lemmatization and stemming can sometimes help in building better (or at least more compact) models, so we suggest you give these techniques a try when trying to squeeze out the last bit of performance on a particular task.

Topic Modeling and Document Clustering

One particular technique that is often applied to text data is *topic modeling*, which is an umbrella term describing the task of assigning each document to one or multiple *topics*, usually without supervision. A good example for this is news data, which might be categorized into topics like “politics,” “sports,” “finance,” and so on. If each document is assigned a single topic, this is the task of clustering the documents, as discussed in [Chapter 3](#). If each document can have more than one topic, the task

relates to the decomposition methods from [Chapter 3](#). Each of the components we learn then corresponds to one topic, and the coefficients of the components in the representation of a document tell us how strongly related that document is to a particular topic. Often, when people talk about topic modeling, they refer to one particular decomposition method called *Latent Dirichlet Allocation* (often LDA for short).⁹

Latent Dirichlet Allocation

Intuitively, the LDA model tries to find groups of words (the topics) that appear together frequently. LDA also requires that each document can be understood as a “mixture” of a subset of the topics. It is important to understand that for the machine learning model a “topic” might not be what we would normally call a topic in everyday speech, but that it resembles more the components extracted by PCA or NMF (which we discussed in [Chapter 3](#)), which might or might not have a semantic meaning. Even if there is a semantic meaning for an LDA “topic”, it might not be something we’d usually call a topic. Going back to the example of news articles, we might have a collection of articles about sports, politics, and finance, written by two specific authors. In a politics article, we might expect to see words like “governor,” “vote,” “party,” etc., while in a sports article we might expect words like “team,” “score,” and “season.” Words in each of these groups will likely appear together, while it’s less likely that, for example, “team” and “governor” will appear together. However, these are not the only groups of words we might expect to appear together. The two reporters might prefer different phrases or different choices of words. Maybe one of them likes to use the word “demarcate” and one likes the word “polarize.” Other “topics” would then be “words often used by reporter A” and “words often used by reporter B,” though these are not topics in the usual sense of the word.

Let’s apply LDA to our movie review dataset to see how it works in practice. For unsupervised text document models, it is often good to remove very common words, as they might otherwise dominate the analysis. We’ll remove words that appear in at least 20 percent of the documents, and we’ll limit the bag-of-words model to the 10,000 words that are most common after removing the top 20 percent:

In[41]:

```
vect = CountVectorizer(max_features=10000, max_df=.15)
X = vect.fit_transform(text_train)
```

⁹ There is another machine learning model that is also often abbreviated LDA: Linear Discriminant Analysis, a linear classification model. This leads to quite some confusion. In this book, LDA refers to Latent Dirichlet Allocation.

We will learn a topic model with 10 topics, which is few enough that we can look at all of them. Similarly to the components in NMF, topics don't have an inherent ordering, and changing the number of topics will change all of the topics.¹⁰ We'll use the "batch" learning method, which is somewhat slower than the default ("online") but usually provides better results, and increase "max_iter", which can also lead to better models:

In[42]:

```
from sklearn.decomposition import LatentDirichletAllocation
lda = LatentDirichletAllocation(n_topics=10, learning_method="batch",
                                 max_iter=25, random_state=0)
# We build the model and transform the data in one step
# Computing transform takes some time,
# and we can save time by doing both at once
document_topics = lda.fit_transform(X)
```

Like the decomposition methods we saw in Chapter 3, `LatentDirichletAllocation` has a `components_` attribute that stores how important each word is for each topic. The size of `components_` is (`n_topics`, `n_words`):

In[43]:

```
lda.components_.shape
```

Out[43]:

```
(10, 10000)
```

To understand better what the different topics mean, we will look at the most important words for each of the topics. The `print_topics` function provides a nice formatting for these features:

In[44]:

```
# For each topic (a row in the components_), sort the features (ascending)
# Invert rows with [:, ::-1] to make sorting descending
sorting = np.argsort(lda.components_, axis=1)[:, ::-1]
# Get the feature names from the vectorizer
feature_names = np.array(vect.get_feature_names())
```

In[45]:

```
# Print out the 10 topics:
mglearn.tools.print_topics(topics=range(10), feature_names=feature_names,
                           sorting=sorting, topics_per_chunk=5, n_words=10)
```

¹⁰ In fact, NMF and LDA solve quite related problems, and we could also use NMF to extract topics.

Out[45]:

topic 0	topic 1	topic 2	topic 3	topic 4
-----	-----	-----	-----	-----
between	war	funny	show	didn
young	world	worst	series	saw
family	us	comedy	episode	am
real	our	thing	tv	thought
performance	american	guy	episodes	years
beautiful	documentary	re	shows	book
work	history	stupid	season	watched
each	new	actually	new	now
both	own	nothing	television	dvd
director	point	want	years	got
-----	-----	-----	-----	-----
topic 5	topic 6	topic 7	topic 8	topic 9
-----	-----	-----	-----	-----
horror	kids	cast	performance	house
action	action	role	role	woman
effects	animation	john	john	gets
budget	game	version	actor	killer
nothing	fun	novel	oscar	girl
original	disney	both	cast	wife
director	children	director	plays	horror
minutes	10	played	jack	young
pretty	kid	performance	joe	goes
doesn	old	mr	performances	around

Judging from the important words, topic 1 seems to be about historical and war movies, topic 2 might be about bad comedies, topic 3 might be about TV series. Topic 4 seems to capture some very common words, while topic 6 appears to be about children's movies and topic 8 seems to capture award-related reviews. Using only 10 topics, each of the topics needs to be very broad, so that they can together cover all the different kinds of reviews in our dataset.

Next, we will learn another model, this time with 100 topics. Using more topics makes the analysis much harder, but makes it more likely that topics can specialize to interesting subsets of the data:

In[46]:

```
lda100 = LatentDirichletAllocation(n_topics=100, learning_method="batch",
                                    max_iter=25, random_state=0)
document_topics100 = lda100.fit_transform(X)
```

Looking at all 100 topics would be a bit overwhelming, so we selected some interesting and representative topics:

In[47]:

```
topics = np.array([7, 16, 24, 25, 28, 36, 37, 45, 51, 53, 54, 63, 89, 97])  
sorting = np.argsort(lda100.components_, axis=1)[:, ::-1]  
feature_names = np.array(vect.get_feature_names())  
mglearn.tools.print_topics(topics=topics, feature_names=feature_names,  
                           sorting=sorting, topics_per_chunk=7, n_words=20)
```

Out[48]:

topic 7	topic 16	topic 24	topic 25	topic 28
thriller	worst	german	car	beautiful
suspense	awful	hitler	gets	young
horror	boring	nazi	guy	old
atmosphere	horrible	midnight	around	romantic
mystery	stupid	joe	down	between
house	thing	germany	kill	romance
director	terrible	years	goes	wonderful
quite	script	history	killed	heart
bit	nothing	new	going	feel
de	worse	modesty	house	year
performances	waste	cowboy	away	each
dark	pretty	jewish	head	french
twist	minutes	past	take	sweet
hitchcock	didn	kirk	another	boy
tension	actors	young	getting	loved
interesting	actually	spanish	doesn	girl
mysterious	re	enterprise	now	relationship
murder	supposed	von	night	saw
ending	mean	nazis	right	both
creepy	want	spock	woman	simple
topic 36	topic 37	topic 41	topic 45	topic 51
performance	excellent	war	music	earth
role	highly	american	song	space
actor	amazing	world	songs	planet
cast	wonderful	soldiers	rock	superman
play	truly	military	band	alien
actors	superb	army	soundtrack	world
performances	actors	tarzan	singing	evil
played	brilliant	soldier	voice	humans
supporting	recommend	america	singer	aliens
director	quite	country	sing	human
oscar	performance	americans	musical	creatures
roles	performances	during	roll	miike
actress	perfect	men	fan	monsters
excellent	drama	us	metal	apes
screen	without	government	concert	clark
plays	beautiful	jungle	playing	burton
award	human	vietnam	hear	tim
work	moving	ii	fans	outer
playing	world	political	prince	men
gives	recommended	against	especially	moon

topic 53	topic 54	topic 63	topic 89	topic 97
scott	money	funny	dead	didn
gary	budget	comedy	zombie	thought
streisand	actors	laugh	gore	wasn
star	low	jokes	zombies	ending
hart	worst	humor	blood	minutes
lundgren	waste	hilarious	horror	got
dolph	10	laughs	flesh	felt
career	give	fun	minutes	part
sabrina	want	re	body	going
role	nothing	funniest	living	seemed
temple	terrible	laughing	eating	bit
phantom	crap	joke	flick	found
judy	must	few	budget	though
melissa	reviews	moments	head	nothing
zorro	imdb	guy	gory	lot
gets	director	unfunny	evil	saw
barbra	thing	times	shot	long
cast	believe	laughed	low	interesting
short	am	comedies	fulci	few
serial	actually	isn	re	half

The topics we extracted this time seem to be more specific, though many are hard to interpret. Topic 7 seems to be about horror movies and thrillers; topics 16 and 54 seem to capture bad reviews, while topic 63 mostly seems to be capturing positive reviews of comedies. If we want to make further inferences using the topics that were discovered, we should confirm the intuition we gained from looking at the highest-ranking words for each topic by looking at the documents that are assigned to these topics. For example, topic 45 seems to be about music. Let's check which kinds of reviews are assigned to this topic:

In[49]:

```
# sort by weight of "music" topic 45
music = np.argsort(document_topics100[:, 45])[::-1]
# print the five documents where the topic is most important
for i in music[:10]:
    # pshow first two sentences
    print(b".join(text_train[i].split(b".")[:2]) + b"\n")
```

Out[49]:

```
b'I love this movie and never get tired of watching. The music in it is great.\n'
b"I enjoyed Still Crazy more than any film I have seen in years. A successful
band from the 70's decide to give it another try.\n"
b'Hollywood Hotel was the last movie musical that Busby Berkeley directed for
Warner Bros. His directing style had changed or evolved to the point that
this film does not contain his signature overhead shots or huge production
numbers with thousands of extras.\n'
b"What happens to washed up rock-n-roll stars in the late 1990's?
They launch a comeback / reunion tour. At least, that's what the members of
Strange Fruit, a (fictional) 70's stadium rock group do.\n"
```

```

b'As a big-time Prince fan of the last three to four years, I really can\'t
believe I\'ve only just got round to watching "Purple Rain". The brand new
2-disc anniversary Special Edition led me to buy it.\n'
b"This film is worth seeing alone for Jared Harris' outstanding portrayal
of John Lennon. It doesn't matter that Harris doesn't exactly resemble
Lennon; his mannerisms, expressions, posture, accent and attitude are
pure Lennon.\n"
b"The funky, yet strictly second-tier British glam-rock band Strange Fruit
breaks up at the end of the wild'n'wacky excess-ridden 70's. The individual
band members go their separate ways and uncomfortably settle into lackluster
middle age in the dull and uneventful 90's: morose keyboardist Stephen Rea
winds up penniless and down on his luck, vain, neurotic, pretentious lead
singer Bill Nighy tries (and fails) to pursue a floundering solo career,
paranoid drummer Timothy Spall resides in obscurity on a remote farm so he
can avoid paying a hefty back taxes debt, and surly bass player Jimmy Nail
installs roofs for a living.\n"
b"I just finished reading a book on Anita Loos' work and the photo in TCM
Magazine of MacDonald in her angel costume looked great (impressive wings),
so I thought I'd watch this movie. I'd never heard of the film before, so I
had no preconceived notions about it whatsoever.\n"
b'I love this movie!!! Purple Rain came out the year I was born and it has had
my heart since I can remember. Prince is so tight in this movie.\n'
b"This movie is sort of a Carrie meets Heavy Metal. It's about a highschool
guy who gets picked on a lot and he totally gets revenge with the help of a
Heavy Metal ghost.\n"

```

As we can see, this topic covers a wide variety of music-centered reviews, from musicals, to biographical movies, to some hard-to-specify genre in the last review. Another interesting way to inspect the topics is to see how much weight each topic gets overall, by summing the `document_topics` over all reviews. We name each topic by the two most common words. [Figure 7-6](#) shows the topic weights learned:

In[50]:

```

fig, ax = plt.subplots(1, 2, figsize=(10, 10))
topic_names = ["{:>2} ".format(i) + " ".join(words)
               for i, words in enumerate(feature_names[sorting[:, :2]])]
# two column bar chart:
for col in [0, 1]:
    start = col * 50
    end = (col + 1) * 50
    ax[col].barh(np.arange(50), np.sum(document_topics100, axis=0)[start:end])
    ax[col].set_yticks(np.arange(50))
    ax[col].set_yticklabels(topic_names[start:end], ha="left", va="top")
    ax[col].invert_yaxis()
    ax[col].set_xlim(0, 2000)
    yax = ax[col].get_yaxis()
    yax.set_tick_params(pad=130)
plt.tight_layout()

```

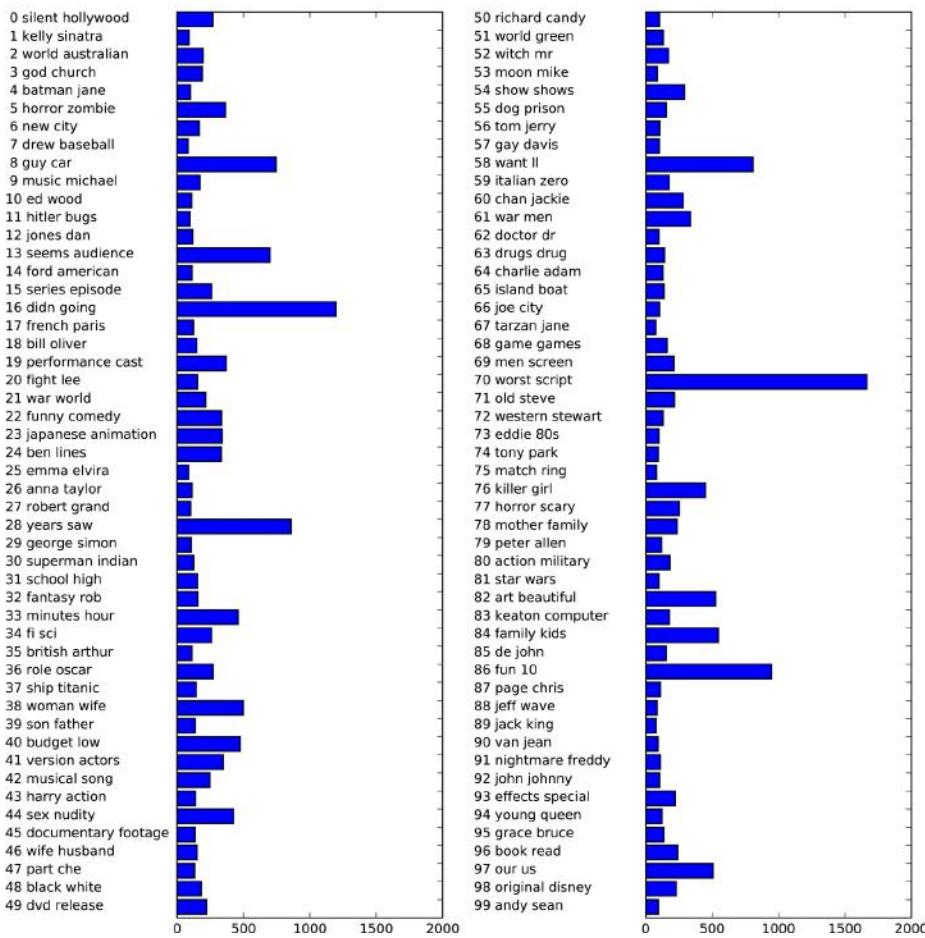


Figure 7-6. Topic weights learned by LDA

The most important topics are 97, which seems to consist mostly of stopwords, possibly with a slight negative direction; topic 16, which is clearly about bad reviews; followed by some genre-specific topics and 36 and 37, both of which seem to contain laudatory words.

It seems like LDA mostly discovered two kind of topics, genre-specific and rating-specific, in addition to several more unspecific topics. This is an interesting discovery, as most reviews are made up of some movie-specific comments and some comments that justify or emphasize the rating.

Topic models like LDA are interesting methods to understand large text corpora in the absence of labels—or, as here, even if labels are available. The LDA algorithm is randomized, though, and changing the `random_state` parameter can lead to quite

different outcomes. While identifying topics can be helpful, any conclusions you draw from an unsupervised model should be taken with a grain of salt, and we recommend verifying your intuition by looking at the documents in a specific topic. The topics produced by the `LDA.transform` method can also sometimes be used as a compact representation for supervised learning. This is particularly helpful when few training examples are available.

Summary and Outlook

In this chapter we talked about the basics of processing text, also known as *natural language processing* (NLP), with an example application classifying movie reviews. The tools discussed here should serve as a great starting point when trying to process text data. In particular for text classification tasks such as spam and fraud detection or sentiment analysis, bag-of-words representations provide a simple and powerful solution. As is often the case in machine learning, the representation of the data is key in NLP applications, and inspecting the tokens and n -grams that are extracted can give powerful insights into the modeling process. In text-processing applications, it is often possible to introspect models in a meaningful way, as we saw in this chapter, for both supervised and unsupervised tasks. You should take full advantage of this ability when using NLP-based methods in practice.

Natural language and text processing is a large research field, and discussing the details of advanced methods is far beyond the scope of this book. If you want to learn more, we recommend the O'Reilly book *Natural Language Processing with Python* by Steven Bird, Ewan Klein, and Edward Loper, which provides an overview of NLP together with an introduction to the `nltk` Python package for NLP. Another great and more conceptual book is the standard reference *Introduction to Information Retrieval* by Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze, which describes fundamental algorithms in information retrieval, NLP, and machine learning. Both books have online versions that can be accessed free of charge. As we discussed earlier, the classes `CountVectorizer` and `TfidfVectorizer` only implement relatively simple text-processing methods. For more advanced text-processing methods, we recommend the Python packages `spacy` (a relatively new but very efficient and well-designed package), `nltk` (a very well-established and complete but somewhat dated library), and `gensim` (an NLP package with an emphasis on topic modeling).

There have been several very exciting new developments in text processing in recent years, which are outside of the scope of this book and relate to neural networks. The first is the use of continuous vector representations, also known as word vectors or distributed word representations, as implemented in the `word2vec` library. The original paper “[Distributed Representations of Words and Phrases and Their Compositionality](#)” by Thomas Mikolov et al. is a great introduction to the subject. Both `spacy`

and `gensim` provide functionality for the techniques discussed in this paper and its follow-ups.

Another direction in NLP that has picked up momentum in recent years is the use of *recurrent neural networks* (RNNs) for text processing. RNNs are a particularly powerful type of neural network that can produce output that is again text, in contrast to classification models that can only assign class labels. The ability to produce text as output makes RNNs well suited for automatic translation and summarization. An introduction to the topic can be found in the relatively technical paper “[Sequence to Sequence Learning with Neural Networks](#)” by Ilya Sutskever, Oriol Vinyals, and Quoc Le. A more practical tutorial using the `tensorflow` framework can be found on the [TensorFlow website](#).

Wrapping Up

You now know how to apply the important machine learning algorithms for supervised and unsupervised learning, which allow you to solve a wide variety of machine learning problems. Before we leave you to explore all the possibilities that machine learning offers, we want to give you some final words of advice, point you toward some additional resources, and give you suggestions on how you can further improve your machine learning and data science skills.

Approaching a Machine Learning Problem

With all the great methods that we introduced in this book now at your fingertips, it may be tempting to jump in and start solving your data-related problem by just running your favorite algorithm. However, this is not usually a good way to begin your analysis. The machine learning algorithm is usually only a small part of a larger data analysis and decision-making process. To make effective use of machine learning, we need to take a step back and consider the problem at large. First, you should think about what kind of question you want to answer. Do you want to do exploratory analysis and just see if you find something interesting in the data? Or do you already have a particular goal in mind? Often you will start with a goal, like detecting fraudulent user transactions, making movie recommendations, or finding unknown planets. If you have such a goal, before building a system to achieve it, you should first think about how to define and measure success, and what the impact of a successful solution would be to your overall business or research goals. Let's say your goal is fraud detection.

Then the following questions open up:

- How do I measure if my fraud prediction is actually working?
- Do I have the right data to evaluate an algorithm?
- If I am successful, what will be the business impact of my solution?

As we discussed in [Chapter 5](#), it is best if you can measure the performance of your algorithm directly using a business metric, like increased profit or decreased losses. This is often hard to do, though. A question that can be easier to answer is “What if I built the perfect model?” If perfectly detecting any fraud will save your company \$100 a month, these possible savings will probably not be enough to warrant the effort of you even starting to develop an algorithm. On the other hand, if the model might save your company tens of thousands of dollars every month, the problem might be worth exploring.

Say you’ve defined the problem to solve, you know a solution might have a significant impact for your project, and you’ve ensured that you have the right information to evaluate success. The next steps are usually acquiring the data and building a working prototype. In this book we have talked about many models you can employ, and how to properly evaluate and tune these models. While trying out models, though, keep in mind that this is only a small part of a larger data science workflow, and model building is often part of a feedback circle of collecting new data, cleaning data, building models, and analyzing the models. Analyzing the mistakes a model makes can often be informative about what is missing in the data, what additional data could be collected, or how the task could be reformulated to make machine learning more effective. Collecting more or different data or changing the task formulation slightly might provide a much higher payoff than running endless grid searches to tune parameters.

Humans in the Loop

You should also consider if and how you should have humans in the loop. Some processes (like pedestrian detection in a self-driving car) need to make immediate decisions. Others might not need immediate responses, and so it can be possible to have humans confirm uncertain decisions. Medical applications, for example, might need very high levels of precision that possibly cannot be achieved by a machine learning algorithm alone. But if an algorithm can make 90 percent, 50 percent, or maybe even just 10 percent of decisions automatically, that might already increase response time or reduce cost. Many applications are dominated by “simple cases,” for which an algorithm can make a decision, with relatively few “complicated cases,” which can be rerouted to a human.

From Prototype to Production

The tools we've discussed in this book are great for many machine learning applications, and allow very quick analysis and prototyping. Python and `scikit-learn` are also used in production systems in many organizations—even very large ones like international banks and global social media companies. However, many companies have complex infrastructure, and it is not always easy to include Python in these systems. That is not necessarily a problem. In many companies, the data analytics teams work with languages like Python and R that allow the quick testing of ideas, while production teams work with languages like Go, Scala, C++, and Java to build robust, scalable systems. Data analysis has different requirements from building live services, and so using different languages for these tasks makes sense. A relatively common solution is to reimplement the solution that was found by the analytics team inside the larger framework, using a high-performance language. This can be easier than embedding a whole library or programming language and converting from and to the different data formats.

Regardless of whether you can use `scikit-learn` in a production system or not, it is important to keep in mind that production systems have different requirements from one-off analysis scripts. If an algorithm is deployed into a larger system, software engineering aspects like reliability, predictability, runtime, and memory requirements gain relevance. Simplicity is key in providing machine learning systems that perform well in these areas. Critically inspect each part of your data processing and prediction pipeline and ask yourself how much complexity each step creates, how robust each component is to changes in the data or compute infrastructure, and if the benefit of each component warrants the complexity. If you are building involved machine learning systems, we highly recommend reading the paper "[Machine Learning: The High Interest Credit Card of Technical Debt](#)", published by researchers in Google's machine learning team. The paper highlights the trade-off in creating and maintaining machine learning software in production at a large scale. While the issue of technical debt *is* particularly pressing in large-scale and long-term projects, the lessons learned can help us build better software even for short-lived and smaller systems.

Testing Production Systems

In this book, we covered how to evaluate algorithmic predictions based on a test set that we collected beforehand. This is known as *offline evaluation*. If your machine learning system is user-facing, this is only the first step in evaluating an algorithm, though. The next step is usually *online testing* or *live testing*, where the consequences of employing the algorithm in the overall system are evaluated. Changing the recommendations or search results users are shown by a website can drastically change their behavior and lead to unexpected consequences. To protect against these surprises, most user-facing services employ *A/B testing*, a form of blind user study. In

A/B testing, without their knowledge a selected portion of users will be provided with a website or service using algorithm A, while the rest of the users will be provided with algorithm B. For both groups, relevant success metrics will be recorded for a set period of time. Then, the metrics of algorithm A and algorithm B will be compared, and a selection between the two approaches will be made according to these metrics. Using A/B testing enables us to evaluate the algorithms “in the wild,” which might help us to discover unexpected consequences when users are interacting with our model. Often A is a new model, while B is the established system. There are more elaborate mechanisms for online testing that go beyond A/B testing, such as *bandit algorithms*. A great introduction to this subject can be found in the book *Bandit Algorithms for Website Optimization* by John Myles White (O'Reilly).

Building Your Own Estimator

This book has covered a variety of tools and algorithms implemented in `scikit-learn` that can be used on a wide range of tasks. However, often there will be some particular processing you need to do for your data that is not implemented in `scikit-learn`. It may be enough to just preprocess your data before passing it to your `scikit-learn` model or pipeline. However, if your preprocessing is data dependent, and you want to apply a grid search or cross-validation, things become trickier.

In Chapter 6 we discussed the importance of putting all data-dependent processing inside the cross-validation loop. So how can you use your own processing together with the `scikit-learn` tools? There is a simple solution: build your own estimator! Implementing an estimator that is compatible with the `scikit-learn` interface, so that it can be used with `Pipeline`, `GridSearchCV`, and `cross_val_score`, is quite easy. You can find detailed instructions in [the `scikit-learn` documentation](#), but here is the gist. The simplest way to implement a transformer class is by inheriting from `BaseEstimator` and `TransformerMixin`, and then implementing the `__init__`, `fit`, and `predict` functions like this:

In[1]:

```
from sklearn.base import BaseEstimator, TransformerMixin

class MyTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, first_parameter=1, second_parameter=2):
        # All parameters must be specified in the __init__ function
        self.first_parameter = 1
        self.second_parameter = 2

    def fit(self, X, y=None):
        # fit should only take X and y as parameters
        # Even if your model is unsupervised, you need to accept a y argument!

        # Model fitting code goes here
        print("fitting the model right here")
        # fit returns self
        return self

    def transform(self, X):
        # transform takes as parameter only X

        # Apply some transformation to X
        X_transformed = X + 1
        return X_transformed
```

Implementing a classifier or regressor works similarly, only instead of `TransformerMixin` you need to inherit from `ClassifierMixin` or `RegressorMixin`. Also, instead of implementing `transform`, you would implement `predict`.

As you can see from the example given here, implementing your own estimator requires very little code, and most `scikit-learn` users build up a collection of custom models over time.

Where to Go from Here

This book provides an introduction to machine learning and will make you an effective practitioner. However, if you want to further your machine learning skills, here are some suggestions of books and more specialized resources to investigate to dive deeper.

Theory

In this book, we tried to provide an intuition of how the most common machine learning algorithms work, without requiring a strong foundation in mathematics or computer science. However, many of the models we discussed use principles from probability theory, linear algebra, and optimization. While it is not necessary to understand all the details of how these algorithms are implemented, we think that

knowing some of the theory behind the algorithms will make you a better data scientist. There have been many good books written about the theory of machine learning, and if we were able to excite you about the possibilities that machine learning opens up, we suggest you pick up at least one of them and dig deeper. We already mentioned Hastie, Tibshirani, and Friedman's book *The Elements of Statistical Learning* in the Preface, but it is worth repeating this recommendation here. Another quite accessible book, with accompanying Python code, is *Machine Learning: An Algorithmic Perspective* by Stephen Marsland (Chapman and Hall/CRC). Two other highly recommended classics are *Pattern Recognition and Machine Learning* by Christopher Bishop (Springer), a book that emphasizes a probabilistic framework, and *Machine Learning: A Probabilistic Perspective* by Kevin Murphy (MIT Press), a comprehensive (read: 1,000+ pages) dissertation on machine learning methods featuring in-depth discussions of state-of-the-art approaches, far beyond what we could cover in this book.

Other Machine Learning Frameworks and Packages

While `scikit-learn` is our favorite package for machine learning¹ and Python is our favorite language for machine learning, there are many other options out there. Depending on your needs, Python and `scikit-learn` might not be the best fit for your particular situation. Often using Python is great for trying out and evaluating models, but larger web services and applications are more commonly written in Java or C++, and integrating into these systems might be necessary for your model to be deployed. Another reason you might want to look beyond `scikit-learn` is if you are more interested in statistical modeling and inference than prediction. In this case, you should consider the `statsmodel` package for Python, which implements several linear models with a more statistically minded interface. If you are not married to Python, you might also consider using R, another lingua franca of data scientists. R is a language designed specifically for statistical analysis and is famous for its excellent visualization capabilities and the availability of many (often highly specialized) statistical modeling packages.

Another popular machine learning package is `vowpal wabbit` (often called `vw` to avoid possible tongue twisting), a highly optimized machine learning package written in C++ with a command-line interface. `vw` is particularly useful for large datasets and for streaming data. For running machine learning algorithms distributed on a cluster, one of the most popular solutions at the time of writing is `mllib`, a Scala library built on top of the `spark` distributed computing environment.

¹ Andreas might not be entirely objective in this matter.

Ranking, Recommender Systems, and Other Kinds of Learning

Because this is an introductory book, we focused on the most common machine learning tasks: classification and regression in supervised learning, and clustering and signal decomposition in unsupervised learning. There are many more kinds of machine learning out there, with many important applications. There are two particularly important topics that we did not cover in this book. The first is *ranking*, in which we want to retrieve answers to a particular query, ordered by their relevance. You've probably already used a ranking system today; this is how search engines operate. You input a search query and obtain a sorted list of answers, ranked by how relevant they are. A great introduction to ranking is provided in Manning, Raghavan, and Schütze's book *Introduction to Information Retrieval*. The second topic is *recommender systems*, which provide suggestions to users based on their preferences. You've probably encountered recommender systems under headings like "People You May Know," "Customers Who Bought This Item Also Bought," or "Top Picks for You." There is plenty of literature on the topic, and if you want to dive right in you might be interested in the now classic "[Netflix prize challenge](#)", in which the Netflix video streaming site released a large dataset of movie preferences and offered a prize of \$1 million to the team that could provide the best recommendations. Another common application is prediction of time series (like stock prices), which also has a whole body of literature devoted to it. There are many more machine learning tasks out there—much more than we can list here—and we encourage you to seek out information from books, research papers, and online communities to find the paradigms that best apply to your situation.

Probabilistic Modeling, Inference, and Probabilistic Programming

Most machine learning packages provide predefined machine learning models that apply one particular algorithm. However, many real-world problems have a particular structure that, when properly incorporated into the model, can yield much better-performing predictions. Often, the structure of a particular problem can be expressed using the language of probability theory. Such structure commonly arises from having a mathematical model of the situation for which you want to predict. To understand what we mean by a structured problem, consider the following example.

Let's say you want to build a mobile application that provides a very detailed position estimate in an outdoor space, to help users navigate a historical site. A mobile phone provides many sensors to help you get precise location measurements, like the GPS, accelerometer, and compass. You also have an exact map of the area. This problem is highly structured. You know where the paths and points of interest are from your map. You also have rough positions from the GPS, and the accelerometer and compass in the user's device provide you with very precise relative measurements. But throwing these all together into a black-box machine learning system to predict positions might not be the best idea. This would throw away all the information you

already know about how the real world works. If the compass and accelerometer tell you a user is going north, and the GPS is telling you the user is going south, you probably can't trust the GPS. If your position estimate tells you the user just walked through a wall, you should also be highly skeptical. It's possible to express this situation using a probabilistic model, and then use machine learning or probabilistic inference to find out how much you should trust each measurement, and to reason about what the best guess for the location of a user is.

Once you've expressed the situation and your model of how the different factors work together in the right way, there are methods to compute the predictions using these custom models directly. The most general of these methods are called probabilistic programming languages, and they provide a very elegant and compact way to express a learning problem. Examples of popular probabilistic programming languages are PyMC (which can be used in Python) and Stan (a framework that can be used from several languages, including Python). While these packages require some understanding of probability theory, they simplify the creation of new models significantly.

Neural Networks

While we touched on the subject of neural networks briefly in Chapters 2 and 7, this is a rapidly evolving area of machine learning, with innovations and new applications being announced on a weekly basis. Recent breakthroughs in machine learning and artificial intelligence, such as the victory of the Alpha Go program against human champions in the game of Go, the constantly improving performance of speech understanding, and the availability of near-instantaneous speech translation, have all been driven by these advances. While the progress in this field is so fast-paced that any current reference to the state of the art will soon be outdated, the recent book *Deep Learning* by Ian Goodfellow, Yoshua Bengio, and Aaron Courville (MIT Press) is a comprehensive introduction into the subject.²

Scaling to Larger Datasets

In this book, we always assumed that the data we were working with could be stored in a NumPy array or SciPy sparse matrix in memory (RAM). Even though modern servers often have hundreds of gigabytes (GB) of RAM, this is a fundamental restriction on the size of data you can work with. Not everybody can afford to buy such a large machine, or even to rent one from a cloud provider. In most applications, the data that is used to build a machine learning system is relatively small, though, and few machine learning datasets consist of hundreds of gigabytes of data or more. This makes expanding your RAM or renting a machine from a cloud provider a viable solution in many cases. If you need to work with terabytes of data, however, or you need

² A preprint of *Deep Learning* can be viewed at <http://www.deeplearningbook.org/>.

to process large amounts of data on a budget, there are two basic strategies: *out-of-core learning* and *parallelization over a cluster*.

Out-of-core learning describes learning from data that cannot be stored in main memory, but where the learning takes place on a single computer (or even a single processor within a computer). The data is read from a source like the hard disk or the network either one sample at a time or in chunks of multiple samples, so that each chunk fits into RAM. This subset of the data is then processed and the model is updated to reflect what was learned from the data. Then, this chunk of the data is discarded and the next bit of data is read. Out-of-core learning is implemented for some of the models in `scikit-learn`, and you can find details on it in the online [user guide](#). Because out-of-core learning requires all of the data to be processed by a single computer, this can lead to long runtimes on very large datasets. Also, not all machine learning algorithms can be implemented in this way.

The other strategy for scaling is distributing the data over multiple machines in a compute cluster, and letting each computer process part of the data. This can be much faster for some models, and the size of the data that can be processed is only limited by the size of the cluster. However, such computations often require relatively complex infrastructure. One of the most popular distributed computing platforms at the moment is the `spark` platform built on top of Hadoop. `spark` includes some machine learning functionality within the `MLlib` package. If your data is already on a Hadoop filesystem, or you are already using `spark` to preprocess your data, this might be the easiest option. If you don't already have such infrastructure in place, establishing and integrating a `spark` cluster might be too large an effort, however. The `vw` package mentioned earlier provides some distributed features and might be a better solution in this case.

Honing Your Skills

As with many things in life, only practice will allow you to become an expert in the topics we covered in this book. Feature extraction, preprocessing, visualization, and model building can vary widely between different tasks and different datasets. Maybe you are lucky enough to already have access to a variety of datasets and tasks. If you don't already have a task in mind, a good place to start is machine learning competitions, in which a dataset with a given task is published, and teams compete in creating the best possible predictions. Many companies, nonprofit organizations, and universities host these competitions. One of the most popular places to find them is [Kaggle](#), a website that regularly holds data science competitions, some of which have substantial prize money attached.

The Kaggle forums are also a good source of information about the latest tools and tricks in machine learning, and a wide range of datasets are available on the site. Even more datasets with associated tasks can be found on the [OpenML platform](#), which

hosts over 20,000 datasets with over 50,000 associated machine learning tasks. Working with these datasets can provide a great opportunity to practice your machine learning skills. A disadvantage of competitions is that they already provide a particular metric to optimize, and usually a fixed, preprocessed dataset. Keep in mind that defining the problem and collecting the data are also important aspects of real-world problems, and that representing the problem in the right way might be much more important than squeezing the last percent of accuracy out of a classifier.

Conclusion

We hope we have convinced you of the usefulness of machine learning in a wide variety of applications, and how easily machine learning can be implemented in practice. Keep digging into the data, and don't lose sight of the larger picture.

Index

A

A/B testing, 359
accuracy, 22, 282
acknowledgments, xi
adjusted rand index (ARI), 191
agglomerative clustering
 evaluating and comparing, 191
 example of, 183
 hierarchical clustering, 184
 linkage choices, 182
 principle of, 182
algorithm chains and pipelines, 305-321
 building pipelines, 308
 building pipelines with make_pipeline,
 313-316
 grid search preprocessing steps, 317
 grid-searching for model selection, 319
 importance of, 305
 overview of, 320
 parameter selection with preprocessing, 306
 pipeline interface, 312
 using pipelines in grid searches, 309-311
algorithm parameter, 118
algorithms (see also models; problem solving)
 evaluating, 28
 minimal code to apply to algorithm, 24
 sample datasets, 30-34
 scaling
 MinMaxScaler, 102, 135-139, 190, 230,
 308, 319
 Normalizer, 134
 RobustScaler, 133
 StandardScaler, 114, 133, 138, 144, 150,
 190-195, 314-320

supervised, classification
 decision trees, 70-83
 gradient boosting, 88-91, 119, 124
 k-nearest neighbors, 35-44
 kernelized support vector machines,
 92-104
 linear SVMs, 56
 logistic regression, 56
 naive Bayes, 68-70
 neural networks, 104-119
 random forests, 84-88
supervised, regression
 decision trees, 70-83
 gradient boosting, 88-91
 k-nearest neighbors, 40
 Lasso, 53-55
 linear regression (OLS), 47, 220-229
 neural networks, 104-119
 random forests, 84-88
 Ridge, 49-55, 67, 112, 231, 234, 310,
 317-319
unsupervised, clustering
 agglomerative clustering, 182-187,
 191-195, 203-207
 DBSCAN, 187-190
 k-means, 168-181
unsupervised, manifold learning
 t-SNE, 163-168
unsupervised, signal decomposition
 non-negative matrix factorization,
 156-163
 principal component analysis, 140-155
alpha parameter in linear models, 50
Anaconda, 6

analysis of variance (ANOVA), 236
area under the curve (AUC), 294-296
attributions, x
average precision, 292

B

bag-of-words representation
 applying to movie reviews, 330-334
 applying to toy dataset, 329
 more than one word (n-grams), 339-344
 steps in computing, 327
BernoulliNB, 68
bigrams, 339
binary classification, 25, 56, 276-296
binning, 144, 220-224
bootstrap samples, 84
Boston Housing dataset, 34
boundary points, 188
Bunch objects, 33
business metric, 275, 358

C

C parameter in SVC, 99
calibration, 288
cancer dataset, 32
categorical features
 categorical data, defined, 324
 defined, 211
 encoded as numbers, 218
 example of, 212
 representation in training and test sets, 217
 representing using one-hot-encoding, 213
categorical variables (see categorical features)
chaining (see algorithm chains and pipelines)
class labels, 25
classification problems
 binary vs. multiclass, 25
 examples of, 26
 goals for, 25
 iris classification example, 14
 k-nearest neighbors, 35
 linear models, 56
 naive Bayes classifiers, 68
 vs. regression problems, 26
classifiers
 DecisionTreeClassifier, 75, 278
 DecisionTreeRegressor, 75, 80
 KNeighborsClassifier, 21-24, 37-43
 KNeighborsRegressor, 42-47

LinearSVC, 56-59, 65, 67, 68
LogisticRegression, 56-62, 67, 209, 253, 279, 315, 332-347
MLPClassifier, 107-119
naive Bayes, 68-70
SVC, 56, 100, 134, 139, 260, 269-272, 273, 305-309, 313-320
uncertainty estimates from, 119-127
cluster centers, 168
clustering algorithms
 agglomerative clustering, 182-187
 applications for, 131
 comparing on faces dataset, 195-207
 DBSCAN, 187-190
 evaluating with ground truth, 191-193
 evaluating without ground truth, 193-195
 goals of, 168
 k-means clustering, 168-181
 summary of, 207
code examples
 downloading, x
 permission for use, x
coef_attribute, 47, 50
comments and questions, xi
competitions, 365
conflation, 344
confusion matrices, 279-286
context, 343
continuous features, 211, 218
core samples/core points, 187
corpus, 325
cos function, 232
CountVectorizer, 334
cross-validation
 analyzing results of, 267-271
 benefits of, 254
 cross-validation splitters, 256
 grid search and, 263-275
 in scikit-learn, 253
 leave-one-out cross-validation, 257
 nested, 272
 parallelizing with grid search, 274
 principle of, 252
 purpose of, 254
 shuffle-split cross-validation, 258
 stratified k-fold, 254-256
 with groups, 259
cross_val_score function, 254, 307

D

data points, defined, 4
data representation, 211-250 (see also feature extraction/feature engineering; text data)
automatic feature selection, 236-241
binning and, 220-224
categorical features, 212-220
effect on model performance, 211
integer features, 218
model complexity vs. dataset size, 29
overview of, 250
table analogy, 4
in training vs. test sets, 217
understanding your data, 4
univariate nonlinear transformations, 232-236

data transformations, 134
(see also preprocessing)

data-driven research, 1

DBSCAN
evaluating and comparing, 191-207
parameters, 189
principle of, 187
returned cluster assignments, 190
strengths and weaknesses, 187

decision boundaries, 37, 56

decision function, 120

decision trees
analyzing, 76
building, 71
controlling complexity of, 74
data representation and, 220-224
feature importance in, 77
if/else structure of, 70
parameters, 82
vs. random forests, 83
strengths and weaknesses, 83

decision_function, 286

deep learning (see neural networks)

dendograms, 184

dense regions, 187

dimensionality reduction, 141, 156

discrete features, 211

discretization, 220-224

distributed computing, 362

document clustering, 347

documents, defined, 325

dual_coef_attribute, 98

E

eigenfaces, 147
embarrassingly parallel, 274
encoding, 328
ensembles
defined, 83
gradient boosted regression trees, 88-92
random forests, 83-88

Enthought Canopy, 6

estimators, 21, 360

estimator_attribute of RFECV, 85

evaluation metrics and scoring
for binary classification, 276-296
for multiclass classification, 296-299
metric selection, 275
model selection and, 300
regression metrics, 299
testing production systems, 359

exp function, 232

expert knowledge, 242-250

F

f(x)=y formula, 18

facial recognition, 147, 157

factor analysis (FA), 163

false positive rate (FPR), 292

false positive/false negative errors, 277

feature extraction/feature engineering, 211-250
(see also data representation; text data)
augmenting data with, 211
automatic feature selection, 236-241
categorical features, 212-220
continuous vs. discrete features, 211
defined, 4, 34, 211
interaction features, 224-232
with non-negative matrix factorization, 156
overview of, 250
polynomial features, 224-232
with principal component analysis, 147
univariate nonlinear transformations, 232-236
using expert knowledge, 242-250

feature importance, 77

features, defined, 4

feature_names attribute, 33

feed-forward neural networks, 104

fit method, 21, 68, 119, 135

fit_transform method, 138

floating-point numbers, 26

folds, 252
forge dataset, 30
frameworks, 362
free string data, 324
freeform text data, 325

G

gamma parameter, 100
Gaussian kernels of SVC, 97, 100
GaussianNB, 68
generalization
 building models for, 26
 defined, 17
 examples of, 27
get_dummies function, 218
get_support method of feature selection, 237
gradient boosted regression trees
 for feature selection, 220-224
 learning_rate parameter, 89
 parameters, 91
 vs. random forests, 88
 strengths and weaknesses, 91
 training set accuracy, 90
graphviz module, 76
grid search
 accessing pipeline attributes, 315
 alternate strategies for, 272
 avoiding overfitting, 261
 model selection with, 319
 nested cross-validation, 272
 parallelizing with cross-validation, 274
 pipeline preprocessing, 317
 searching non-grid spaces, 271
 simple example of, 261
 tuning parameters with, 260
 using pipelines in, 309-311
 with cross-validation, 263-275
GridSearchCV
 best_estimator_ attribute, 267
 best_params_ attribute, 266
 best_score_ attribute, 266

H

handcoded rules, disadvantages of, 1
heat maps, 146
hidden layers, 106
hidden units, 105
hierarchical clustering, 184
high recall, 293

high-dimensional datasets, 32
histograms, 144
hit rate, 283
hold-out sets, 17
human involvement/oversight, 358

I

imbalanced datasets, 277
independent component analysis (ICA), 163
inference, 363
information leakage, 310
information retrieval (IR), 325
integer features, 218
"intelligent" applications, 1
interactions, 34, 224-232
intercept_attribute, 47
iris classification application
 data inspection, 19
 dataset for, 14
 goals for, 13
 k-nearest neighbors, 20
 making predictions, 22
 model evaluation, 22
 multiclass problem, 26
 overview of, 23
 training and testing data, 17
iterative feature selection, 240

J

Jupyter Notebook, 7

K

k-fold cross-validation, 252
k-means clustering
 applying with scikit-learn, 170
 vs. classification, 171
 cluster centers, 169
 complex datasets, 179
 evaluating and comparing, 191
 example of, 168
 failures of, 173
 strengths and weaknesses, 181
 vector quantization with, 176
k-nearest neighbors (k-NN)
 analyzing KNeighborsClassifier, 37
 analyzing KNeighborsRegressor, 43
 building, 20
 classification, 35-37

vs. linear models, 46
parameters, 44
predictions with, 35
regression, 40
strengths and weaknesses, 44
Kaggle, 365
kernelized support vector machines (SVMs)
 kernel trick, 97
 linear models and nonlinear features, 92
 vs. linear support vector machines, 92
 mathematics of, 92
 parameters, 104
 predictions with, 98
 preprocessing data for, 102
 strengths and weaknesses, 104
 tuning SVM parameters, 99
 understanding, 98
knn object, 21

L
L1 regularization, 53
L2 regularization, 49, 60, 67
Lasso model, 53
Latent Dirichlet Allocation (LDA), 348-355
leafs, 71
leakage, 310
learn from the past approach, 243
learning_rate parameter, 89
leave-one-out cross-validation, 257
lemmatization, 344-347
linear functions, 56
linear models
 classification, 56
 data representation and, 220-224
 vs. k-nearest neighbors, 46
 Lasso, 53
 linear SVMs, 56
 logistic regression, 56
 multiclass classification, 63
 ordinary least squares, 47
 parameters, 67
 predictions with, 45
 regression, 45
 ridge regression, 49
 strengths and weaknesses, 67
linear regression, 47, 224-232
linear support vector machines (SVMs), 56
linkage arrays, 185
live testing, 359

log function, 232
loss functions, 56
low-dimensional datasets, 32

M

machine learning
 algorithm chains and pipelines, 305-321
 applications for, 1-5
 approach to problem solving, 357-366
 benefits of Python for, 5
 building your own systems, vii
 data representation, 211-250
 examples of, 1, 13-23
 mathematics of, vii
 model evaluation and improvement,
 251-303
 preprocessing and scaling, 132-140
 prerequisites to learning, vii
 resources, ix, 361-366
 scikit-learn and, 5-13
 supervised learning, 25-129
 understanding your data, 4
 unsupervised learning, 131-209
 working with text data, 323-356
make_pipeline function
 accessing step attributes, 314
 displaying steps attribute, 314
 grid-searched pipelines and, 315
 syntax for, 313
manifold learning algorithms
 applications for, 164
 example of, 164
 results of, 168
 visualizations with, 163
mathematical functions for feature transformations, 232
matplotlib, 9
max_features parameter, 84
meta-estimators for trees and forests, 266
method chaining, 68
metrics (see evaluation metrics and scoring)
mglearn, 11
mllib, 362
model-based feature selection, 238
models (see also algorithms)
 calibrated, 288
 capable of generalization, 26
 coefficients with text data, 338-347
 complexity vs. dataset size, 29

cross-validation of, 252-260
effect of data representation choices on, 211
evaluation and improvement, 251-252
evaluation metrics and scoring, 275-302
iris classification application, 13-23
overfitting vs. underfitting, 28
pipeline preprocessing and, 317
selecting, 300
selecting with grid search, 319
theory behind, 361
tuning parameters with grid search, 260-275
movie reviews, 325
multiclass classification
 vs. binary classification, 25
 evaluation metrics and scoring for, 296-299
 linear models for, 63
 uncertainty estimates, 124
multilayer perceptrons (MLPs), 104
MultinomialNB, 68

N

n-grams, 339
naive Bayes classifiers
 kinds in scikit-learn, 68
 parameters, 70
 strengths and weaknesses, 70
natural language processing (NLP), 325, 355
negative class, 26
nested cross-validation, 272
Netflix prize challenge, 363
neural networks (deep learning)
 accuracy of, 114
 estimating complexity in, 118
 predictions with, 104
 randomization in, 113
 recent breakthroughs in, 364
 strengths and weaknesses, 117
 tuning, 108
non-negative matrix factorization (NMF)
 applications for, 156
 applying to face images, 157
 applying to synthetic data, 156
normalization, 344
normalized mutual information (NMI), 191
NumPy (Numeric Python) library, 7

O

offline evaluation, 359
one-hot-encoding, 213-217

one-out-of-N encoding, 213-217
one-vs.-rest approach, 63
online resources, ix
online testing, 359
OpenML platform, 365
operating points, 289
ordinary least squares (OLS), 47
out-of-core learning, 364
outlier detection, 197
overfitting, 28, 261

P

pair plots, 19
pandas
 benefits of, 10
 checking string-encoded data, 214
 column indexing in, 216
 converting data to one-hot-encoding, 214
 get_dummies function, 218
parallelization over a cluster, 364
permissions, x
pipelines (see algorithm chains and pipelines)
polynomial features, 224-232
polynomial kernels, 97
polynomial regression, 228
positive class, 26
POSIX time, 244
pre- and post-pruning, 74
precision, 282, 358
precision-recall curves, 289-292
predict for the future approach, 243
predict method, 22, 37, 68, 267
predict_proba function, 122, 286
preprocessing, 132-140
 data transformation application, 134
 effect on supervised learning, 138
 kinds of, 133
 parameter selection with, 306
 pipelines and, 317
 purpose of, 132
 scaling training and test data, 136
principal component analysis (PCA)
 drawbacks of, 146
 example of, 140
 feature extraction with, 147
 unsupervised nature of, 145
 visualizations with, 142
 whitening option, 150
probabilistic modeling, 363

probabilistic programming, 363
problem solving
 building your own estimators, 360
 business metrics and, 358
 initial approach to, 357
 resources, 361-366
 simple vs. complicated cases, 358
 steps of, 358
 testing your system, 359
 tool choice, 359
production systems
 testing, 359
 tool choice, 359
pruning for decision trees, 74
pseudorandom number generators, 18
pure leafs, 73
PyMC language, 364
Python
 benefits of, 5
 prepackaged distributions, 6
 Python 2 vs. Python 3, 12
 Python(x,y), 6
 statsmodel package, 362

R

R language, 362
radial basis function (RBF) kernel, 97
random forests
 analyzing, 85
 building, 84
 data representation and, 220-224
 vs. decision trees, 83
 vs. gradient boosted regression trees, 88
 parameters, 88
 predictions with, 84
 randomization in, 83
 strengths and weaknesses, 87
random_state parameter, 18
ranking, 363
real numbers, 26
recall, 282
receiver operating characteristics (ROC)
 curves, 292-296
recommender systems, 363
rectified linear unit (relu), 106
rectifying nonlinearity, 106
recurrent neural networks (RNNs), 356
recursive feature elimination (RFE), 240
regression

f_regression, 236, 310
LinearRegression, 47-56, 81, 247
regression problems
 Boston Housing dataset, 34
 vs. classification problems, 26
 evaluation metrics and scoring, 299
 examples of, 26
 goals for, 26
 k-nearest neighbors, 40
 Lasso, 53
 linear models, 45
 ridge regression, 49
 wave dataset illustration, 31
regularization
 L1 regularization, 53
 L2 regularization, 49, 60
rescaling
 example of, 132-140
 kernel SVMs, 102
resources, ix
ridge regression, 49
robustness-based clustering, 194
roots, 72

S

Safari Books Online, x
samples, defined, 4
scaling, 132-140
 data transformation application, 134
 effect on supervised learning, 138
 into larger datasets, 364
 kinds of, 133
 purpose of, 132
 training and test data, 136
scatter plots, 19
scikit-learn
 alternate frameworks, 362
 benefits of, 5
 Bunch objects, 33
 cancer dataset, 32
 core code for, 24
 data and labels in, 18
 documentation, 6
 feature_names attribute, 33
 fit method, 21, 68, 119, 135
 fit_transform method, 138
 installing, 6
 knn object, 21
 libraries and tools, 7-11

predict method, 22, 37, 68
Python 2 vs. Python 3, 12
random_state parameter, 18
scaling mechanisms in, 139
score method, 23, 37, 43
transform method, 135
user guide, 6
versions used, 12

scikit-learn classes and functions

- accuracy_score, 193
- adjusted_rand_score, 191
- AgglomerativeClustering, 182, 191, 203-207
- average_precision_score, 292
- BaseEstimator, 360
- classification_report, 284-288, 298
- confusion_matrix, 279-299
- CountVectorizer, 329-355
- cross_val_score, 253, 256, 300, 307, 360
- DBSCAN, 187-190
- DecisionTreeClassifier, 75, 278
- DecisionTreeRegressor, 75, 80
- DummyClassifier, 278
- ElasticNet class, 55
- ENGLISH_STOP_WORDS, 334
- Estimator, 21
- export_graphviz, 76
- f1_score, 284, 291
- fetch_lfw_people, 147
- f_regression, 236, 310
- GradientBoostingClassifier, 88-91, 119, 124
- GridSearchCV, 263-275, 300-301, 305-309, 315-320, 360
- GroupKFold, 259
- KFold, 256, 260
- KMeans, 174-181
- KNeighborsClassifier, 21-24, 37-43
- KNeighborsRegressor, 42-47
- Lasso, 53-55
- LatentDirichletAllocation, 348
- LeaveOneOut, 257
- LinearRegression, 47-56, 81, 247
- LinearSVC, 56-59, 65, 67, 68
- load_boston, 34, 230, 317
- load_breast_cancer, 32, 38, 59, 75, 134, 144, 236, 305
- load_digits, 164, 278
- load_files, 326
- load_iris, 14, 124, 253

LogisticRegression, 56-62, 67, 209, 253, 279, 315, 332-347

make_blobs, 92, 119, 136, 173-183, 188, 286

make_circles, 119

make_moons, 85, 108, 175, 190-195

make_pipeline, 313-319

MinMaxScaler, 102, 133, 135-139, 190, 230, 308, 309, 319

MLPClassifier, 107-119

NMF, 140, 159-163, 179-182, 348

Normalizer, 134

OneHotEncoder, 218, 247

ParameterGrid, 274

PCA, 140-166, 179, 195-206, 313-314, 348

Pipeline, 305-319, 320

PolynomialFeatures, 227-230, 248, 317

precision_recall_curve, 289-292

RandomForestClassifier, 84-86, 238, 290, 319

RandomForestRegressor, 84, 231, 240

RFE, 240-241

Ridge, 49, 67, 112, 231, 234, 310, 317-319

RobustScaler, 133

roc_auc_score, 294-301

roc_curve, 293-296

SCORERS, 301

SelectFromModel, 238

SelectPercentile, 236, 310

ShuffleSplit, 258, 258

silhouette_score, 193

StandardScaler, 114, 133, 138, 144, 150, 190-195, 314-320

StratifiedKFold, 260, 274

StratifiedShuffleSplit, 258, 347

SVC, 56, 100, 134, 139, 260-267, 269-272, 305-309, 313-320

SVR, 92, 229

TfidfVectorizer, 336-356

train_test_split, 17-19, 251, 286, 289

TransformerMixin, 360

TSNE, 166

SciPy, 8

score method, 23, 37, 43, 267, 308

sensitivity, 283

sentiment analysis example, 325

shapes, defined, 16

shuffle-split cross-validation, 258

sin function, 232

soft voting strategy, 84

spark computing environment, 362
sparse coding (dictionary learning), 163
sparse datasets, 44
splits, 252
Stan language, 364
statsmodel package, 362
stemming, 344-347
stopwords, 334
stratified k-fold cross-validation, 254-256
string-encoded categorical data, 214
supervised learning, 25-129 (see also classification problems; regression problems)
algorithms for
 decision trees, 70-83
 ensembles of decision trees, 83-92
 k-nearest neighbors, 35-44
 kernelized support vector machines, 92-104
 linear models, 45-68
 naive Bayes classifiers, 68
 neural networks (deep learning), 104-119
 overview of, 2
data representation, 4
examples of, 3
generalization, 26
goals for, 25
model complexity vs. dataset size, 29
overfitting vs. underfitting, 28
overview of, 127
sample datasets, 30-34
uncertainty estimates, 119-127
support vectors, 98
synthetic datasets, 30

T

t-SNE algorithm (see manifold learning algorithms)
tangens hyperbolicus (tanh), 106
term frequency-inverse document frequency (tf-idf), 336-347
terminal nodes, 71
test data/test sets
 Boston Housing dataset, 34
 defined, 17
 forge dataset, 30
 wave dataset, 31
 Wisconsin Breast Cancer dataset, 32
text data, 323-356

bag-of-words representation, 327-334
examples of, 323
model coefficients, 338
overview of, 355
rescaling data with tf-idf, 336-338
sentiment analysis example, 325
stopwords, 334
topic modeling and document clustering, 347-355
types of, 323-325
time series predictions, 363
tokenization, 328, 344-347
top nodes, 72
topic modeling, with LDA, 347-355
training data, 17
train_test_split function, 254
transform method, 135, 312, 334
transformations
 selecting, 235
 univariate nonlinear, 232-236
 unsupervised, 131
tree module, 76
trigrams, 339
true positive rate (TPR), 283, 292
true positives/true negatives, 281
typographical conventions, ix

U

uncertainty estimates
 applications for, 119
 decision function, 120
 in binary classification evaluation, 286-288
 multiclass classification, 124
 predicting probabilities, 122
underfitting, 28
unigrams, 340
univariate nonlinear transformations, 232-236
univariate statistics, 236
unsupervised learning, 131-209
algorithms for
 agglomerative clustering, 182-187
 clustering, 168-207
 DBSCAN, 187-190
 k-means clustering, 168-181
 manifold learning with t-SNE, 163-168
 non-negative matrix factorization, 156-163
 overview of, 3
 principal component analysis, 140-155

challenges of, 132
data representation, 4
examples of, 3
overview of, 208
scaling and preprocessing for, 132-140
types of, 131
unsupervised transformations, 131

V

value_counts function, 214
vector quantization, 176
vocabulary building, 328
voting, 36
vowpal wabbit, 362

W

wave dataset, 31
weak learners, 88
weights, 47, 106
whitening option, 150
Wisconsin Breast Cancer dataset, 32
word stems, 344

X

xgboost package, 91
xkcd Color Survey, 324

About the Authors

Andreas Müller received his PhD in machine learning from the University of Bonn. After working as a machine learning researcher on computer vision applications at Amazon for a year, he joined the Center for Data Science at New York University. For the last four years, he has been a maintainer of and one of the core contributors to `scikit-learn`, a machine learning toolkit widely used in industry and academia, and has authored and contributed to several other widely used machine learning packages. His mission is to create open tools to lower the barrier of entry for machine learning applications, promote reproducible science, and democratize the access to high-quality machine learning algorithms.

Sarah Guido is a data scientist who has spent a lot of time working in start-ups. She loves Python, machine learning, large quantities of data, and the tech world. An accomplished conference speaker, Sarah attended the University of Michigan for grad school and currently resides in New York City.

Colophon

The animal on the cover of *Introduction to Machine Learning with Python* is a hellbender salamander (*Cryptobranchus alleganiensis*), an amphibian native to the eastern United States (ranging from New York to Georgia). It has many colorful nicknames, including “Allegheny alligator,” “snot otter,” and “mud-devil.” The origin of the name “hellbender” is unclear: one theory is that early settlers found the salamander’s appearance unsettling and supposed it to be a demonic creature trying to return to hell.

The hellbender salamander is a member of the giant salamander family, and can grow as large as 29 inches long. This is the third-largest aquatic salamander species in the world. Their bodies are rather flat, with thick folds of skin along their sides. While they do have a single gill on each side of the neck, hellbenders largely rely on their skin folds to breathe: gas flows in and out through capillaries near the surface of the skin.

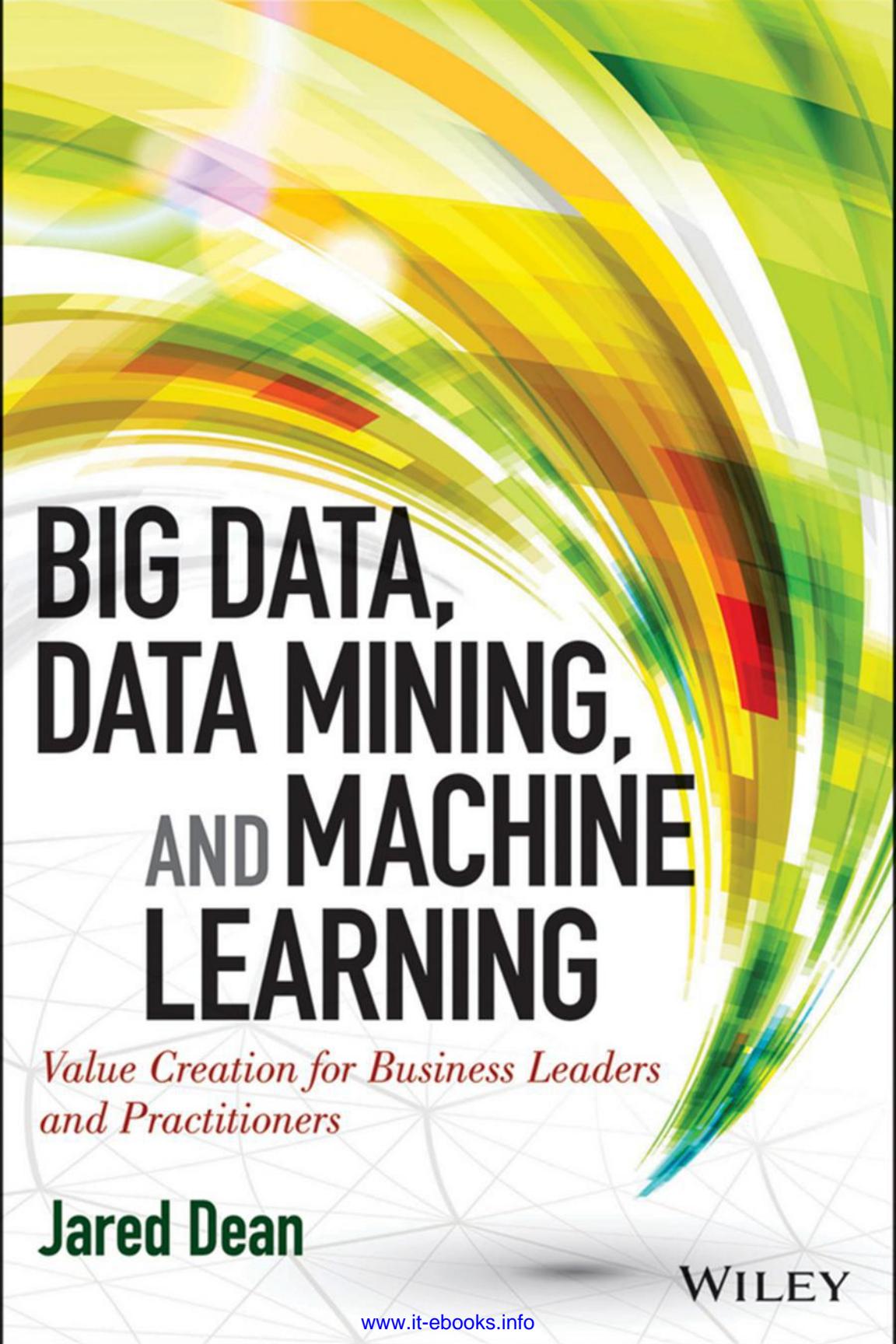
Because of this, their ideal habitat is in clear, fast-moving, shallow streams, which provide plenty of oxygen. The hellbender shelters under rocks and hunts primarily by sense of smell, though it is also able to detect vibrations in the water. Its diet is made up of crayfish, small fish, and occasionally the eggs of its own species. The hellbender is also a key member of its ecosystem as prey: predators include various fish, snakes, and turtles.

Hellbender salamander populations have decreased significantly in the last few decades. Water quality is the largest issue, as their respiratory system makes them very sensitive to polluted or murky water. An increase in agriculture and other human

activity near their habitat means greater amounts of sediment and chemicals in the water. In an effort to save this endangered species, biologists have begun to raise the amphibians in captivity and release them when they reach a less vulnerable age.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from *Wood's Animate Creation*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.



BIG DATA, DATA MINING, AND MACHINE LEARNING

*Value Creation for Business Leaders
and Practitioners*

Jared Dean

WILEY

Additional praise for *Big Data, Data Mining, and Machine Learning: Value Creation for Business Leaders and Practitioners*

“Jared’s book is a great introduction to the area of High Powered Analytics. It will be useful for those who have experience in predictive analytics but who need to become more versed in how technology is changing the capabilities of existing methods and creating new possibilities. It will also be helpful for business executives and IT professionals who’ll need to make the case for building the environments for, and reaping the benefits of, the next generation of advanced analytics.”

—Jonathan Levine, Senior Director, Consumer Insight Analysis at Marriott International

“The ideas that Jared describes are the same ideas that being used by our Kaggle contest winners. This book is a great overview for those who want to learn more and gain a complete understanding of the many facets of data mining, knowledge discovery and extracting value from data.”

—Anthony Goldbloom Founder and CEO of Kaggle

“The concepts that Jared presents in this book are extremely valuable for the students that I teach and will help them to more fully understand the power that can be unlocked when an organization begins to take advantage of its data. The examples and case studies are particularly useful for helping students to get a vision for what is possible. Jared’s passion for analytics comes through in his writing, and he has done a great job of making complicated ideas approachable to multiple audiences.”

—Tonya Etchison Balan, Ph.D., Professor of Practice, Statistics, Poole College of Management, North Carolina State University

Big Data, Data Mining, and Machine Learning

Wiley & SAS Business Series

The Wiley & SAS Business Series presents books that help senior-level managers with their critical management decisions.

Titles in the Wiley & SAS Business Series include:

Activity-Based Management for Financial Institutions: Driving Bottom-Line Results by Brent Bahnhub

Analytics in a Big Data World: The Essential Guide to Data Science and its Applications by Bart Baesens

Bank Fraud: Using Technology to Combat Losses by Revathi Subramanian

Big Data Analytics: Turning Big Data into Big Money by Frank Ohlhorst

Branded! How Retailers Engage Consumers with Social Media and Mobility by Bernie Brennan and Lori Schafer

Business Analytics for Customer Intelligence by Gert Laursen

Business Analytics for Managers: Taking Business Intelligence beyond Reporting by Gert Laursen and Jesper Thorlund

The Business Forecasting Deal: Exposing Bad Practices and Providing Practical Solutions by Michael Gilliland

Business Intelligence Applied: Implementing an Effective Information and Communications Technology Infrastructure by Michael Gendron

Business Intelligence and the Cloud: Strategic Implementation Guide by Michael S. Gendron

Business Intelligence Success Factors: Tools for Aligning Your Business in the Global Economy by Olivia Parr Rud

Business Transformation: A Roadmap for Maximizing Organizational Insights by Aiman Zeid

CIO Best Practices: Enabling Strategic Value with Information Technology, second edition by Joe Stenzel

Connecting Organizational Silos: Taking Knowledge Flow Management to the Next Level with Social Media by Frank Leistner

Credit Risk Assessment: The New Lending System for Borrowers, Lenders, and Investors by Clark Abrahams and Mingyuan Zhang

Credit Risk Scorecards: Developing and Implementing Intelligent Credit Scoring by Naeem Siddiqi

The Data Asset: How Smart Companies Govern Their Data for Business Success by Tony Fisher

Delivering Business Analytics: Practical Guidelines for Best Practice by Evan Stubbs

Demand-Driven Forecasting: A Structured Approach to Forecasting, second edition by Charles Chase

Demand-Driven Inventory Optimization and Replenishment: Creating a More Efficient Supply Chain by Robert A. Davis

Developing Human Capital: Using Analytics to Plan and Optimize Your Learning and Development Investments by Gene Pease, Barbara Beresford, and Lew Walker

The Executive's Guide to Enterprise Social Media Strategy: How Social Networks Are Radically Transforming Your Business by David Thomas and Mike Barlow

Economic and Business Forecasting: Analyzing and Interpreting Econometric Results by John Silvia, Azhar Iqbal, Kaylyn Swankoski, Sarah Watt, and Sam Bullard

Executive's Guide to Solvency II by David Buckham, Jason Wahl, and Stuart Rose

Fair Lending Compliance: Intelligence and Implications for Credit Risk Management by Clark R. Abrahams and Mingyuan Zhang

Foreign Currency Financial Reporting from Euros to Yen to Yuan: A Guide to Fundamental Concepts and Practical Applications by Robert Rowan

Harness Oil and Gas Big Data with Analytics: Optimize Exploration and Production with Data Driven Models by Keith Holdaway

Health Analytics: Gaining the Insights to Transform Health Care by Jason Burke

Heuristics in Analytics: A Practical Perspective of What Influences Our Analytical World by Carlos Andre Reis Pinheiro and Fiona McNeill

Human Capital Analytics: How to Harness the Potential of Your Organization's Greatest Asset by Gene Pease, Boyce Byerly, and Jac Fitz-enz

Implement, Improve and Expand Your Statewide Longitudinal Data System: Creating a Culture of Data in Education by Jamie McQuiggan and Armistead Sapp

Information Revolution: Using the Information Evolution Model to Grow Your Business by Jim Davis, Gloria J. Miller, and Allan Russell

Killer Analytics: Top 20 Metrics Missing from your Balance Sheet by Mark Brown

Manufacturing Best Practices: Optimizing Productivity and Product Quality by Bobby Hull

Marketing Automation: Practical Steps to More Effective Direct Marketing by Jeff LeSueur

Mastering Organizational Knowledge Flow: How to Make Knowledge Sharing Work by Frank Leistner

The New Know: Innovation Powered by Analytics by Thornton May

Performance Management: Integrating Strategy Execution, Methodologies, Risk, and Analytics by Gary Cokins

Predictive Business Analytics: Forward-Looking Capabilities to Improve Business Performance by Lawrence Maisel and Gary Cokins

Retail Analytics: The Secret Weapon by Emmett Cox

Social Network Analysis in Telecommunications by Carlos Andre Reis Pinheiro

Statistical Thinking: Improving Business Performance, second edition, by Roger W. Hoerl and Ronald D. Snee

Taming the Big Data Tidal Wave: Finding Opportunities in Huge Data Streams with Advanced Analytics by Bill Franks

Too Big to Ignore: The Business Case for Big Data by Phil Simon

The Value of Business Analytics: Identifying the Path to Profitability by Evan Stubbs

The Visual Organization: Data Visualization, Big Data, and the Quest for Better Decisions by Phil Simon

Visual Six Sigma: Making Data Analysis Lean by Ian Cox, Marie A. Gaudard, Philip J. Ramsey, Mia L. Stephens, and Leo Wright

Win with Advanced Business Analytics: Creating Business Value from Your Data by Jean Paul Isson and Jesse Harriott

For more information on any of the above titles, please visit www.wiley.com.

Big Data, Data Mining, and Machine Learning

*Value Creation for Business Leaders
and Practitioners*

Jared Dean

WILEY

Cover Design: Wiley
Cover Image: © iStockphoto / elly99

Copyright © 2014 by SAS Institute Inc. All rights reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600, or on the Web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at <http://booksupport.wiley.com>. For more information about Wiley products, visit www.wiley.com.

Library of Congress Cataloging-in-Publication Data:

Dean, Jared, 1978-

Big data, data mining, and machine learning : value creation for business leaders and practitioners / Jared Dean.

1 online resource.—(Wiley & SAS business series)

Includes index.

ISBN 978-1-118-92069-5 (ebk); ISBN 978-1-118-92070-1 (ebk);

ISBN 978-1-118-61804-2 (hardback) 1. Management—Data processing.

2. Data mining. 3. Big data. 4. Database management. 5. Information technology—Management. I. Title.

HD30.2

658'.05631—dc23

2014009116

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

*To my wife, without whose help, love, and devotion,
this book would not exist. Thank you, Katie!*

*For Geoffrey, Ava, Mason, and Chase: Remember that the
quickest path to easy is through hard.*

Contents



Forward xiii

Preface xv

Acknowledgments xix

Introduction 1

 Big Data Timeline 5

 Why This Topic Is Relevant Now 8

 Is Big Data a Fad? 9

 Where Using Big Data Makes a Big Difference 12

Part One The Computing Environment 23

Chapter 1 Hardware 27

 Storage (Disk) 27

 Central Processing Unit 29

 Memory 31

 Network 33

Chapter 2 Distributed Systems 35

 Database Computing 36

 File System Computing 37

 Considerations 39

Chapter 3 Analytical Tools 43

 Weka 43

 Java and JVM Languages 44

 R 47

 Python 49

 SAS 50

Part Two Turning Data into Business Value 53

Chapter 4 Predictive Modeling 55

- A Methodology for Building Models 58
- sEMMA 61
- Binary Classification 64
- Multilevel Classification 66
- Interval Prediction 66
- Assessment of Predictive Models 67

Chapter 5 Common Predictive Modeling Techniques 71

- RFM 72
- Regression 75
- Generalized Linear Models 84
- Neural Networks 90
- Decision and Regression Trees 101
- Support Vector Machines 107
- Bayesian Methods Network Classification 113
- Ensemble Methods 124

Chapter 6 Segmentation 127

- Cluster Analysis 132
- Distance Measures (Metrics) 133
- Evaluating Clustering 134
- Number of Clusters 135
- K-means Algorithm 137
- Hierarchical Clustering 138
- Profiling Clusters 138

Chapter 7 Incremental Response Modeling 141

- Building the Response Model 142
- Measuring the Incremental Response 143

Chapter 8 Time Series Data Mining 149

- Reducing Dimensionality 150
- Detecting Patterns 151
- Time Series Data Mining in Action: Nike+ FuelBand 154

Chapter 9 Recommendation Systems 163

- What Are Recommendation Systems? 163
- Where Are They Used? 164

How Do They Work?	165
Assessing Recommendation Quality	170
Recommendations in Action: SAS Library	171
Chapter 10 Text Analytics	175
Information Retrieval	176
Content Categorization	177
Text Mining	178
Text Analytics in Action: Let's Play <i>Jeopardy!</i>	180
Part Three Success Stories of Putting It All Together	193
Chapter 11 Case Study of a Large U.S.-Based Financial Services Company	197
Traditional Marketing Campaign Process	198
High-Performance Marketing Solution	202
Value Proposition for Change	203
Chapter 12 Case Study of a Major Health Care Provider	205
CAHPS	207
HEDIS	207
HOS	208
IRE	208
Chapter 13 Case Study of a Technology Manufacturer	215
Finding Defective Devices	215
How They Reduced Cost	216
Chapter 14 Case Study of Online Brand Management	221
Chapter 15 Case Study of Mobile Application Recommendations	225
Chapter 16 Case Study of a High-Tech Product Manufacturer	229
Handling the Missing Data	230
Application beyond Manufacturing	231
Chapter 17 Looking to the Future	233
Reproducible Research	234
Privacy with Public Data Sets	234
The Internet of Things	236

Software Development in the Future 237
Future Development of Algorithms 238
In Conclusion 241

About the Author 243

Appendix 245

References 247

Index 253

Foreword

I love the field of predictive analytics and have lived in this world for my entire career. The mathematics are fun (at least for me), but turning what the algorithms uncover into solutions that a company uses and generates profit from makes the mathematics worthwhile. In some ways, Jared Dean and I are unusual in this regard; we really do love seeing these solutions work for organizations we work with. What amazes us, though, is that this field that we used to do in the back office, a niche of a niche, has now become one of the sexiest jobs of the twenty-first century. How did this happen?

We live in a world where data is collected in ever-increasing amounts, summarizing more of what people and machines do, and capturing finer granularity of their behavior. These three ways to characterize data are sometimes described as volume, variety, and velocity—the definition of big data. They are collected because of the perceived value in the data even if we don’t know exactly what we will do with it. Initially, many organizations collect it and report summaries, often using approaches from business intelligence that have become commonplace.

But in recent years, a paradigm shift has taken place. Organizations have found that predictive analytics transforms the way they make decisions. The algorithms and approaches to predictive modeling described in this book are not new for the most part; Jared himself describes the big-data problem as nothing new. The algorithms he describes are all at least 15 years old, a testimony to their effectiveness that fundamentally new algorithms are not needed. Nevertheless, predictive modeling is in fact new to many organizations as they try to improve decisions with data. These organizations need to gain an understanding not only of the science and principles of predictive modeling but how to apply the principles to problems that defy the standard approaches and answers.

But there is much more to predictive modeling than just building predictive models. The operational aspects of predictive modeling

projects are often overlooked and are rarely covered in books and courses. First, this includes specifying hardware and software needed for a predictive modeling. As Jared describes, this depends on the organization, the data, and the analysts working on the project. Without setting up analysts with the proper resources, projects flounder and often fail. I've personally witnessed this on projects I have worked on, where hardware was improperly specified causing me to spend a considerable amount of time working around the limitations in RAM and processing speed.

Ultimately, the success of predictive modeling projects is measured by the metric that matters to the organization using it, whether it be increased efficiency, ROI, customer lifetime value, or soft metrics like company reputation. I love the case studies in this book that address these issues, and you have a half-dozen here to whet your appetite. This is especially important for managers who are trying to understand how predictive modeling will impact their bottom line.

Predictive modeling is science, but successful implementation of predictive modeling solutions requires connecting the models to the business. Experience is essential to recognize these connections, and there is a wealth of experience here to draw from to propel you in your predictive modeling journey.

Dean Abbott
Abbott Analytics, Inc.
March 2014

Preface

This book project was first presented to me during my first week in my current role of managing the data mining development at SAS. Writing a book has always been a bucket-list item, and I was very excited to be involved. I've come to realize why so many people want to write books, but why so few get the chance to see their thoughts and ideas bound and published.

I've had the opportunity during my studies and professional career to be front and center to some great developments in the area of data mining and to study under some brilliant minds. This experience helped position me with the skills and experience I needed to create this work.

Data mining is a field I love. Ever since childhood, I've wanted to explain how things work and understand how systems function both in the "average" case but also at the extremes. From elementary school through high school, I thought engineering would be the job that would couple both my curiosity and my desire to explain the world around me. However, before my last year as an undergraduate student, I found statistics and information systems, and I was hooked.

In Part One of the book, I explore the foundations of hardware and system architecture. This is a love that my parents were kind enough to indulge me in, in a day when computers cost much much more than \$299. The first computer in my home was an Apple IIc, with two 5.25" floppy disk drives and no hard drive. A few years later I built an Intel 386 PC from a kit, and I vividly remember playing computer games and hitting the turbo button to move the CPU clock speed from 8 MHz to 16 MHz. I've seen Moore's Law firsthand, and it still amazes me that my smartphone holds more computing power than the computers used in the Mercury space program, the Apollo space program, and the Orbiter space shuttle program combined.

After I finished my undergraduate degree in statistics, I began to work for the federal government at the U.S. Bureau of the Census. This is where I got my first exposure to big data. Prior to joining the Census

Bureau, I had never written a computer program that took more than a minute to run (unless the point was to make the program run for more than a minute). One of my first projects was working with the Master Address File (MAF),¹ which is an address list maintained by the Census Bureau. This address list is also the primary survey frame for current surveys that the Census Bureau administers (yes, there is lots of work to do the other nine years). The list has more than 300 million records, and combining all the address information, longitudinal information, and geographic information, there are hundreds of attributes associated with each housing unit. Working with such a large data set was where I first learned about programming efficiency, scalability, and hardware optimization. I'm grateful to my patient manager, Maryann, who gave me the time to learn and provided me with interesting, valuable projects that gave me practical experience and the opportunity to innovate. It was a great position because I got to try new techniques and approaches that had not been studied before in that department. As with any new project, some ideas worked great and others failed. One specific project I was involved in was trying to identify which blocks (the Census Bureau has the United States divided up into unique geographic areas—the hierarchy is state, county, tract, block group, and block; there are about 8.2 million blocks in the United States) from Census 2000 had been overcounted or undercounted. Through the available data, we did not have a way to verify that our model for predicting the deviation of actual housing unit count from reported housing unit count was accurate. The program was fortunate to have funding from congress to conduct field studies to provide feedback and validation of the models. This was the first time I had heard the term "data mining" and I was first exposed to SAS™ Enterprise Miner® and CART® by Salford Systems. After a period of time working for the Census Bureau, I realized that I needed more education to achieve my career goals, and so I enrolled in the statistics department at George Mason University in Fairfax, VA.

During graduate school, I learned in more detail about the algorithms common to the fields of data mining, machine learning, and statistics; these included survival analysis, survey sampling, and

¹ The MAF is created during decennial census operations for every housing unit, or potential housing unit, in the United States.

computational statistics. Through my graduate studies, I was able to merge the lessons taught in the classroom to the practical data analysis and innovations required in the office. I acquired an understanding of the theory and the relative strengths and weaknesses of different approaches for data analysis and predictive analytics.

After graduate school, I changed direction in my career, moving from a data analysis² role and becoming a software developer. I went to work for SAS Institute Inc., where I was participating in the creation of the software that I had previously used. I had moved from using the software to building it. This presented new challenges and opportunities for growth as I learned about the rigorous numerical validation that SAS imposes on the software, along with its thorough documentation and tireless effort to make new software enhancements consistent with existing software and to consistently deliver new software features that customers need.

During my years at SAS, I've come to thoroughly understand how the software is made and how our customers use it. I often get the chance to visit with customers, listen to their business challenges, and recommend methods or process that help lead them to success; creating value for their organizations.

It is from this collection of experience that I wrote this book, along with the help of the wonderful staff and my colleagues both inside and outside of SAS Institute.

²I was a data scientist before the term was invented

Acknowledgments

I would like to thank all those who helped me to make this book a reality. It was a long journey and a wonderful learning and growing experience.

Patrick Hall, thank you for your validation of my ideas and contributing many of your own. I appreciate that I could discuss ideas and trends with you and get thoughtful, timely, and useful feedback.

Joseph Pingot, Ilknur Kabul, Jorge Silva, Larry Lewis, Susan Haller, and Wendy Czika, thank you for sharing your domain knowledge and passion for analytics.

Michael Wallis, thank you for your help in the text analytics area and developing the *Jeopardy!* example.

Udo Sglavo and Taiyeong Lee, thank you for reviewing and offering significant contributions in the analysis of times series data mining.

Barbara Walters and Vicki Jones, thank you for all the conversations about reads and feeds in understanding how the hardware impacted the software.

Jared Peterson for his help in downloading the data from my Nike+ FuelBand.

Franklin So, thank you for your excellent description of a customer's core business problem.

Thank you Grandma Catherine Coyne, who sacrificed many hours to help a fellow author in editing the manuscript to greatly improve its readability. I am very grateful for your help and hope that when I am 80-something I can be half as active as you are.

I would also like to thank the staff of SAS Press and John Wiley & Sons for the feedback and support through all phases of this project, including some major detours along the way.

Finally, I need to acknowledge my wife, Katie, for shouldering many burdens as I researched, wrote, edited, and wrote more. Meeting you was the best thing that has happened to me in my whole life.

Introduction

Hiding within those mounds of data is knowledge that could change the life of a patient, or change the world.

—Atul Butte, Stanford University

“Cancer” is the term given for a class of diseases in which abnormal cells divide in an uncontrolled fashion and invade body tissues. There are more than 100 unique types of cancer. Most are named after the location (usually an organ) where they begin. Cancer begins in the cells of the body. Under normal circumstances, the human body controls the production of new cells to replace cells that are old or have become damaged. Cancer is not normal. In patients with cancer, cells do not die when they are supposed to and new cells form when they are not needed (like when I ask my kids to use the copy machine and I get back ten copies instead of the one I asked for). The extra cells may form a mass of tissue; this is referred to as a tumor. Tumors come in two varieties: benign tumors, which are not cancerous, and malignant tumors, which are cancerous. Malignant tumors spread through the body and invade the tissue. My family, like most I know, has lost a family member to the disease. There were an estimated 1.6 million new cases of cancer in the United States in 2013 and more than 580,000 deaths as a result of the disease.

An estimated 235,000 people in the United States were diagnosed with breast cancer in 2014, and about 40,000 people will die in 2014 as a result of the disease. The most common type of

breast cancer is ductal carcinoma, which begins in the lining of the milk ducts. The next most common type of breast cancer is lobular carcinoma. There are a number of treatment options for breast cancer including surgery, chemotherapy, radiation therapy, immunotherapy, and vaccine therapy. Often one or more of the treatment options is used to help ensure the best outcome for patients. About 60 different drugs are approved by the Food and Drug Administration (FDA) for the treatment of breast cancer. The course of treatment and which drug protocols should be used is decided based on consultation between the doctor and patient, and a number of factors go into those decisions.

One of the FDA-approved drug treatments for breast cancer is tamoxifen citrate. It is sold under the brand name of Nolvadex and was first prescribed in 1969 in England but approved by the FDA in 1998. Tamoxifen is normally taken as a daily tablet with doses of 10 mg, 20 mg, or 40 mg. It carries a number of side effects including nausea, indigestion, and leg cramps. Tamoxifen has been used to treat millions of women and men diagnosed with hormone-receptor-positive breast cancer. Tamoxifen is often one of the first drugs prescribed for treating breast cancer because it has a high success rate of around 80%.

Learning that a drug is 80% successful gives us hope that tamoxifen will provide good patient outcomes, but there is one important detail about the drug that was not known until the big data era. It is that tamoxifen is not 80% effective in patients but 100% effective in 80% of patients and ineffective in the rest. That is a life-changing finding for thousands of people each year. Using techniques and ideas discussed in this book, scientists were able to identify genetic markers that can identify, in advance, if tamoxifen will effectively treat a person diagnosed with breast cancer. This type of analysis was not possible before the era of big data. Why was it not possible? Because the volume and granularity of the data was missing; volume came from pooling patient results and granularity came from DNA sequencing. In addition to the data, the computational resources needed to solve a problem like this were not readily available to most scientists outside of the super computing lab. Finally the third component, the algorithms or modeling techniques needed to understand this relationship, have matured greatly in recent years.

The story of Tamoxifen highlights the exciting opportunities that are available to us as we have more and more data along with computing resources and algorithms that aid in classification and prediction. With knowledge like that was gained by the scientists studying tamoxifen, we can begin to reshape the treatment of disease and disrupt positively many other areas of our lives. With these advances we can avoid giving the average treatment to everyone but instead determine which people will be helped by a particular drug. No longer will a drug be 5% effective; now we can identify which 5% of patients the drug will help. The concept of personalized medicine has been discussed for many years. With advances in working with big data and improved predictive analytics, it is more of a reality than ever. A drug with a 2% success rate will never be pursued by a drug manufacturer or approved by the FDA unless it can be determined which patients it will help. If that information exists, then lives can be saved. Tamoxifen is one of many examples that show us the potential that exists if we can take advantage of the computational resources and are patient enough to find value in the data that surrounds us.

We are currently living in the big data era. That term “big data” was first coined around the time the big data era began. While I consider the big data era to have begun in 2001, the date is the source of some debate and impassioned discussion on blogs—and even the *New York Times*. The term “big data” appears to have been first used, with its currently understood context, in the late 1990s. The first academic paper was presented in 2000, and published in 2003, by Francis X. Diebolt—“Big Data Dynamic Factor Models for Macroeconomic Measurement and Forecasting”—but credit is largely given to John Mashey, the chief scientist for SGI, as the first person to use the term “big data.” In the late 1990s, Mashey gave a series of talks to small groups about this big data tidal wave that was coming. The big data era is an era described by rapidly expanding data volumes, far beyond what most people imagined would ever occur.

The large data volume does not solely classify this as the big data era, because there have always been data volumes larger than our ability to effectively work with the data have existed. What sets the current time apart as the big data era is that companies, governments,

and nonprofit organizations have experienced a shift in behavior. In this era, they want to start using all the data that it is possible for them to collect, for a current or future unknown purpose, to improve their business. It is widely believed, along with significant support through research and case studies, that organizations that use data to make decisions over time in fact do make better decisions, which leads to a stronger, more viable business. With the velocity at which data is created increasing at such a rapid rate, companies have responded by keeping every piece of data they could possibly capture and valuing the future potential of that data higher than they had in the past. How much personal data do we generate? The first question is: What is personal data? In 1995, the European Union in privacy legislation defined it as any information that could identify a person, directly or indirectly. International Data Corporation (IDC) estimated that 2.8 zettabytes¹ of data were created in 2012 and that the amount of data generated each year will double by 2015. With such a large figure, it is hard to understand how much of that data is actually about you. It breaks down to about 5 gigabytes of data per day for the average American office worker. This data consists of email, downloaded movies, streamed audio, Excel spreadsheets, and so on. In this data also includes the data that is generated as information moves throughout the Internet. Much of this generated data is not seen directly by you or me but is stored about us. Some examples of nondirect data are things like traffic camera footage, GPS coordinates from our phones, or toll transactions as we speed through automated E-ZPass lanes.

Before the big data era began, businesses assigned relatively low value to the data they were collecting that did not have immediate value. When the big data era began, this investment in collecting and storing data for its potential future value changed, and organizations made a conscious effort to keep every potential bit of data. This shift in behavior created a virtuous circle where data was stored and then, because data was available, people were assigned to find value in it for the organization. The success in finding value led to more data being gathered and so on. Some of the data stored was a dead end, but many times the

¹ A zettabyte is 1 billion terabytes.

results were confirmed that the more data you have, the better off you are likely to be. The other major change in the beginning of the big data era was the rapid development, creation, and maturity of technologies to store, manipulate, and analyze this data in new and efficient ways.

Now that we are in the big data era, our challenge is not getting data but getting the right data and using computers to augment our domain knowledge and identify patterns that we did not see or could not find previously.

Some key technologies and market disruptions have led us to this point in time where the amount of data being collected, stored, and considered in analytical activities has grown at a tremendous rate. This is due to many factors including Internet Protocol version 6 (IPv6), improved telecommunications equipment, technologies like RFID, telematics sensors, the reduced per unit cost of manufacturing electronics, social media, and the Internet.

Here is a timeline that highlights some of the key events leading up to the big data era and events that continue to shape the usage of big data and the future of analytics.

BIG DATA TIMELINE

Here are a number of items that show influential events that prepared the way for the big data era and significant milestones during the era.

1991

- The Internet, or World Wide Web as we know it, is born. The protocol Hypertext Transfer Protocol (HTTP) becomes the standard means for sharing information in this new medium.

1995

- Sun releases the Java platform. Java, invented in 1991, has become the second most popular language behind C. It dominates the Web applications space and is the de facto standard for middle-tier applications. These applications are the source for recording and storing web traffic.
- Global Positioning System (GPS) becomes fully operational. GPS was originally developed by DARPA (Defense Advanced Research Projects Agency) for military applications in the early 1970s.

This technology has become omnipresent in applications for car and airline navigation and finding a missing iPhone.

1998

- Carlo Strozzi develops an open-source relational database and calls it NoSQL. Ten years later, a movement to develop NoSQL databases to work with large, unstructured data sets gains momentum.
- Google is founded by Larry Page and Sergey Brin, who worked for about a year on a Stanford search engine project called BackRub.

1999

- Kevin Ashton, cofounder of the Auto-ID Center at the Massachusetts Institute of Technology (MIT), invents the term “the Internet of Things.”

2001

- Wikipedia is launched. The crowd-sourced encyclopedia revolutionized the way people reference information.

2002

- Version 1.1 of the Bluetooth specification is released by the Institute of Electrical and Electronics Engineers (IEEE). Bluetooth is a wireless technology standard for the transfer of data over short distances. The advancement of this specification and its adoption lead to a whole host of wearable devices that communicate between the device and another computer. Today nearly every portable device has a Bluetooth receiver.

2003

- According to studies by IDC and EMC, the amount of data created in 2003 surpasses the amount of data created in all of human history before then. It is estimated that 1.8 zettabytes (ZB) was created in 2011 alone (1.8 ZB is the equivalent of 200 billion high-definition movies, each two hours long, or 47 million years of footage with no bathroom breaks).
- LinkedIn, the popular social networking website for professionals, launches. In 2013, the site had about 260 million users.

2004

- Wikipedia reaches 500,000 articles in February; seven months later it tops 1 million articles.
- Facebook, the social networking service, is founded by Mark Zuckerberg and others in Cambridge, Massachusetts. In 2013, the site had more than 1.15 billion users.

2005

- The Apache Hadoop project is created by Doug Cutting and Mike Caferella. The name for the project came from the toy elephant of Cutting's young son. The now-famous yellow elephant becomes a household word just a few years later and a foundational part of almost all big data strategies.
- The National Science Board recommends that the National Science Foundation (NSF) create a career path for "a sufficient number of high-quality data scientists" to manage the growing collection of digital information.

2007

- Apple releases the iPhone and creates a strong consumer market for smartphones.

2008

- The number of devices connected to the Internet exceeds the world's population.

2011

- IBM's Watson computer scans and analyzes 4 terabytes (200 million pages) of data in seconds to defeat two human players on the television show *Jeopardy!* (There is more about the show in Part Two.)
- Work begins in UnQL, a query language for NoSQL databases.
- The available pools in the IPv4 address space have all been assigned. IPv4 is a standard for assigning an Internet protocol (IP) address. The IPv4 protocol was based on a 32-bit number, meaning there are 2^{32} or 4.5 billion unique addresses available. This event shows the real demand and quantity of Internet-connected devices.

2012

- The Obama administration announces the Big Data Research and Development Initiative, consisting of 84 programs in six departments. The NSF publishes “Core Techniques and Technologies for Advancing Big Data Science & Engineering.”
- IDC and EMC estimate that 2.8 ZB of data will be created in 2012 but that only 3% of what could be usable for big data is tagged and less is analyzed. The report predicts that the digital world will by 2020 hold 40 ZB, 57 times the number of grains of sand on all the beaches in the world.
- The *Harvard Business Review* calls the job of data scientist “the sexiest job of the 21st century.”

2013

- The democratization of data begins. With smartphones, tablets, and Wi-Fi, everyone generates data at prodigious rates. More individuals access large volumes of public data and put data to creative use.

The events of the last 20 years have fundamentally changed the way data is treated. We create more of it each day; it is not a waste product but a buried treasure waiting to be discovered by curious, motivated researchers and practitioners who see these trends and are reaching out to meet the current challenges.

WHY THIS TOPIC IS RELEVANT NOW

You’ve read this far in the book because I expect you are looking for ideas and information to help you turn data into information and knowledge. What I hope you learn in the subsequent pages are strategies and concrete ideas for accomplishing your business objective or personal edification regarding how you can harness the data to better your situation, whether in the office, the home, or a fantasy football league.

You should also understand that this is not a new problem—data has always been “too big” to work with effectively. This problem has only been exacerbated as now individuals are generating so much more data than ever before as they go through their daily lives. This

increase in data, however, has caused the information management industry to provide better solutions than ever on how to store, manage, and analyze the data we are producing.

In addition, we also have more opportunity to engage with data. A simple example that is discussed in more detail in Part Two is the recommendations you get from Amazon. That small application at the bottom of its web pages illustrates this point very well. In order to make these recommendations, Amazon can use a few different techniques that mostly center on three pieces of information; how you are similar to other shoppers, similar shoppers' opinions of the product you are viewing, and what product similar shoppers ultimately purchased. Alternatively, Amazon could make recommendations from an item point of view. Take, for example, my recent purchase of a baseball glove. The recommendations included items like baseball bats, baseballs, baseball glove oil, and other baseball-related equipment. These recommendations are based on item-to-item recommendations. Baseball gloves are usually sold with baseballs, bats, and glove oil so Amazon recommends them to me. The other method is to look at my profile and find users who have purchased similar items or have similar details as they relate to Amazon and then recommend to me what they purchased. To be effective at making recommendations requires a real commitment to recording, storing, and analyzing extremely large volumes of data.

I think you only need to look up from this book to see a device that is generating data at this very moment. That data will soon be used to inform some business process, recommendation, or public safety issue. This not a future or theoretical problem, this is now.

IS BIG DATA A FAD?

Data! Data! Data! he cried impatiently. I can't make bricks without clay!

—Sherlock Holmes, “The Adventure in the Copper Beeches”

Many informed individuals in the analytics and information technology (IT) communities are becoming sensitive to the actual term “big data.” It has no doubt been co-opted for self-promotion by many people and organizations with little or no ties to storing and processing

large amounts of data or data that requires large amounts of computation. Aside from marketing and promotional mischaracterizations, the term has become vague to the point of near meaninglessness even in some technical situations. “Big data” once meant petabyte scale, unstructured chunks of data mined or generated from the Internet.

I submit that “big data” has expanded to mean a situation where the logistics of storing, processing, or analyzing data have surpassed traditional operational abilities of organizations; said another way, you now have too much data to store effectively or compute efficiently using traditional methods. This may also include having too little time to process the data to use the information to make decisions. In addition, big data means using all available data in your models, not just samples. Big data indicates the capability of using entire data sets instead of just segments as in years past. In the purview of wider popular usage, the definition of this term will likely continue to broaden.

For certain leaders in the field, “big data” is fast becoming, or perhaps always was, just data. These people are long accustomed to dealing with the large amounts of data that other fields are just beginning to mine for information. For evidence, look to the larger web companies and certain entities within the U.S. government that have been using extremely large amounts of unstructured data operationally for years, long before anyone ever coined the term “big data.” To them, it was just “data.” In addition, the big banks and insurance companies have been pushing up against the limits of commercial, column-oriented data storage technologies for decades, and to them this was just “data” too. Consider whether the scale at which Google is indexing all available information, or which the National Security Agency is recording, has really changed since before the term “big data” entered the popular lexicon. It was difficult to comprehend how much data this was before, and it is still is just as hard to comprehend. However, to these leaders in the field, dealing with it is just a day’s work. The rest of world is now joining these industries in storing, computing, and analyzing these immense amounts of data, and now we have a word to describe it and a time period to reference. Figure I.1 shows the popularity of the term “big data” as it came into common usage beginning in 2011. Since the 1940s when computer was a job title or in the 1960s when file transfer involved

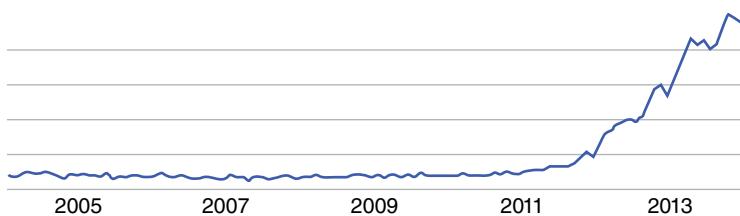


Figure I.1 Trend of Google Searches of “Big Data” over Time Showing the Popularity of the Term

Source: Google Trends

moving a carton of punch cards from one location to another and hoping you did not trip, organizations have had data challenges. Today those challenges are just on a larger scale. Nearly every company must deal with these challenges or accept the idea that the company itself may become irrelevant. *Forbes* in 2013 published an article that said that companies without a big data strategy miss out on \$71.2 million per year. If you could raise revenue over \$70 million this year and each subsequent year, I am sure your future would be very bright in your organization. The key to capitalize on this opportunity is to have a well-thought-out strategy on big data and execute to the strategy.

To be clear, the solutions surrounding the storage, processing, and analyzing “big data” are not a fad even if the term turns out to be one. Although some are overhyped right now, they are extremely valuable and in some cases actually revolutionary technologies. They have drawn such attention for this reason. Data is not magic—it’s just a valuable raw material that can be refined and distilled into valuable specific insights. These insights are then converted to information that eventually creates knowledge.

The bigger the data, the more resource-intensive it is to work with, the better the value of the information must be to make the trade-off a wise business decision. While there is simply no underlying principle stating that the size of data is positively correlated with its value, the size of a data set is positively correlated with its cost to maintain. The value of using big data is defined by how valuable the information gleaned from its process is compared to the time and resources it took to process that information.

This being said, there is a great deal of evidence that prudent data analysis can create value, whether knowledge or monetary, for most organizations. This is why the technologies that allow us to store, process, and analyze large amounts of data will continue to receive increased usage and are anything but a passing fad. For the leading analytic companies and entities, using large amounts of data has been a favorable value proposition for some time, and there is no reason for that trend to decelerate.

Data is the new oil. It has all of the same challenges in that it is plentiful but difficult and sometimes messy to extract. There are a few entities that control most of it, and a vast infrastructure has been built to transport, refine, and distribute it. With the performance of the required technologies increasing and their prices decreasing, if organizations currently struggle to deal efficiently with the medium-size data, they will be ill prepared and at a strategic disadvantage against their competition when data sizes increase, which they inevitably will do. If nothing else, companies, hospitals, and universities will face competition that will drive them to adopt technology to handle the growing volume of data. Other organizations, such as nongovernmental agencies, may be slower to invest in the new generation of data technologies and personnel. This is due to planning and implementation costs as well as the shortage of analytical professionals needed to produce value from this significant investment in hardware human capital; and some smaller organizations will not need sophisticated analysis to understand their operational environment.

WHERE USING BIG DATA MAKES A BIG DIFFERENCE

There have been so many news stories and hype about big data, and how it can transform your business, that it begins to sound like something you would find in Shangri-La. It is often portrayed as the answer to all things that cause problems for organizations. There are promises that it will identify the right customers for marketing campaigns and help academic institutions select the perfect students for their admissions process. Don't let skepticism turn you into a cynic.

It is indeed true that having more data, especially historical data, often will help model predictions be more accurate. Using additional

sources of data, such as social networks, will help organizations make better predictions about customer choices and preferences, because all of us are influenced to some degree by those in our social network, either the physical or the virtual one.

Consider a situation where I have a poor customer experience with my cable provider; so poor that I cancel all of my services and look for another provider. Does this situation make my friends, family, and associates more likely or less likely purchase new services? How does that knowledge of my cancellation because of poor customer service, along with knowing my close friends, family, and associates, affect the cable provider's action? This is a prime example of big data in action; five to ten years ago, this type of analysis would not have been possible because the data sources just did not exist. The answer to my question of how this affects those who know is that my family and friends are less likely to add new services and potentially may follow suit and cancel their cable service as well. Having more data about your customers, products, and processes allows you to consider these types of effects in predicting customers' future behavior. It also needs to be pointed out that having the data and doing the analysis are vital steps to taking advantage of the opportunity in the big data era, but unless the organization is equipped to use new data sources and methods in their processes and act on this information, all of the data and the best analysis in the world will not help it improve.

There is danger in not taking the time to know how much weight to give to this large amount of newly available information, when that information is compared to all of the other attributes that affect a person's decision-making process. Returning to the poor customer service and my social network example, it is clear that the people in my social network are now more likely to cancel services, but how much more likely? One company that takes many different metrics and creates a single aggregate score is Klout; the higher your Klout score, the more influential you are online. Klout uses comments, mentions, retweets, likes and so on to create this score. The company is able to measure online influence only because that is all the data to which it has access.

The question of sampling and big data is a hotly debated topic, and I have read and heard on several occasions that sampling is dead. As a

trained statistician and former employee of the U.S. Census Bureau, I would never say that sampling is dead or that it has no place in business today. Sampling is useful and valid. For certain types of problems, sampling from the population yields just as good a result as performing the same analysis using the entire population (all the data).

However, sampling cannot meet the objectives of many critical high-value projects, such as finding outliers. Companies that cling to sampling will miss out on opportunities to learn insights that can be found only by considering all the data. Outliers are one such example. In a statistical case, the term “outliers” usually has a negative connotation. But in many business problems, the outliers are your most profitable customers or the new market segments that can be exploited.

The best advice is to be an informed data user. Having seven years of customer history instead of three in August 2008 would not have helped in any way to predict people’s upcoming spending habits in the United States. In just a few weeks, the financial markets were going to collapse and several large investment firms were going to declare bankruptcy or be purchased at fire sale prices. The U.S. government would begin a massive bailout of the financial markets that would, in some way, affect everyone in the industrialized world. No amount of data would have helped. No amount of data analysis modeling of the preceding months’ spending could forecast what the months after the bailout would look like for the average consumer. In order to build useful models in that time, you needed competent practitioners who understood how to simulate and adjust for economic conditions that no one in the workforce had seen before.

There are, however, two major advantages of using all the available data in solving your analytical business problems. The first is technical, and the other is productivity through an improved workflow for analysts.

Technical Issue

Many statistical and machine learning techniques are averaging processes. An averaging process is an algorithm or methodology that seeks to minimize or maximize the overall error and therefore make

the best prediction for the average situation. Two examples of averaging algorithms are linear regression and neural networks. Both of these methods are explained in more detail in Part Two, but for now understand that regression seeks to minimize the overall error by fitting a line that minimizes the squared distance from the line to the data points. The square is used because the distances from the line to the data points will be both negative and positive. A neural network works by connecting all the input, or dependent variables, to a hidden set of variables and iteratively reweighting the connections between them until the classification of the holdout sample cannot be improved.

These averaging methods can be used on big data, and they will work very well. It is also very common for these methods to be very efficient in processing the data due to clever and persistent developers who have organized the computation in a way that takes advantage of modern computing systems. These systems, which have multiple cores, can each be working on a part of the problem; to get even greater speedups, multiple computers can be used in a distributed computing environment. Nonparametric techniques, such as a rank sign test, also fall into this category of an averaging model technique. I call this type of algorithm an averaging model process. Because we are seeking to average, sampling is a potential substitute that can provide comparable answers.

However, a second class of modeling problem is not averaging but is an extremity-based model or tail-based modeling process. These model types are used for a different objective and seek to find extreme or unusual values. These are sometimes referred to as outliers, but not in the strict statistical sense. Instead, there are notable and unusual points or segments that are often the problems that are the most challenging to companies and carry the biggest return on investment in the present business climate. Examples of tail-based processes are fraud detection, offer optimization, manufacturing quality control, or microsegmentation in marketing.

Next I show why it is imperative to use complete data sets in these types of problems from several different industries and domains. In these cases and many others, these problems cannot be solved effectively without all the data, which is large and complex.

Fraud

If the rate of credit card fraud is 1 in 1,000 transactions,² and you sample 20% of your data to build your fraud model, it is likely that you will not have a single fraudulent activity. In fact, if you take anything less than all the data, you likely will never include the fraudulent activity. Predictive models work by using past performance to identify current behavior that has the same characteristics. Without having those fraudulent transactions, you will lack enough information to create an equation that can recognize fraud when it is happening. It is necessary to capture and analyze past fraudulent activity to create effective models for predicting future fraudulent activity.

In addition, if you are sampling the current incoming credit card transactions looking for fraud, you will miss those transactions that could have been flagged had they been present in the data being analyzed.

Optimization

For the optimization family of problems, consider the example of U.S. commercial airline traffic. If we want to understand the congestion points in the air traffic network, or model the effect of a tornado in Dallas, or a tropical storm in Miami, we need to use the entire data set. This is necessary to measure and accurately describe the scope of the problem and its effects on overall flight delays or cancellations and measure the costs to the industry. Imagine if a sample was taken in this case. First we would have to design a sampling scheme. Do you take a sequential sample of every tenth flight across the system? Do you weigh each airport by the number of commercial flights that occur there on the average day? Do you use a random sample? All of these schemes have shortfalls in seeing the complete picture, and those shortfalls will be most prominent in the calculation of the standard error and the confidence intervals of the prediction. To do these calculations correctly will take time to measure, calculate, and verify. If it is not done correctly, then your answer is wrong, but you will never know. The sampling also requires a front-loaded investment; before

²The incidence rate is actually much smaller, but this makes for easier math.

I can work with the sampled data, I have to invest the time to create the sample and validate that it is accurate. I also cannot see any type of result until a significant amount of work is already completed.

Relationship Management

Retention of telecommunications customers is a key component to revenue stream. Consider the challenge of predicting which customers are likely to cancel their contract and move to a different carrier. (This is referred to as attrition or churn.) It would be useful to know the typical behavior patterns of customers in the period of time before they cancel, including whom they call or from whom they receive calls. If you use only 10% of the calls each customer sends or receives, use only 10% of the customers, or look at only 10% of the numbers they call, you could be misled in predicting the true likelihood that a particular customer will cancel their contract. This is true for two reasons; first, since only a small percentage of customers leave each month, it would be probable that not a single dissatisfied costumer (or even a whole segment of customers) would be included in the sample. It would also be possible that some of a dissatisfied customer's calls are not included in the sample. However, without the complete set of calls for a given customer, it is much more difficult to identify the pattern that you are looking for. (This is like working on a puzzle that is missing most of the pieces.) With the inability to identify those customers likely to cancel their contract, the problem will grow over time. Given the significant costs to acquire new customers in the telecommunications market, implementing an effective strategy to keep existing customers is worth millions and millions of dollars in annual revenue.

Work Flow Productivity

The second consideration is ensuring that the productivity of the analyst stays as high as possible. Analytics has become a very hot topic in the past few years, and predictions from McKinsey & Company project a shortfall of 140,000 to 190,000 people with the analytical expertise and 1.5 million managers needed to evaluate and make decisions based on big data. This translates to a deficit of 50% to 60% of the required personnel by the year 2018 in the United States alone.

With this significant shortfall in capable people, the human capital you have already made in your organization needs to be preserved and improved. The analytical talent you already have in your organization will become more scarce as other organizations work to make better use of their big data through better analytics and governance. It will be critical to keep analytical talent engaged and productive.

From the same report:

“Several issues will have to be addressed to capture the full potential of big data. Policies related to privacy, security, intellectual property, and even liability will need to be addressed in a big data world. Organizations need not only to put the right talent and technology in place but also structure workflows and incentives to optimize the use of big data. Access to data is critical—companies will increasingly need to integrate information from multiple data sources, often from third parties, and the incentives have to be in place to enable this.” (McKinsey) www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation

As I mentioned in the prior section, the work to sample is a very front-loaded task; the majority of the work is done before any results can be created or exploration can begin. This is really backward from the optimum work flow. The best thing is to make the access to data exploration and quick modeling against the data simple and readily available. Organizations should enable the “failing fast” paradigm. Fail has a negative connotation, but failing fast is a useful strategy for determining which projects have merit and which do not. When people have the ability to work with the entire set of data, they can explore and prototype in a more efficient and natural way that does not require a great deal of up-front work to access the data. To enable this type of environment for your organization, ongoing commitments to capital and technological investments are required.

Making effective work flow and data computing resources for employees translates to large productivity gains and short timelines to pay back the return on investment. I have seen this transformation first-hand when I was working on a credit card modeling project for a large U.S. bank. Using the traditional methods (hardware and software), it

was taking many hours to solve the problem. When I switched to a new distributed computing environment, I was able to solve the same problem in two to three minutes. I no longer had to multitask across so many projects because each one had significant downtime while models were being built. I was able to try a number of algorithms and tune each one to a degree that would not have been possible before. The work flow was reminiscent of class projects in school where data volumes were small and software ran nearly instantaneously. This was the method I had been trained in, and it felt more natural. I saw immediate benefits in the form of better model lift, which the customer saw as millions of dollars in revenue increases.

The Complexities When Data Gets Large

Big data is not inherently harder to analyze than small data. The computation of a mean is still just the sum of the values divided by the number of observations, and computing a frequency table still requires reading the data and storing the number of times a distinct value occurs in the data. Both of these situations can be done by reading the data only one time. However, when data volumes gets large or when the data complexity increases, analytics run times can grow to the point that they take longer to compute than the operational constraints will allow. This can result in misleading results or a failure to find a result at all.

Nonlinear Relationships

In real data, there are often nonlinear relationships between variables. Sometimes these relationships can be described “well enough” using linear relationships, but sometimes they cannot. A linear relationship is sometimes hard to imagine, so let us use exiting a parking lot as an example. My family and a few others attended a symphony performance and fireworks show for the Fourth of July (Independence Day in the United States). We parked near each other in the same section of the parking lot, which was about a seven-minute walk from the venue. After the fireworks concluded, our group made our way to the exit, but one of the families became separated. Instead of taking the closest exit to our seats that is open only after the event, they took

a longer route through the venue to the main entrance where we had entered. This alternate route added about three minutes to their departure time. A parking lot after a major event is always something I, as a quantitative person, dread. I easily grow frustrated over inefficiency, and this exit situation is known to be poor and bordering on terrible. My family arrived at the car, loaded our cooler, chairs, and blankets, and began to drive to the exit. Traffic inside the parking lot was quite slow, because of poor visibility and all the pedestrian traffic leaving the venue. We proceeded to move with the traffic and following police direction made it home in about 40 minutes.³ As we were arriving home, my wife received a text message from our friends who had taken the other exit, asking if we were stuck in the parking lot like they were. So, while our drive took twice as long as it does on a normal day, those three extra minutes added not three minutes (which is what we would expect from a linear relationship between time of departure and time of arrival) to their drive but almost another 45 minutes to their drive home (in addition to the 40 minutes it took my family) from the event. This example is one many can relate to, and it illustrates an important point: Knowing about your data can be a huge asset in applying analytics to your data.

A second example of nonlinear relationship is that of the space shuttle Challenger disaster in 1986. Even though it has been almost 30 years, I still remember sitting in Mrs. Goodman's class in my elementary school, with eager anticipation as we were going to see the Challenger liftoff and take a teacher, Sharon McAuliffe, into space. Many of you know the tragic events of that day and the findings of the NASA Commission. To review the details, 73 seconds after liftoff, the primary and secondary O-rings on the solid-state boosters failed and caused an explosion due to excess hydrogen gas and premature fuel ignition. This resulted in the Challenger being torn apart. The reason the O-rings failed is blamed primarily on the weather. That January day was only about 30 degrees at launch time,⁴ much colder than any space shuttle launch NASA had attempted before. The cold weather

³ The drive between my home and the venue takes about 20 minutes on the average day.

⁴ The launch control parameters for the Space Shuttle were a 24-hour average temperature of 41 degrees F and not warmer than 99 degrees F for 30 consecutive minutes.

created a problem, because NASA personnel planned assuming a linear relationship between the air temperature and O-ring performance, but instead that relationship was nonlinear and the O-ring was actually much more brittle and ineffective than preplanning had anticipated. This was a tragic lesson to learn as it cost the lives of many remarkable people. After this incident, NASA changed a number of procedures in an effort to make space flight safer.⁵

In statistics, there are several terms of art to describe the shape or distribution of your data. The terms are: mean, standard deviation, skewness, and kurtosis. At this point, the important facts to understand and keep in mind are that: (1) there are often nonlinear relationships in real-world data; (2) as the data size increases, you are able to see those relationships more clearly; and, more frequently (3) nonlinear relationships can have a very significant effect on your results if you do not understand and control for them.

In Part One of this book, the focus is on the technology aspects of creating an analytical environment for data mining, machine learning, and working with big data and the trade-offs that result from certain technology choices. In Part Two, the focus is on algorithms and methods that can be used to gain information from your data. In Part Three, case studies show how, by utilizing these new technology advances and algorithms, organizations were able to make big impacts. Part Three also illustrates that using high-performance computing, analytical staff productivity went up in meaningful ways.

⁵I got to witness the safety protocol in action. I was at Cape Canaveral for the launch of STS-111, but it was scrubbed with less than an hour before liftoff.

PART
ONE

The Computing Environment

With data collection, “the sooner the better” is always the best answer.

— Marissa Mayer

Data mining is going through a significant shift with the volume, variety, value and velocity of data increasing significantly each year. The volume of data created is outpacing the amount of currently usable data to such a degree that most organizations do not know what value is in their data. At the same time data mining is changing, hardware capabilities have also undergone dramatic changes. Just as data mining is not one thing but a collection of many steps, theories, and algorithms, hardware can be dissected into a number of components. The corresponding component changes are not always in sync with this increased demand in data mining, machine learning, and big analytical problems.

The four components of disk, memory, central processing unit, and network can be thought of as four legs of the hardware platform stool. To have a useful stool, all the legs must be of the same length or users will be frustrated, stand up, and walk away to find a better stool; so too must the hardware system for data mining be in balance in regard to the components to give users the best experience for their analytical problems.

Data mining on any scale cannot be done without specialized software. In order to explain the evolution and progression of the hardware, there needs to be a small amount of background on the traditional interaction between hardware and software. Data mining software packages are discussed in detail in Part One.

In the past, traditional data mining software was implemented by loading data into memory and running a single thread of execution over the data. The process was constrained by the amount of memory available and the speed of a processor. If the process could not fit entirely into memory, the process would fail. The single thread of execution also failed to take advantage of multicore servers unless multiple users were on the system at the same time.

The main reason we are seeing dramatic changes in data mining is related to the changes in storage technologies as well as computational capabilities. However, all software packages cannot take advantage of current hardware capacity. This is especially true of the distributed computing model. A careful evaluation should be made to ensure that algorithms are distributed and effectively leveraging all the computing power available to you.

CHAPTER 1

Hardware

I am often asked what the best hardware configuration is for doing data mining. The only appropriate answer for this type of question is that it depends on what you are trying to do. There are a number of considerations to be weighed when deciding how to build an appropriate computing environment for your big data analytics.

STORAGE (DISK)

Storage of data is usually the first thing that comes to mind when the topic of big data is mentioned. It is the storage of data that allows us to keep a record of history so that it can be used to tell us what will likely happen in the future.

A traditional hard drive is made up of platters which are actual disks coated in a magnetized film that allow the encoding of 1s and 0s that make up data. The spindles that turn the vertically stacked platters are a critical part of rating hard drives because the spindles determine how fast the platters can spin and thus how fast the data can be read and written. Each platter has a single drive head; they both move in unison so that only one drive head is reading from a particular platter.

This mechanical operation is very precise and also very slow compared to the other components of the computer. It can be a large

contributor to the time required to solve high-performance data mining problems.

To combat the weakness of disk speeds, disk arrays¹ became widely available, and they provide higher throughput. The maximum throughput of a disk array to a single system from external storage subsystems is in the range of 1 to 6 gigabytes (GB) per second (a speedup of 10 to 50 times in data access rates).

Another change in disk drives as a response to the big data era is that their capacity has increased 50% to 100% per year in the last 10 years. In addition, prices for disk arrays have remained nearly constant, which means the price per terabyte (TB) has decreased by half per year.

This increase in disk drive capacity has not been matched by the ability to transfer data to/from the disk drive, which has increased by only 15% to 20% per year. To illustrate this, in 2008, the typical server drive was 500 GB and had a data transfer rate of 98 megabytes per second (MB/sec). The entire disk could be transferred in about 85 minutes ($500\text{ GB} = 500,000\text{ MB}/98\text{ MB/sec}$). In 2013, there were 4 TB disks that have a transfer rate of 150 MB/sec, but it would take about 440 minutes to transfer the entire disk. When this is considered in light of the amount of data doubling every few years, the problem is obvious. Faster disks are needed.

Solid state devices (SSDs) are disk drives without a disk or any moving parts. They can be thought of as stable memory, and their data read rates can easily exceed 450 MB/sec. For moderate-size data mining environments, SSDs and their superior throughput rates can dramatically change the time to solution. SSD arrays are also available, but SSDs still cost significantly more per unit of capacity than hard disk drives (HDDs). SSD arrays are limited by the same external storage bandwidth as HDD arrays. So although SSDs can solve the data mining problem by reducing the overall time to read and write the data, converting all storage to SSD might be cost prohibitive. In this case, hybrid strategies that use different types of devices are needed.

Another consideration is the size of disk drives that are purchased for analytical workloads. Smaller disks have faster access times, and

¹ A disk array is a specialized hardware storage that provides larger storage capacity and data access because of its specialized implementation. NetApp and EMC are two major vendors of disk arrays.

there can be advantages in the parallel disk access that comes from multiple disks reading data at the same time for the same problem. This is an advantage only if the software can take advantage of this type of disk drive configuration.

Historically, only some analytical software was capable of using additional storage to augment memory by writing intermediate results to disk storage. This extended the size of problem that could be solved but caused run times to go up. Run times rose not just because of the additional data load but also due to the slower access of reading intermediate results from disk instead of reading them from memory. For a typical desktop or small server system, data access to storage devices, particularly writing to storage devices, is painfully slow. A single thread of execution for an analytic process can easily consume 100 MB/sec, and the dominant type of data access is sequential read or write. A typical high-end workstation has a 15K RPM SAS drive; the drive spins at 15,000 revolutions per minute and uses the SAS technology to read and write data at a rate of 100 to 150 MB/sec. This means that one or two cores can consume all of the disk bandwidth available. It also means that on a modern system with many cores, a large percentage of the central processing unit (CPU) resources will be idle for many data mining activities; this is not a lack of needed computation resources but the mismatch that exists among disk, memory, and CPU.

CENTRAL PROCESSING UNIT

The term “CPU” has had two meanings in computer hardware. CPU is used to refer to the plastic and steel case that holds all the essential elements of a computer. This includes the power supply, motherboard, peripheral cards, and so on. The other meaning of CPU is the processing chip located inside the plastic and steel box. In this book, CPU refers to the chip.

The speed of the CPU saw dramatic improvements in the 1980s and 1990s. CPU speed was increasing at such a rate that single threaded software applications would run almost twice as fast on new CPU versions as they became available. The CPU speedup was described by Gordon Moore, cofounder of Intel, in the famous Moore’s law, which is an observation that the number of transistors and integrated circuits

that are able to be put in a given area doubles every two years and therefore instructions can be executed at twice the speed. This trend in doubling CPU speed continued into the 1990s, when Intel engineers observed that if the doubling trend continued, the heat that would be emitted from these chips would be as hot as the sun by 2010. In the early 2000s, the Moore's law free lunch was over, at least in terms of processing speed. Processor speeds (frequencies) stalled, and computer companies sought new ways to increase performance. Vector units, present in limited form in x86 since the Pentium MMX instructions, were increasingly important to attaining performance and gained additional features, such as single- and then double-precision floating point.

In the early 2000s, then, chip manufacturers also turned to adding extra threads of execution into their chips. These multicore chips were scaled-down versions of the multiprocessor supercomputers, with the cores sharing resources such as cache memory. The number of cores located on a single chip has increased over time; today many server machines offer two six-core CPUs.

In comparison to hard disk data access, CPU access to memory is faster than a speeding bullet; the typical access is in the range of 10 to 30 GB/sec. All other components of the computer are racing to keep up with the CPU.

Graphical Processing Unit

The graphical processing unit (GPU) has gotten considerable publicity as an unused computing resource that could reduce the run times of data mining and other analytical problems by parallelizing the computations. The GPU is already found in every desktop computer in the world.

In the early 2000s, GPUs got into the computing game. Graphics processing has evolved considerably from early text-only displays of the first desktop computers. This quest for better graphics has been driven by industry needs for visualization tools. One example is engineers using three-dimensional (3D) computer-aided design (CAD) software to create prototypes of new designs prior to ever building them. An even bigger driver of GPU computing has been the consumer video game industry, which has seen price and performance trends similar to the rest of the consumer computing industry. The relentless

drive to higher performance at lower cost has given the average user unheard-of performance both on the CPU and the GPU.

Three-dimensional graphics processing must process millions or billions of 3D triangles in 3D scenes multiple times per second to create animation. Placing and coloring all of these triangles in their 3D environment requires a huge number of very similar calculations. Initially, 3D graphics were done using a fixed rendering pipeline, which took the 3D scene information and turned it into pixels that could be presented to the user in a video or on the screen. This fixed pipeline was implemented in hardware, with various parts of the GPU doing different pieces of the problem of turning triangles into pixels. In the early 2000s, this fixed pipeline was gradually replaced by generalized software shaders, which were miniprograms that performed the operations of the earlier fixed hardware pipeline.

With these shaders, high-performance computing folks noticed that the floating-point coordinates and colors could look an awful lot like physics or chemistry problems if you looked at them just right. The more hardcore hacker types started creating graphics problems that looked a lot like nonsense except that the underlying calculations being done solved hard problems remarkably fast. The performance gains got noticed, and computing frameworks, which used the GPUs for doing nongraphics calculations, were developed. These calculations are the same type needed for data mining.

GPUs are a green field. Historically the ability to develop code to run on the GPU was restrictive and costly. Those programming interfaces for developing software that takes advantage of GPUs have improved greatly in the last few years. Software has only started to take advantage of the GPU, and it will be several years before the computations needed for data mining are efficiently delegated to the GPU for execution. When that time comes, the speedup in many types of data mining problems will be reduced from hours to minutes and from minutes to seconds.

MEMORY

Memory, or random access memory (RAM) as it is commonly referred to, is the crucial and often undervalued component in building a data mining platform. Memory is the intermediary between the storage of

data and the processing of mathematical operations that are performed by the CPU. Memory is volatile, which means that if it loses power, the data stored in it is lost.

In the 1980s and 1990s, the development of data mining algorithms was very constrained by both memory and CPU. The memory constraint was due to the 32-bit operating systems, which allow only 4 GB of memory to be addressed. This limit effectively meant that no data mining problem that required more than 4 GB of memory² (minus the software and operating system running on the machine) could be done using memory alone. This is very significant because the data throughput of memory is typically 12 to 30 GB/sec, and the fastest storage is only around 6 GB/sec with most storage throughput being much less.

Around 2004, commodity hardware (Intel and AMD) supported 64-bit computing. At the same time operating systems became capable of supporting larger amounts of memory, the actual price of memory dropped dramatically. In 2000, the average price of 1 MB of RAM was \$1.12. In 2005, the average price was \$0.185; and in 2010, it was \$0.0122.

With this support of 64-bit computing systems that can address up to 8 TB of memory and the drop in memory prices, it was now possible to build data mining platforms that could store the entire data mining problem in memory. This in turn produced results in a fraction of the time.

Data mining algorithms often require all data and computation to be done in memory. Without external storage, the increase in virtual and real address space as well as the dramatic drop in the price of memory created an opportunity to solve many data mining problems that previously were not feasible.

To illustrate this example, consider a predictive modeling problem that uses a neural network algorithm. The neural network will perform an iterative optimization to find the best model. For each iteration, it will have to read the data one time. It is not uncommon for neural networks to make thousands of passes through the data to find

²The largest integer value that 32-bit operating systems can use to address or reference memory is $2^{32}-1$, or 3.73 GB, of memory.

the optimal solution. If these passes are done in memory at 20 GB/sec versus on disk at 1 GB/sec, a problem that is only 10 seconds to solve in memory will be more than 3 minutes to solve using disk. If this scenario is repeated often, the productivity of the data miner plummets. In addition to the productivity of the human capital, if the data mining processes relied on disk storage, the computation would take many times longer to complete. The longer a process takes to complete, the higher the probability of some sort of hardware failure. These types of failure are typically unrecoverable, and the entire process must be restarted.

Memory speeds have increased at a much more moderate rate than processor speeds. Memory speeds have increased by 10 times compared to processor speeds, which have increased 10,000 times. Disk storage throughput has been growing at an even slower rate than memory. As a result, data mining algorithms predominantly maintain all data structures in memory and have moved to distributed computing to increase both computation and memory capacity. Memory bandwidth is typically in the 12 to 30 GB/sec range, and memory is very inexpensive. High-bandwidth storage maxes out in the 6 GB/sec range and is extremely expensive. It is much less expensive to deploy a set of commodity systems with healthy amounts of memory than to purchase expensive high-speed disk storage systems.

Today's modern server systems typically come loaded with between 64 GB and 256 GB of memory. To get fast results, the sizing of memory must be considered.

NETWORK

The network is the only hardware component that is always external to the computer.³ It is the mechanism for computers to communicate to other computers. The many protocols and standards for network communication will not be discussed here beyond the very limited details in this section.

The network speed should be a factor only for a distributed computing environment. In the case of a single computer (workstation or

³ Storage can sometimes be external in a storage area network (SAN).

server), the data, memory, and CPU should all be local, and performance of your analytical task will be unaffected by network speeds.

The standard network connection for an analytical computing cluster is 10 gigabit Ethernet (10 GbE), which has an upper-bound data transfer rate of 4 gigabytes per second (GB/sec). This data transfer rate is far slower than any of the other essential elements that have been discussed. Proprietary protocols like Infiniband® give better data throughput but still do not match the speed of the other components. For this reason, it is very important to minimize usage of the network for data movement or even nonessential communication between the different nodes in the computing appliance.

It is this network speed bottleneck that makes parallelization of a number of the data mining algorithms so challenging. Considerable skill in the software infrastructure, algorithm selection, and final implementation is required to fit a model efficiently and precisely using one of many algorithms while not moving any data and limiting communication between computers.

The network speed of your high-performance data mining platform will be important if you have a distributed computing environment. Because the network data transfer rate is much slower than other components, you must consider the network component when evaluating data mining software solutions.

CHAPTER 2

Distributed Systems

If you look back before electronic computers, the term “computer” meant a person who performed numerical calculations. Because roomfuls of these computers were in use for most of human history, arguably distributed (i.e., multiperson) computing dates back much farther than conventionally asserted.

Modern distributed computing arguably derives from the 1990s efforts to build “Beowulf” clusters of computers and from early ad hoc computing.¹ Beowulf clusters of computers were standard server or even desktop computers that were networked. The software on the computers then communicated closely with each other over the network to split the work to be done across the computers. Early ad hoc computing efforts, including distributed.net and SETI@Home, used idle home computers to search for cryptography keys and alien radio signals in radio astronomy data. Neither of these projects had the computing power in-house to solve the problem, nor sufficient budget to do it. However, the problems were simple enough that a large number

¹ I tried with limited success to build a Beowulf cluster in the late 1990s with cast-off computers.

of computers could make considerable progress with minimal coordination between them. Home computers running the distributed.net or SETI@Home software contacted the main server to get a chunk of work to be done (keys to check or radio signal data to examine).

Classical supercomputers were very large, very expensive machines that contained specialized fast hardware and processors. These supercomputers had contained in their specialized hardware features such as multiple central processing units and vector hardware. Vector hardware is used to perform identical operations on multiple pieces of data. For instance, a two-dimensional vector has components X and Y . Adding a set of vectors is done by components, so providing a single instruction that adds both X and Y simultaneously can result in doubling processing speed.

The cost of hardware has dropped dramatically. Both high-core/large-memory (massively parallel processing [MPP] systems) and clusters of moderate systems allow much larger data mining projects to be feasible. For big data analytics, the only solution is to move the analytics to the data; it is impractical to move the data to the analytics because of the time required for the data transfer.

Harder problems—problems that consume much larger volumes of data, with much higher numbers of variables—may now be considered for analysis. Cluster computing can be divided into two main groups of distributed computing systems. The first is the database, which has been a fixture in data centers for decades. The second is the distributed file system, which is currently dominated by Hadoop.

DATABASE COMPUTING

The relational database management system (RDBMS) or simply database has been around since the 1970s and until recently has been the most common place to store data generated by organizations. A number of huge vendors as well as open source projects provide database systems. Traditional RDBMSs are designed to support databases that are much larger than the memory or storage available on a single computer. From those early days, there are now a number of different databases that serve special purposes for organizations around high-performance data mining and big data analytics.

In-memory databases (IMDBs) were developed starting in the 1990s. IMDBs are now a popular solution used to accelerate mission-critical data transactions for finance, e-commerce, social media, information technology, and other industries. The idea behind IMDB technology is straightforward—holding data in memory and not on disk increases performance. However, there are notable drawbacks to IMDBs, namely the increased cost and volatility of memory.

MPP (massively parallel processing) databases began to evolve from traditional DBMS technologies in the 1980s. MPP databases are meant to serve many of the same operational, transactional, and analytic purposes as the previous generation of commercial databases but offer performance, availability, and scalability features designed to handle large volumes of data while utilizing standard user interfaces. MPP databases are positioned as the most direct update for organizational enterprise data warehouses (EDWs). The technology behind MPP databases usually involves clusters of commodity or specialized servers that hold data on multiple hard disks.

A big advantage of database computing is the time saved in moving the data to the analytics. In a truly big data problem, the time to move the data and the hardware resources needed to process it efficiently once it is moved make the strategy inefficient. Using software that can move the analytics to the data and process it in place leveraging the large computational resources that a distributed database provides will lead to faster models and shorter run times when compared to nondistributed computing systems.

FILE SYSTEM COMPUTING

There are many options in choosing a platform for file system computing, but the market is rapidly consolidating on Hadoop with its many distributions and tools that are compatible with its file system, such as Hive, MapReduce, and HBase.

Hadoop was created by Doug Cutting and Mike Cafarella. (See Figure 2.1.) The name “Hadoop” is not an acronym or a reference to anything really. It came from the name that Cutting’s son had given to a stuffed yellow elephant. It was initially developed in 2004 based

on the work Cutting and Cafarella had done on Nutch² and a paper published by Google that introduced the MapReduce paradigm for processing data on large clusters. In 2008, Hadoop had become a top-level Apache project and was being used by several large data companies such as Yahoo!, Facebook, and *The New York Times*.

A current trend is to store all available data in Hadoop. Hadoop is attractive because it can store and manage very large volumes of data on commodity hardware and can expand easily by adding hardware resources with incremental cost. Traditionally, systems were sized based on expected volumes of data, without the expectation that data would accumulate in perpetuity. Hadoop has made large-scale accumulation of data feasible and potentially is a significant differentiator for competitive advantage. Those who can exploit the value of historical data successfully can gain a huge advantage in the future.

Hadoop is becoming the front-runner for housing large volumes of historical data. However, this data is rarely (actually probably never) in an appropriate form for data mining.

Also, for many problems, other data repositories are used to augment the data in Hadoop. For example, credit card transactions may be stored in Hadoop, but the cardholder account information may be

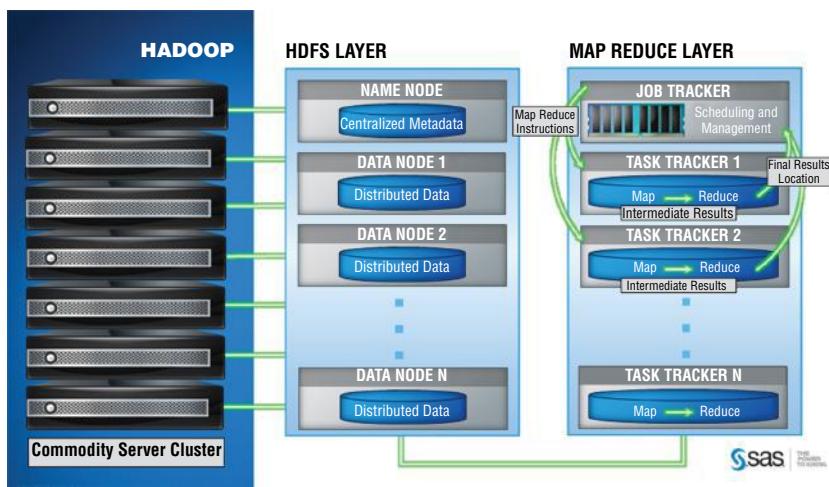


Figure 2.1 Graphical illustration of a Hadoop System

² Nutch is an Apache open source web crawling project.

stored and maintained in a traditional database. The data survey step includes identifying the data and the data repositories that will be used in the modeling process. Then the data must be combined, summarized, and stored for data mining. This work normally will be done in Hadoop because the computational cost per unit is lower than MPP or RDBMS. This hybrid approach to storing data is likely to be common practice for many years until either Hadoop matures to the level of databases or another technology emerges that makes both databases and Hadoop less desirable.

Cloud computing, because it is able to rapidly provision new servers or to bring down servers no longer needed, can be used in either capacity. However, not all of the high-end tweaks, such as specialized networking hardware, are available. Their flexibility allows one to rapidly change between simple ad hoc computing and coordinated computing and even to mix the models on the fly.

CONSIDERATIONS

Here are some questions to consider as you consider your high-performance data mining platform.

- What is the size of your data now?
- What is the anticipated growth rate of your data in the next few years?
- Is the data you are storing mostly structured data or unstructured data?
- What data movement will be required to complete your data mining projects?
- What percentage of your data mining projects can be solved on a single machine?
- Is your data mining software a good match for the computing environment you are designing?
- What are your users' biggest complaints about the current system?

Figure 2.2 compares a number of big data technologies. The figure highlights the different types of systems and their comparative strengths

	In-Memory Database	MPP Database	Big Data Appliance	Hadoop	NoSQL Database
Consistent	●	●	●	▲	▲
Available	●	●	●	▲	▲
Fault tolerant	●	●	▲	●	●
Suitable for real-time transactions	●	●	●	◆	◆
Suitable for analytics	▲	▲	●	●	◆
Suitable for extremely big data	◆	▲	▲	●	●
Suitable for unstructured data	◆	◆	▲	●	●

Figure 2.2 Comparison of Big Data Technologies

and weaknesses. The purchasing decision for the data mining computing platform will likely be made by the IT organization, but it must first understand the problems being solved now and the problems that are needed to be solved in the near future. The consideration of platform trade-offs and the needs of the organization, all shared in a transparent way, will lead to the best outcome for the entire organization and the individual stakeholders.

In Figure 2.2 the symbols have the following meaning:

- Circle: Meets widely held expectations.
- Triangle: Potentially meets widely held expectations.
- Diamond: Fails to meet widely held expectations

Big data/data mining projects must address how data moves through the entire end-to-end process.

The computing environment is critical to success: Your computing environment comprises four important resources to consider: network, disk, central processing units (CPUs), and memory. Time to solution goals, expected data volumes, and budget will direct your decisions

regarding computation resources. Appropriate computing platforms for data analysis depend on many dimensions, primarily the volume of data (initial, working set, and output data volumes), the pattern of access to the data, and the algorithm for analysis. These will vary depending on the phase of the data analysis.

CHAPTER 3

Analytical Tools

Information is the oil of the 21st century, and analytics is the combustion engine.

—Peter Sondergaard

When faced with a business challenge, people need tools to overcome issues and find ways to create value. In the analytics forum, frequently that begins very early on with deciding what tool should be used. This is the third consideration before work is begun. The first two, where to store the data and how to prepare it for analysis, were discussed in Chapters 1 and 2. This chapter details some tools that are commonly used. Some I have great familiarity with and, in fact, contributed to their development. Others I have used as tools, just as you or your teams will, and a few I have not used personally, but they also should be considered. I have tried to provide a balanced assessment of the strengths and weakness of each tool.

WEKA

Weka (Waikato Environment for Knowledge Analysis) is an open source data mining offering, fully implemented in Java, and primarily developed at the University of Waikato, New Zealand. Weka is notable for its broad range of extremely advanced training algorithms, its

work flow graphical user interface (GUI), and its incorporation of data visualization tools. Weka allows users access to its sophisticated data mining routines through a GUI designed for productive data analysis, a command line interface, and a Java application programming interface (API). However, Weka does not scale well for big data analytics, as it is limited to available RAM resources, typically on one machine. Users with 64-bit operating systems will have access to much larger RAM resources, but Weka's documentation directs users to its data preprocessing and filtering algorithms to sample big data before analysis. As many of the most powerful algorithms available in Weka are unavailable in other environments without custom software development, sampling may be the best practice for use cases necessitating Weka. In recent releases, Weka has made strides toward multithreading and simple multitasking. In Weka 3.6, some classifiers can train models on multiple cross-validation folds simultaneously, and the Weka server deployment allows for running data mining tasks concurrently on one machine or a cluster. Weka routines also provide advanced analytics components for two Java-based big data analysis environments: Pentaho and MOA (Massive Online Analysis).

Weka is a good option for people who may not have Java programming experience and need to get started quickly to prove value. I used Weka during a survey course in graduate school about data mining tools. It was a few years after Weka was completely rewritten to use Java. The interface provided big productivity gains for me, because I was not a very experienced Java programmer. I was able to analyze data with far less start-up investment in the programming language.

JAVA AND JVM LANGUAGES

For organizations looking to design custom analytics platforms from scratch, Java and several other languages that run on the Java Virtual Machine (JVM) are common choices. For large, concurrent, and networked applications, Java presents considerable development advantages over lower-level languages, without excessive performance sacrifices, especially in the realm of big data analytics. While languages that execute directly on native hardware, particularly FORTRAN and C, continue to

outperform Java in RAM and CPU-bound calculations, technological advances in the Java platform have brought its performance nearly in line with native languages for input/output and network-bound processes, like those at the core of many open source big data applications. Probably the most recognizable Java-based big data environment, Apache Hadoop, beat out several other technologies in the 2008 and 2009 TeraByte Sort Benchmark. It was the first Java-based technology to win the well-respected bench marking contest.

Java's advantages and disadvantages as a general-purpose programming language have been widely discussed in many forums, and will be addressed only briefly here. Developers of analytic applications often avoided Java in the past, because the exhaustive runtime library was unnecessary for numerical routines and the memory management by the JVM platform caused unacceptable slowdowns, to name just two reasons. As analytic applications grew dramatically in scale and complexity, development time became a more serious concern, and memory and CPU-bound performance became less so. In the specific context of building data mining applications, Java's strengths arise from its development efficiency: its rich libraries, many application frameworks, inherent support for concurrency and network communications, and a preexisting open source code base for data mining functionality, such as the Mahout and Weka libraries. Although at some cost to performance, development advantages also stem from the JVM platform. In addition to portability, the JVM provides memory management, memory profiling, and automated exception handling.

Scala and Clojure are newer languages that also run on the JVM and are used for data mining applications. Scala is an open source language developed in Switzerland at the École Polytechnique Fédérale de Lausanne. It was first released in 2003 and aims to be an improvement on Java.

Scala is used to construct data mining applications because it encompasses the development efficiencies of Java—through interoperability with the Java Runtime Environment—and adds functional programming, all with a more concise syntax. Scala allows developers to switch between functional programming and object-oriented (OO) programming paradigms, whereas Java's grammar and syntax tend to enforce

OO design. Functional languages are regarded positively for data mining applications because they handle memory differently from OO and procedural languages. For instance, a major concern in procedural and OO multithreaded applications is preventing multiple threads from changing the same variable at the same time. Functional programming languages avoid this by never changing variables. Even though it is a less common programming paradigm, functional programming is a valuable tool that should not be discounted. Scala is gaining in popularity and has been used to implement major open source projects like Akka and Spark. Akka is a toolkit for building large, parallel, and distributed applications on the JVM, and Spark is a high-performance data mining environment from the University of California Berkeley AMP Lab. Scala is also used commercially for various purposes by FourSquare, the Guardian, LinkedIn, Novell, Siemens, Sony, Twitter, and others.

Clojure was written by Rich Hickey and released in 2007. It is a functional programming language and a dialect of the Lisp programming language, with additional scripting and concurrency features. Although not designed specifically for big data applications, these intrinsic characteristics make Clojure an excellent candidate for building data analysis software. Clojure defaults to immutable data structures, removing an entire category of thread safety design concerns as compared to procedural and OO languages. Clojure treats data in a very holistic manner, presenting developers many convenient ways to process its comprehensive set of data structures and inheriting the concept of code as data from Lisp. This approach to sequences of data eliminates another class of common programming errors related to boundary conditions, specifically in comparison to C and FORTRAN. Clojure also inherits its simple syntax, lazy evaluation, and highly effective macro facility from Lisp. The macro facility allows for the creation of domain-specific languages (DSLs), and DSLs for SQL (ClojureQL) and Hadoop (Cascalog) integration have already been established. When Clojure's unique strengths are combined with those of the JVM platform and access to Java libraries, it becomes a formidable general-purpose programming asset. The first adopters of Clojure have generally been smaller, technologically or analytically advanced companies and start-ups. However, larger corporations and institutions, such as Citigroup and the Max Planck Institute, have reported using Clojure as well.

R

R is an open source fourth-generation programming language designed for statistical analysis. R grew from the commercial S language, which was developed at Bell Laboratories starting in the late 1970s. R was ported from S and SPLUS—another commercial implementation now licensed by TIBCO—by researchers at the University of Auckland, New Zealand, and first appeared in 1996.

R is very popular in the academic community, where I was first exposed to it. I used it to experiment with new analysis that I didn't have time or expertise to code myself and to complete homework assignments. At the time the software also had wonderful graphing functionality superior to others I had used. I leveraged this graphing facility in graduate school and while working at the U.S. Census Bureau. Although these graphing facilities are still present in R, it has lost its competitive advantage to other software.

R has grown to a position of prominence in the broad and growing data science community, enjoying wide usage in academic settings and increasing acceptance in the private sector. Large companies, such as Bank of America, Google, Intercontinental Hotels, Merck, Pfizer and Shell, are known to use R for a variety of purposes. The September 2013 TIOBE general survey of programming languages put R in 18th place in overall development language popularity, far outranking S and S-SPLUS, and in close proximity to commercial solutions like SAS (at 21st) and MATLAB (at 19th) (TIOBE, 2013).¹ R is also extremely customizable. There are thousands of extensions for R, up from about 1,600 in 2009. Extension packages incorporate everything from speech analysis, to genomic science, to text mining into R's baseline analysis functionality. R also boasts impressive graphics, free and polished integrated development environments (IDEs), programmatic access to and from many general-purpose languages, interfaces with popular proprietary analytics solutions including MATLAB and SAS, and even commercial support from Revolution Analytics.

¹ The TIOBE programming community index is a measure of popularity of programming languages, calculated from a number of search engine results for queries containing the name of the language.

R is experiencing a rapid growth in popularity and functionality, and many of the memory issues that made dealing with larger data difficult in previous versions of R have been resolved. Significant development work is currently under way to circumvent remaining memory limitations through cluster computing and interfaces to Hadoop. R's short update cycle has made this work available to R consumers through official packages like snow, multicore, and parallel, and younger projects like RHadoop and RHIPE. Aside from Rhadoop and RHIPE, these packages are not meant for true big data analysis but for CPU- and memory-intensive calculations of the “embarrassingly parallel” variety, often executed through R's lapply() construct, where a function is applied to all the elements of a list.

Snow is used to run R calculations on a cluster of computers. It uses sockets, MPI, PVM or NetWorkSpaces to communicate between cluster nodes.² The snow package employs traditional master-worker parallelization architecture, holds results in memory on the master node, and requires arguments to snow function calls to fit in memory. Thus, being memory bound, the package is used primarily for CPU-intensive computations, such as simulations, bootstrapping, and certain machine learning algorithms.

Multicore is a simple-to-install-and-implement package that splits sequential R processes running on a single POSIX-compliant machine (OS X, Linux, or UNIX) into a multithreaded process on the same machine. Processes executed simultaneously using multicore are limited to single-machine RAM and other shared memory constraints.

Parallel is a parallel execution package that comes standard in R 2.14 and higher. It expands out-of-the-box parallel execution to the cluster level for POSIX-compliant operating systems and to the multiprocessor level for Windows. Parallel shares a great deal of syntax,

² PVM (Parallel Virtual Machine) is a software package that permits a heterogeneous collection of Unix and/or Windows computers connected together by a network to be used as a single large parallel computer. Thus, large computational problems can be solved more cost effectively by using the aggregate power and memory of many computers. The software is very portable. NetWorkSpaces (NWS) is a powerful, open source software package that makes it easy to use clusters from within scripting languages like Matlab, Python, and R. While these scripting languages are powerful and flexible development tools, their inherent ease of use can, in many cases, cause natural limitations.

functionality, and limitations with both snow and multicore, and can use snow cluster objects.

Rhadoop and **Rhipe** allow programmatic access to Hadoop, using the R language. Map and reduce tasks can be run from R on a Hadoop cluster.

PYTHON

From its inception in the late 1980s, Python was designed to be an extensible, high-level language with a large standard library and a simple, expressive syntax. Python can be used interactively or programmatically, and it is often deployed for scripting, numerical analysis, and OO general-purpose and Web application development. Python is an interpreted language and typically performs computationally intensive tasks slower than native, compiled languages and JVM languages. However, Python programs can call compiled, native FORTRAN, C and C++ binaries at run time through several documented APIs. Python programs can be multithreaded, and like JVM languages, the Python platform conducts memory management and exception handling, relieving the developer from that responsibility. The Python language has also been ported to the JVM platform, in a project called Jython.

Python's general programming strengths combined with its many database, mathematical, and graphics libraries make it a good fit for projects in the data exploration and data mining problem domains. Python's inherent file handling and text manipulation capabilities, along with its ability to connect with most SQL and NoSQL databases, allow Python programs to load and process raw and tabular data with relative ease. Mathematical and statistical models can be implemented using the Scipy and Numpy libraries, which have strong support for linear algebra, numerical analysis, and statistical operations. Newer Python libraries, like Orange and Pattern, provide high-level data mining APIs, and the Matplotlib library can generate complex and nuanced data visualizations. Python can also be used to write map and reduce directives for MapReduce jobs using the Hadoop Streaming utility.

Python recently has become much more used in the data mining and data science communities with the maturity of the scikit-learn toolkit. The toolkit is currently on release 0.14 but offers a number of

useful algorithms and manipulation techniques. Python's scikit does not have any user interface beyond the programming IDE, which allows users flexibility but makes the initial learning curve steep and decreases productivity for routine tasks.

SAS

SAS is the leading analytical software on the market. In the 2013 IDC report ("Worldwide Business Analytics Software"), SAS not only had a significant market share but held more market share than the next 16 vendors combined. The SAS System was first sold in 1976, and two of the four cofounders are still active in the company—Dr. Jim Goodnight and John Sall. The SAS System is divided into a number of product areas including statistics, operations research, data management, engines for accessing data, and business intelligence (BI). For the topic of this book, the most relevant products are SAS/STAT®, SAS® Enterprise Miner™, and the SAS text analytics suite. SAS has been rated as the leading vendor in predictive modeling and data mining by the most recent Forrester Wave and Gartner Magic Quadrant. These ratings reveal the breadth and depth of the capability of the software. In my experience, there is no analytical challenge that I have not been able to accomplish with SAS. Having worked for the company for almost a decade, I have a much better understanding of the meticulous work and dedication of the development team and the tremendous lengths they go to ensure software quality, meet customer requests, and anticipate customers' future needs.

The SAS system is divided into two main areas: procedures to perform an analysis and the fourth-generation language that allows users to manipulate data. This is known as the DATA Step.

One of the unique achievements of SAS is the backward compatibility that is maintained release after release. I saw this firsthand at the U.S. Census Bureau. We had several SAS programs that were well over 15 years old, and they provided the same verifiable results with each update to SAS. This was essential because frequently the original author of the computer programs had moved on to other departments or in some cases no longer worked at the Census Bureau.

SAS uses a comprehensive scheme to provide the best computational performance while being flexible to individual customer environments. The procedure, or PROC as it is commonly referred to, is a module built for a specific analysis, like the REG procedure for building regression models or the UNIVARIATE procedure for doing descriptive statistics on a single variable. In a simple example, let us take a task of computing a mean for a given variable.

This can be trivially accomplished in any programming language because the mean is simple: \sum_i^n / n . This can be computed in the data step with code like this:

```
Data a;
      Set b end=last;
      Retain x;
      Sum+x;
      If last then mean=sum/_n_;
      Run;
```

This example shows where a new data set that will be created from data set b. The retain statement tells SAS to keep the values when moving to a new observation (row); the if statement tells SAS that if this is the last observation in the data set, divide the sum by the number of observations, which is stored in the automatic variable _n_. This code is not difficult to write and explain, but there is an alternative approach that is better for SAS users:

```
Proc means data=b;
      Var x;
      Run;
```

Besides being shorter, this code is also easier to read, uses code written for a specific purpose instead of general-purpose code, and I did not need to make a copy of the data just to add one new column. The procedure also handles and documents a number of behaviors that become very important when you develop your own software (e.g., working with ties, missing values, or other data quality issues). SAS has 82 procedures in the SAS/STAT product alone. Literally hundreds of procedures perform specific analyses. A major advantage of SAS over other software packages is the documentation. SAS/STAT has over 9,300 pages of documentation and Enterprise Miner has over 2,000 pages.

The SAS system processes data in memory when possible and then uses system page files to manage the data at any size efficiently. In the last few years, SAS has made major changes in its architecture to take better advantage of the processing power and falling price per FLOP (floating point operations per second) of modern computing clusters.

PART
TWO

Turning Data into Business Value

This second part of the book shifts from the storage of data, preparation of data, hardware considerations, and the software tools needed to perform data mining to the methodology, algorithms, and approaches that can be applied to your data mining activities. This includes a proven method for effective data mining in the sEMMA approach, discussion about the different types of predictive modeling target models, and understanding which methods and techniques are required to handle that data effectively. From my experience, most business environments use several people to perform each of the tasks. In larger organizations, the tasks might be split across many groups and organizationally only meet at the executive level of the organization. A quote from Shakespeare that I have always appreciated to introduce this topic is:

If you can look into the seeds of time, And say which grain will grow and which will not, Speak then to me, who neither beg nor fear Your favours nor your hate.

—*Macbeth*, Act 1, Scene 3

This verse shows an appreciation and understanding from hundreds of years ago that those people who can predict future behavior have a distinct advantage regardless of the venue. In sports, this is often call field presence: the talent some players have to effectively anticipate where the ball will be played and be there before the ball arrives. On Wall Street, fortunes are won by correctly anticipating the movement of the market in advance. In your business, correctly anticipating customer behavior or reducing future expenditures has a real impact. The goal in this section is to explain a methodology for predictive modeling. This process has been in place for over a decade and proven useful for thousands and thousands of users. I also discuss how this methodology applies in the big data era, which reduces or makes optional the need for data sampling.

Part Two discusses the types of target models, their characteristics, and information about their specific uses in business.

In addition, I discuss a number of predictive modeling techniques to help you understand the fundamental ideas behind these techniques, their origins, how they differ, and some of their drawbacks.

Finally, I present a set of methods that you might be less familiar with that address more modern methods for analysis or analysis on specific types of data.

CHAPTER 4

Predictive Modeling

I never guess. It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.

—Sir Arthur Conan Doyle,
author of Sherlock Holmes stories

Predictive modeling is one of the most common data mining tasks. As the name implies, it is the process of taking historical data (the past), identifying patterns in the data that are seen through some methodology (the model), and then using the model to make predictions about what will happen in the future (scoring new data).

Data mining is a composite discipline that overlaps other branches of science. In Figure 4.1, we can see the contributions of many different fields in the development of the science of data mining. Because of the contributions of many disciplines, staying up to date on the progress being made in the field of data mining is a continuous educational challenge. In this section I discuss algorithms that come primarily from statistics and machine learning. These two groups largely live in different university departments (statistics and

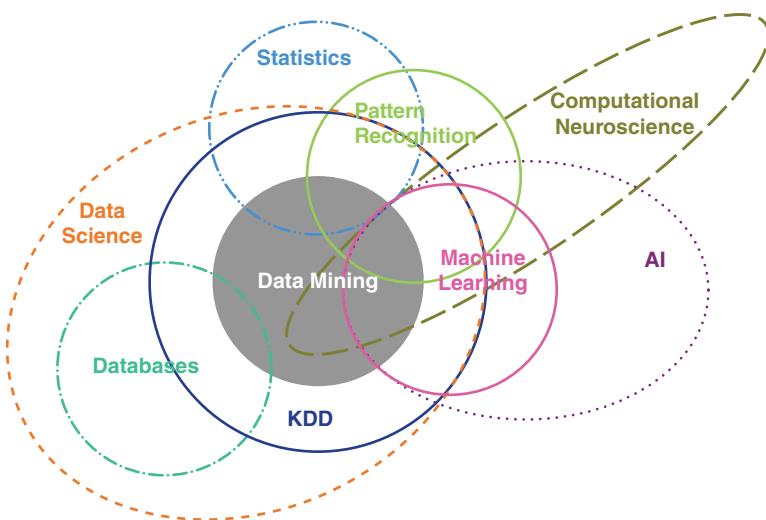


Figure 4.1 Multidisciplinary Nature of Data Mining
Source: SAS Enterprise Miner Training material from 1998.

computer science respectively) and in my opinion are feuding about the best way to prepare students for the field of data science. Statistics departments teach a great deal of theory but produce students with limited programming skills. Computer science departments produce great programmers with a solid understanding of how computer languages interact with computer hardware but have limited training on how to analyze data. This gap requires organizations to train new hires in these areas. There is a recent trend to give students a more complete education and better prepare them to contribute immediately in a data mining role. Until these programs build up their student bases, job applicants will likely know only half of the algorithms commonly used for modeling. For statisticians, that is regression, General Linear Models (GLMs), and decision trees. For the computer scientist, it is neural networks, support vector machines, and Bayesian methods.

Predictive modeling is not necessarily complicated or difficult; it is a fundamental task that we learn at an early age and hone as we grow and gain experience. In your daily life, you inherently do this many times a day. For instance, how long will your afternoon commute be?

My most variable commute has been living in the Washington, DC, area. The important factors in my commute time were these:

- The time I left the office
- The day of the week
- The season (was it summertime?)

Unless you work from home, your commute probably has similar key variables and perhaps a few more that were not important in my commute. Another prediction you make each day is what the weather will be. My children constantly want to wear shorts and T-shirts later into the fall season than social convention dictates. This year after the daily discussion about what the temperature might be during lunch, recess, PE, and the walk to and from school, my wife and I made a firm rule that after Veteran's Day (November 11), pants and long-sleeve shirts were required. Looking at average temperatures, it would have been reasonable to assume that the temperature would not rise to the point of "needing" to wear shorts, but looking at averages is sometimes not a good prediction technique. In the first week of December, while it was 30 degrees in Los Angeles, it was 75 degrees in Cary, North Carolina.

This short story illustrates a number of key points about predictive modeling:

- Sometimes models are wrong.
- The farther your time horizon, the more uncertainty there is.
- Averages (or averaging techniques) do not predict extreme values.

George Box was a pioneer and influential statistician (also the son-in-law of Sir R. A. Fischer) who taught at the University of Wisconsin. In 1987, he published a book titled *Empirical Model Building and Response Surfaces*. In it he makes two statements now famous in the statistical community related to the quality of predictive models. He says: "Remember that all models are wrong; the practical question is how wrong do they have to be to not be useful." Later in the same text he says: "Essentially, all models are wrong but some are useful." These points are important to keep in mind as you use increasing amounts of data to make more specific predictions. Logic and reason should not

be ignored because of a model result. Following a developed methodology and using clear processes will help reduce error and allow you to improve processes when things go wrong.

Building on the weather analogy, it is clear that for a stable process, predicting something immediate is more certain than farther out. The weather prediction for tomorrow is more likely to be accurate than the one for five or ten days into the future. The same is true for other stable processes like the stock market, real estate home prices, or gasoline. In statistical terms, this is referred to as a prediction interval, and it becomes larger the farther time horizon you predict for.

This chapter explains a methodology for building predictive models that has been used by thousands and thousands of data miners. After the methodology is developed, you are presented with information about the types of predictive models and their commonality and distinction.

A METHODOLOGY FOR BUILDING MODELS

The process of building models has been developed and refined by many practitioners over many years. Here is a simple, proven approach to building successful and profitable models.

1. **Prepare the data.** This step has to be completed before any meaningful exploration or analysis can take place.

In larger organizations, this is done by a separate team and perhaps even a different business unit. Regardless of how your data is prepared, either through a formal request and accompanying specifications, as I did at the U.S. Census Bureau, or if you have permission to query directly from the enterprise data warehouse (EDW), this step is essential to a successful model.

Investment in understanding the data preparation process within your organization often pays benefits in the long run. As you need access to increasingly larger and more granular data, knowledge of what data exists and how it can be combined to other data sources will provide insight that was

not possible just a few years ago. If your IT organization is not keeping more data for longer and at finer levels, then you are behind the trend and at risk for becoming irrelevant in your market.

2. **Perform exploratory data analysis.** This is the step where you understand your data and begin to gain intuition about relationships between variables. This exploration is best done with a domain expert, if you do not have that expertise. Fantastic data miners will discover relationships and trends through complicated exploration. They will, with great excitement, present these findings to a domain expert who will politely reply that those things have been known for some time. You cannot give a complete and thorough analysis and recommendation in today's business climate without domain expertise to complement analytical skill.

The tools used for exploratory data analysis have changed in the past few years. In the late 1990s, this was done using mostly tabular output by means of SQL or a scripting language. There were correlation matrix and static graphical output to be pored over. The work was often slow, and programming skills were a requirement. In the last few years, the graphical tools have improved dramatically. There are now many commercial products from SAS, IBM, and SAP, as well as smaller vendors like QlikTech, and Tableau, to name a few, in the space of business visualization. These products are able to load data for visual exploration, generally through a browser-based interface, and provide a highly interactive experience in exploring data. This technology has been proven to work with billions of observations, assuming sufficient hardware resources are available.

The exploration of data is never complete. There are always more ways to look at the data and interactions and relationships to consider, so the principle of sufficiency and the law of diminishing returns need to be observed. The law of diminishing returns comes from the field of economics and states that adding one more unit of effort (time in our case) will yield less increased value per unit for each successive

unit of effort you put into the task. In this case, the insight and knowledge you gain between hours 15 and 16 on data exploration is most likely less than the insight you gained between hours 2 and 3. The principle of sufficiency acknowledges the law of diminishing returns and sets a threshold on productivity loss. Stated in common language, this is: Know when to stop exploration. The software development methodology has moved to incorporate this idea through agile processes of learning a little, getting started, and continuous improvement.

3. **Build your first model.** The key for this step is to realize up front that the successful model-building process will involve many iterations. During some projects, the Thomas Edison quote that “I have not failed. I’ve just found 10,000 ways that won’t work” will seem very apt. Until you build the first model, you are not able to accurately evaluate what the potential impact of the model will be. Build this model quickly with a method you are most comfortable with. (I frequently use a decision tree.) Building the first model helps to cement the criteria for success and set appropriate expectations for the people who will use the model predictions. Human nature is to be optimistic. We think products will sell better than they do, jobs will be completed without complications, and so on. Building the first model is a reality check for future performance and expectations. This first model is, by default, the champion model.
4. **Iteratively build models.** This phase is where the majority of time should be spent. This step is a feedback loop where you will build a model (the challenger) and then compare it to the champion model using some objective criteria that defines the best model. If the challenger is better than the champion model, then evaluate if the challenger model satisfies the project objectives. If the project objectives are not met or the champion is not displaced then build another model. Often there is not a concrete model evaluation to determine when to stop but rather a time window that forces the project to end. Say you are contracted to provide a list of customers for a marketing

campaign. The campaign has a deadline for providing the customer list for next Tuesday, so model building will continue until that point in time.

sEMMA

sEMMA is a data mining methodology, created by SAS, that focuses on logical organization of the model development phase of data mining projects. It describes the process one must go through to capture insight and knowledge from their data. The acronym sEMMA—sample, explore, modify, model, assess—refers to the core process of conducting data mining. Beginning with a statistically representative sample of your data, sEMMA makes it easy to apply exploratory statistical and visualization techniques, select and transform the most significant predictive variables, model the variables to predict outcomes, and confirm a model's accuracy.

Before examining each stage of sEMMA, let us address a common misconception. A common misunderstanding is to refer to sEMMA as a data mining methodology. sEMMA is not a data mining methodology but rather a logical organization of the functional tool set of SAS Enterprise Miner for carrying out the core tasks of data mining. Enterprise Miner can be used as part of any iterative data mining methodology adopted by the client. Naturally, steps such as formulating a well-defined business or research problem and assembling quality representative data sources are critical to the overall success of any data mining project. sEMMA is focused on the model development aspects of data mining:

Sample (optional) your data by extracting a portion of a large data set. This must be big enough to contain the significant information, yet small enough to manipulate quickly. For optimal cost and performance, SAS Institute advocates a sampling strategy that applies a reliable, statistically representative sample of large full-detail data sources. Mining a representative sample instead of the whole volume reduces the processing time required to get crucial business information. If general patterns appear in the data as a whole, these will be traceable in a representative sample. If a niche is so tiny that it is not represented in a sample, and yet so

important that it influences the big picture, it can be discovered using summary methods. We also advocate creating partitioned data sets with the Data Partition node:

- Training—used for model fitting.
- Validation—used for assessment and to prevent overfitting.
- Test—used to obtain an honest assessment of how well a model generalizes.

Explore your data by searching for unanticipated trends and anomalies in order to gain understanding and ideas. Exploration helps refine the discovery process. If visual exploration does not reveal clear trends, you can explore the data through statistical techniques including factor analysis, correspondence analysis, and clustering. For example, in data mining for a direct mail campaign, clustering might reveal groups of customers with distinct ordering patterns. Knowing these patterns creates opportunities for personalized mailings or promotions.

Modify your data by creating, selecting, and transforming the variables to focus the model selection process. Based on your discoveries in the exploration phase, you may need to manipulate your data to include information, such as the grouping of customers and significant subgroups, or to introduce new variables. You may also need to look for outliers and reduce the number of variables to narrow them down to the most significant ones. You may also need to modify data when the “mined” data change either due to new data becoming available or newly discovered data errors. Because data mining is a dynamic, iterative process, you can update data mining methods or models when new information is available.

Model your data by allowing the software to search automatically for a combination of data that reliably predicts a desired outcome. Modeling techniques in data mining include neural networks, tree-based models, logistic models, and other statistical models—such as time series analysis, memory-based reasoning, and principal components. Each type of model has particular strengths and is appropriate within specific data mining situations depending on the data. For example, neural networks are very good at fitting highly complex nonlinear relationships.

Assess your data by evaluating the usefulness and reliability of the findings from the data mining process, and estimate how well it performs. A common means of assessing a model is to apply it to a portion of data set aside during the sampling stage. If the model is valid, it should work for this reserved sample as well as for the sample used to construct the model. Similarly, you can test the model against known data. For example, if you know which customers in a file had high retention rates and your model predicts retention, you can check to see whether the model selects these customers accurately. In addition, practical applications of the model, such as partial mailings in a direct mail campaign, help prove its validity.

By assessing the results gained from each stage of the sEMMA process, you can determine how to model new questions raised by the previous results and thus proceed back to the exploration phase for additional refinement of the data.

Once you have developed the champion model using the sEMMA-based mining approach, it then needs to be deployed to score new customer cases. Model deployment is the end result of data mining—the final phase in which the return on investment from the mining process is realized. Enterprise Miner automates the deployment phase by supplying scoring code in SAS, C, Java, and PMML. It not only captures the code for analytic models but also captures the code for pre-processing activities. You can seamlessly score your production data on a different machine and deploy the scoring code in batch or real time on the Web or directly in relational databases. This results in faster implementation and frees you to spend more time evaluating existing models and developing new ones.

sEMMA for the Big Data Era

How is sEMMA methodology impacted in the era of big data? The short answer is that largely it is not. sEMMA is a logical process that can be followed regardless of data size or complexity. However, the ‘s,’ or sample, in sEMMA is less likely to be as critical with the more powerful systems available for mining the data. From my experience in working with big data, very large analytical databases can be addressed using sEMMA.

BINARY CLASSIFICATION

From my experience, binary classification is the most common type of predictive model. Key decision makers for corporations and other organizations must often make critical decisions quickly, requiring a system to arrive at a yes/no decision with confidence. These systems are not designed to do things part way or with some likelihood. They either do them or they do not. This is illustrated very well in the famous “go or no go for launch” checklist that each system’s flight controller must answer. A launch can proceed only after the flight director has run through each system and received the “go for launch” answer. In the business world, many decisions are binary, such as should I extend you credit for this car purchase, or will you respond to this marketing campaign.

With predictive modeling of a binary target, the probability that an event will or will not occur is also very useful information. Let me provide an example to illustrate this point. With a binary target we are creating a black-and-white situation; will it rain tomorrow—yes or no? It will either rain or it will not, but you are very unlikely to see your local meteorologist (or even the National Weather Service) give a chance of rain at 0% or 100% because while it will either rain (100%) or not rain (0%), the estimate is implicitly the degree of confidence we have in our prediction. This confidence estimate is often much more useful than the binary prediction itself. Does your behavior differ if you see a 10% chance of rain versus a 40% chance of rain? Are you willing to leave your windows down or the top off of your convertible? Both of these predictions, because the percentage is less than 50, indicate that it is less likely to rain than not. However, from experience with having wet car seats for my commute home from a day when it was unlikely to rain (30%), I keep my windows up if there is *any* chance of rain because the downside risk is much greater than the potential upside.

An often-seen case in binary predictive modeling is when you are dealing with a “rare event.” What qualifies as a rare event differs depending on the industry and domain, but it universally means that the event happens with extremely low probability and therefore might require some special treatment in the analysis.

Using an example from the sport of golf, let us examine the rare event more closely.¹ The most sought-after event in golf is the hole in one. This is where a golfer takes a tee shot during at least a 9-hole round and makes it in the hole in one shot. If you make a hole in one during some tournaments, you can often win a car, and custom dictates that regardless of when it happens, you buy a round of drinks for everyone in the clubhouse.² To be official, the shot must be witnessed by someone else. The United States Golf Association (USGA) keeps a registry of all holes in one and estimates the odds of hitting a hole in one to be 1 in 33,000. So while the odds of anyone hitting a hole in one is very small (.003%), some people, like Tiger Woods, not only have hit one but hit multiple holes in one. Tiger has 18 recorded holes in one but does not lead in this category. The most holes in one recorded goes to Norman Manley, with 59. To put quantitative terms around the odds of hitting a hole in one, we would say that no one is likely to hit a hole in one. Assuming 1 in 33,000 tee shots results in a hole in one and (after removing long holes) that there are 14 holes you could reasonably make a hole in one on, if you golfed a round every day of the year you would, on average, hit a hole in one every seven years.

Once we have determined the probability of the event occurring for each observation, they are then sorted in descending order from largest probability to smallest probability based on the likelihood the event will occur. Using the example above, if we were marketing commemorative hole-in-one plaques, we would apply the model to all golfers in our database sorting the list from highest probability to lowest probability of making a hole in one. We would then mail an offer letter to all the golfers from the top of the list down until our marketing campaign budget is exhausted or to those who exceeded a certain threshold. This will ensure that we send the offer for a commemorative plaque to the golfers, based on the model, most likely to hit a hole in one and who would be interested in our services.

¹ For full disclosure, I golfed a lot growing up but never hit a hole in one. I'm now in retirement since I haven't played a round of golf since my oldest child was born.

² Did you know that you can purchase hole-in-one insurance? Policies start at few hundred dollars depending on the prize offered during the tournament.

MULTILEVEL CLASSIFICATION

Multilevel or nominal classification is very similar to binary classification with the exception that there are now more than two levels. Nominal classification is an extension of binary classification. There are several examples where nominal classification is common, but for the most part this is the rarest of targets. An example of such a model can be seen in the cell phone industry when looking at customer churn. A company may not only be interested in the binary response of whether an account remains active or not. Instead, it may want to dive deeper and look at a nominal response of voluntary churn (customer chooses to cancel the contract), involuntary churn (e.g., contract was terminated due to delinquent payments), or an active customer.

In many cases, a nominal classification problem arises when an exception case is added to what could be a binary decision. This happens in the case of preventing credit card fraud, for example. When a credit card transaction is initiated, the card issuer has a very short window to accept or decline the transaction. While this could be thought of as a simple binary problem, accept or decline, there are some transactions in which the decision is not that straightforward and may fall into a gray area. A third level of response may be added to indicate that this transaction needs further review before a decision to accept or decline can be made.

The nominal classification problem poses some additional complications from a computational and also reporting perspective. Instead of just computing one probability of event (P) and then taking $1 - P$ to arrive at the probability of nonevent, you need to compute the probability of event #1 (P_1), the probability of event #2 (P_2), and so on until the last level which can be computed using $1 - \sum_{i=1}^{n-1} P_i$. There is also a challenge in computing the misclassification rate. Since there are many choices, the report values must be calibrated to be easily interpreted by the report reader.

INTERVAL PREDICTION

The final type of prediction is interval prediction, which is used when the target level is continuous on the number line. Salary is an example of a prediction that employers and employees alike would like to make

accurately. The property and casualty insurance industry is an area with many interval prediction models. If you own a car in the United States, you are required to have insurance for it. To get that insurance, you likely requested a quote from several different insurance companies, and each one gave you a slightly different price. That price discrepancy is due to different predictive models and business factors that each insurance company uses. The business factors are the amount of exposure in certain geographic markets or an overall directive to try to gain market share in a certain market either geographic or economic strata.

Companies in the insurance industry generally utilize three different types of interval predictive models including claim frequency, severity, and pure premium. The insurance company will make predictions for each of these models based on its historical data and your specific information including the car make and model, your annual driving mileage, your past history with driving infractions, and so on.

ASSESSMENT OF PREDICTIVE MODELS

One consideration in the process of building predictive models will be to determine which model is best. A model is comprised of all the transformations, imputations, variable selection, variable binning, and so on manipulations that are applied to the data in addition to the chosen algorithm and its associated parameters. The large number of options and combinations makes a brute-force “try everything” method infeasible for any practical data mining problem. So the issue of model assessment plays an important role is selecting the best model. Model assessment, stated simply, is trying to find the best model for your application to the given data. The complexity comes from the term “best.” Just like purchasing a washing machine, there are a number of aspects to consider and not everyone agrees on a single definition of “best.” The common set of model assessment measures are listed and defined next. As an example, consider a local charity that is organizing a food drive. It has a donor mailing list of 125,000 people and data from past campaigns that it will use to train the model. We partition the data so that 100,000 people are in the training partition and 25,000 are in the validation partition. Both partitions have a response rate of 10%. The validation partition will be used for assessment, which is a best practice.

Table 4.1 Decision Matrix for Model Assessment

	Predicted Nonevent	Predicted Event
Nonevent	True negative	False positive
Event	False negative	True positive

Classification

There are a set of assessment measures that are based on the 2×2 decision matrix as shown in Table 4.1.

Classification is a popular method because it is easy to explain, it closely aligns with what most people associate at the “best” model, and it measures the model fit across all values. If the proportion of events and nonevents are not approximately equal, then the values need to be adjusted for making proper decisions. (See Table 4.2.)

Receiver Operating Characteristic

The receiver operating characteristics (ROC) are calculated for all points and displayed graphically for interpretation. The axis of the ROC plot are the Sensitivity and 1-Specificity, which were calculated from the classification rates.

Table 4.2 Formulas to Calculate Different Classification Measures

Measure	Formula
Classification rate (accuracy)	$\frac{\text{true negative} + \text{true positive}}{\text{total observations}} \times 100$
Misclassification rate	$(1 - \frac{\text{true negative} + \text{true positive}}{\text{total observations}}) \times 100$
Sensitivity (true positive rate)	$\frac{\text{true positive}}{\text{true positive} + \text{false negative}} \times 100$
Specificity (true negative rate)	$\frac{\text{true negative}}{\text{false positive} + \text{true negative}} \times 100$
1-Specificity (false positive rate)	$\frac{\text{false positive}}{\text{false positive} + \text{true negative}} \times 100$

Lift

Lift is the ratio of percentage of correct responders to percentage of baseline response. To calculate lift, it must be accompanied by a percentile in the data. This is commonly referred to as a depth of file, and usually the first or second decile is chosen. For the food drive example, if we compute the lift at the first decile (10% of the data), the baseline (or random) model should have 2,500 responders to the campaign so that in the first decile there will be 250 responders ($2,500 \times .1$). Our model is good; it captures 300 responders in the first decile so the lift at the first decile is 1.2 ($300/2500=12\%$ captured response/10% baseline response). I prefer to use cumulative lift for my model assessment because it is monotonic and in practice campaigns will sort a list by the likelihood to respond and then market until a natural break is observed or the budget for the campaign is exhausted.

Gain

Gain is very similar to lift except 1 is subtracted from the value $\frac{\% \text{ of model events}}{\% \text{ of baseline events}} - 1$ for a given decile. For the food drive example, the gain would be 0.2 at the first decile.

Akaike's Information Criterion

Akaike's information criterion (AIC) is a statistical measure of the goodness of fit for a particular model. It maximizes the expression $-2(LL + k)$ where

k = number of estimated parameters (for linear regression the number of terms in the mode)

LL = maximized value of the log-likelihood function for the given model

The smaller the AIC, the better the model fits the data. Because of the k term, the smaller number of model parameters are favored. AIC values can be negative, but I do not remember ever encountering that in a real-world data mining problem.

Bayesian Information Criterion

The Bayesian information criterion (BIC) is a statistical measure similar to AIC but that maximizes the expression $-2LL + k \times \ln(n)$ where:

n = number of observations (or sample size)

k = number of estimated parameters (for linear regression the number of terms in the model)

LL = maximized value of the log-likelihood function for the given model

For any two given models, the one with the lower BIC is better. This is because the number of terms in the model is smaller, the variables in the model better explain the variation of the target, or both.

Kolmogorov-Smirnov

The Kolmogorov-Smirnov (KS) statistic shows the point of maximum separation of the model sensitivity and the baseline on the ROC curve.

Model assessment is ideally done on a holdout partition that is representative of the data but was not used in the model-building phase. This holdout partition (often called a validation or test partition) is essential to measure how well your model will generalize to new incoming data.

CHAPTER 5

Common Predictive Modeling Techniques

The key part of data mining and in my opinion the most fun is after the data has been prepared and the data scientist is able to conduct a “model tournament.” Because of the complexity and size of most modern data mining problems, the best practice is to try a number of different modeling techniques or algorithms and a number of attempts within a particular algorithm using different settings or parameters. The reason for so many trials is that there is an element of brute force that needs to be exerted to arrive at the best answer. The number of iterations needed to achieve the best model can vary widely depending on the domain and the specific modeling challenge.

*The data analysis for this chapter was generated using SAS Enterprise Miner software, Version 13.1 of the SAS System for Windows. Copyright © 2013 SAS Institute Inc. SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc., Cary, NC, USA.

It is important to note that while this chapter is focused on predictive modeling or supervised learning, many of these techniques can be used in unsupervised approaches to identify the hidden structure of a set of data. Supervised learning has a target variable, and the methods attempt to correctly classify or predict as many of the observations as possible. Supervised learning has clear measures and assessment as to the model quality. In contrast, unsupervised learning lacks a target and therefore strong objectiveness as to measure model quality.

Each of the following sections covers a different modeling technique and has the general progression of brief history on the technique, a simple example or story to illustrate how the method can be used followed by a high-level mathematical approach to the method and finally a references section for you to get additional details.

Finally, in this chapter, many of the most popular predictive modeling techniques are discussed with examples, graphics, and some mathematical details. This is designed to give you an overview of the techniques and their uses.

RFM

RFM, or recency, frequency, and monetary modeling, is a very simple but effective way to rank customers. It looks at how long since a customer's last visit (recency), how often they purchase (frequency), or how much they spend (monetary). This method is desirable because it does not require sophisticated software or knowledge of "fancy" algorithms. The method can be easily explained to all levels of management and domain knowledge. The method is most common in database marketing or direct marketing. A basic approach is to take each metric and divide them into segments. These might be data driven, such as the deciles of how much households spend, or they might be rule driven, such as recency of 0–90 days, 91–180 days, and 181+ days, to align with common accounting practices. If we take the simple example of just two levels in each dimension, then we will have eight segments as shown in Table 5.1. The number of segments can be generally computed using:

$$\text{Levels}_{\text{Recency}} \times \text{Levels}_{\text{Frequency}} \times \text{Levels}_{\text{Monetary}}$$

Table 5.1 RFM Segmentation with Two Groups per Variable

Segment	Recency	Frequency	Monetary
1	2	2	2
2	1	2	2
3	2	1	2
4	1	1	2
5	2	2	1
6	1	2	1
7	2	1	1
8	1	1	1

All customers will fall into one of these segments. By segmenting the customers, you can design a specific campaign or treatment to help maximize profits. Looking at the segments from Table 5.1, it is clear that segment 1 is the most valuable set of customers. They have recently made a purchase, they frequently purchase, and they spend an above average amount compared to your average customer.

Using the segment assignments from Table 5.1, it is clear that segment 1 is the most valuable and segment 8 is the least. However, there is less clarity in the other segments, and it is common in practice to profile the segments and perhaps combine some if there are not sufficient differences.

The typical RFM segmentation is to use five bins, one for each quintile, per factor for a total of 125 segments. It is advisable not to use the RFM method for prediction because while it is accurate under many circumstances, it is only a historical measure. It has no ability to account for changes in macroeconomic factors or individual life transitions, such as birth, death, divorce, or retirement, which would drastically modify your marketing strategy.

A couple of differences between RFM and the other techniques in this section are that RFM does not produce any type of score code that can be applied to new observations. This is in contrast to the other methods, such as decision trees, where a main output is to deploy the model into a production system that will very quickly score incoming records or could be used to score a large customer table. In the case of RFM, the whole process must be repeated when new data is added to

the data set. Another distinction is because of the lack of score code and repeating of the process, scores cannot be compared across data sets. If you look at the 555 group (which denotes the best of the best customers in a typical setting) from a consumer retail scenario for December sales, that will likely be a very different set of customers with different profiles and different qualifications than the customers from May sales. The number of sales, their amounts, and their frequency are very different for the Christmas holiday season versus May.

There are two main strategies to using the RFM scores, and they vary depending on if you feel that recency is more important than frequency, which is more important than monetary. If that is the case, then a simple sorting of the data will allow targeting the most important clients, and you can continue sequentially through the list until your campaign budget is exhausted. The rationale for believing that recency is most important is that the longer it takes for your customers to return to your business, the less likely they are to ever return. It is easier to affect the customer behavior of how often they come or how much they spend, but a greater time since the last contact increases that chance that you have already lost that high-value customer. Alternatively, if you do not feel that constraint applies perhaps because of a variety of reasons, then a score of 345, 534, or 453 are all roughly equal and so you can sum the individual digits of the score (12) in this case and sort the list by the summed score beginning with the customers receiving a 15 and moving down the list until the campaign budget is exhausted. A number of variations on this method can also be applied to add some level of sophistication, such as secondary sorts or multiplying the individual scores (R,F, or M) by a weighting factor to reflect the relative importance of each.

A last consideration that is needed for RFM and for many of the modeling techniques is the data preparation before the technique can be applied. In a retail setting, the sales database is full of transactions that have relatively few details: typically just date, time, account number, store ID, and amount. There will typically be a transaction ID that will link to a different table in the enterprise data warehouse that includes a full list of the shopping cart. But all of this data is transactional, and RFM needs customer-level data so an extract, transform, and

load (ETL) process (which was discussed earlier in Part One) to create that data set is summarized at the customer level.

This method is largely based on the Pareto principle, which is commonly explained as the 80/20 rule. It describes that 80% of your revenue comes from just 20% of your customers. The name comes from an observation by the Italian economist Vilfredo Pareto in 1906 that 80% of Italy's land was owned by only 20% of the population. This also led to the Pareto distribution, which is a power distribution and has applications in many observable phenomena in the social and actuarial fields, among others.

REGRESSION

Regression analysis is likely the first predictive modeling method you learned as a practitioner during your academic studies or the most common modeling method for your analytics group. Regression concepts were first published in the early 1800s by Adrien-Marie Legendre and Carl Gauss. Legendre was born into a wealthy French family and contributed to a number of advances in the fields of mathematics and statistics. Gauss, in contrast, was born to a poor family in Germany. Gauss was a child math prodigy but throughout his life he was reluctant to publish any work that he felt was not above criticism. After his death, many of his personal journals were discovered that detailed his ideas and thoughts. Those ideas were significant advances in the field of mathematics. It is estimated that if Gauss had been more aggressive in publishing his work, he could have advanced the field of mathematics more than 50 years. Those two brilliant men along with Sir R.A. Fisher in the early twentieth century describe the method of regression.

The term “regression” was coined by Francis Galton (who was a cousin of Charles Darwin) to describe the biologic process of extreme values moving toward a population mean. A common example of this is children of tall (above-average height) parents “regressing” toward the mean of the population and being shorter than their parents; conversely children of short (below-average height) parents being taller than their parents. Prior to Galton, the technique was referred to as “least squares.”

From my experience, regression is the most dominant force in driving business decisions today. Regression analysis has many useful

characteristics; one is the easy interpretation of results. Regression concepts are widely understood, and the methodology is well developed such that a well-tuned regression model by a skilled practitioner can outperform many algorithms that are gaining popularity from the machine learning discipline.

Basic Example of Ordinary Least Squares

As an example, let us look at the relationship of weight and height for a class of 19 middle school students. There is an expected and natural relationship, a positive correlation, between weight and height where we expect students who weigh more to be taller. Let us first show a plot of weight and height to see what that relationship looks like (Figure 5.1).

As you can see in Figure 5.1, there is a trend that the more you weigh, the taller you are, but there is in no way a perfect relationship.

To begin, let us consider the question of interest. We want to find a line that best predicts the height of a student knowing his/her weight, as in the next equation.

$$H_i = \beta_0 + W_i\beta_1$$

where

H_i = height of an individual student

W_i = weight of an individual student

β_0 = intercept

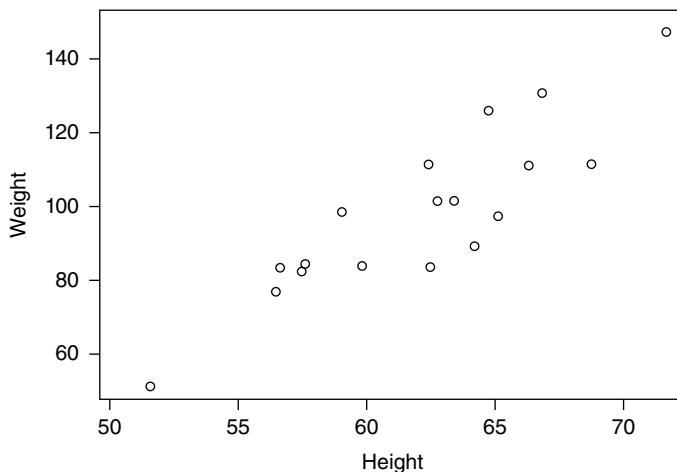


Figure 5.1 Scatter Plot for Height and Weight for Class

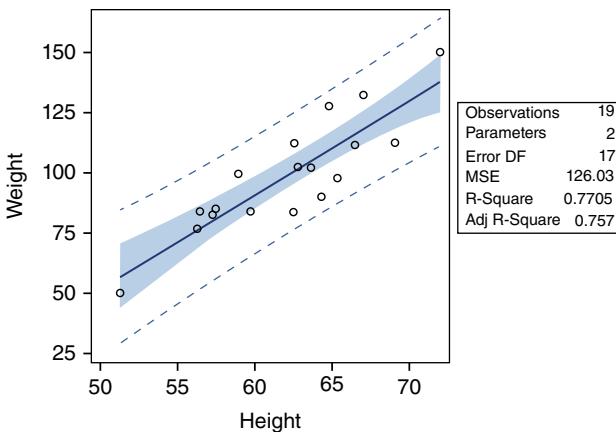


Figure 5.2 Fit Plot for Weight

In regression analysis, the best line is the one that minimizes the sum of error across all data points. The error for an observation is measured as the squared distance between the proposed line, which is the predicted value, and the open circle, which is the actual value. Why is the vertical distance between the point and the line squared? Because with points above the line and other points below the line, a simple adding of the distance between the points will result in an infinite number of lines that will give us a summed value of 0. But using the square of the distance there is a single best line that gives us the minimum sum of the squares. This statistic is called the sum of squares, and this is a very useful measure of how well the regression line fits the data. You can see the best regression line in Figure 5.2.

In addition to the calculation of the sum of squares to find the best-fitting line, a number of diagnostics need to be considered as part of performing a regression analysis.

The residual plot is the single most important plot to review before accepting the prediction from a regression model. A proper residual plot will look like a random scatter of points (like the one shown in Figure 5.3).

If you see a funnel shape where the points start close to the reference line near 0 on the X-axis and then spread out as they move to

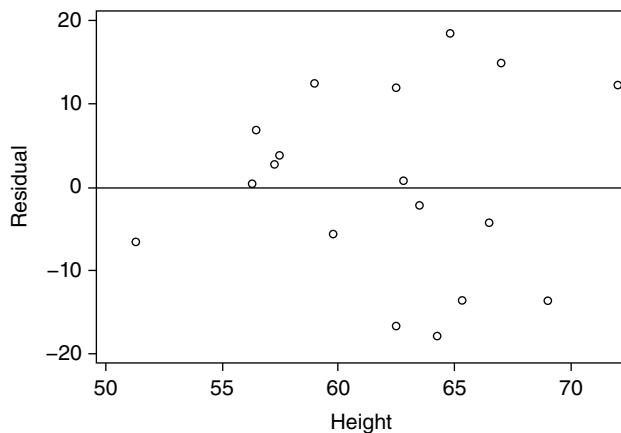


Figure 5.3 Residuals for Weight

the right, variable transformations are needed for your input variables. (See Figure 5.4.)

If the points have consistent spread but form a curve in either direction (a happy face or sad face), such as Figure 5.5, then you need to transform your input variables. The Box-Cox transformation method can often be used to find a transformation to improve the fit of your model in this scenario.

The diagnostic plots shown in Figure 5.6 present a great deal of information. Here is a brief explanation for several of the key diagnostic plots.

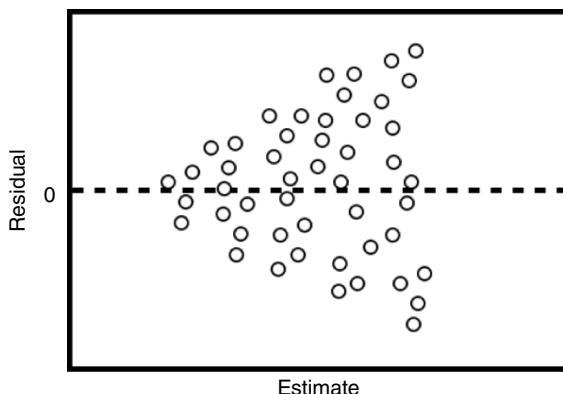


Figure 5.4 Plot Residuals in a Funnel Shape

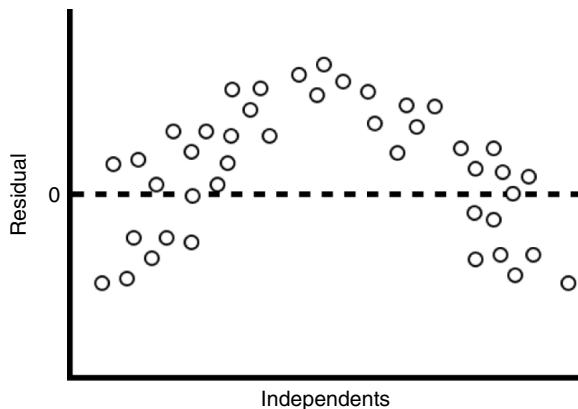


Figure 5.5 Residuals in an Arc

Beginning with the upper left and moving across the first row in Figure 5.6:

The residual plot has already been discussed, but additionally it shows the fit of the errors to the data. It is an essential plot to assess model fit and a residual plot where the points appear random is the ideal plot.

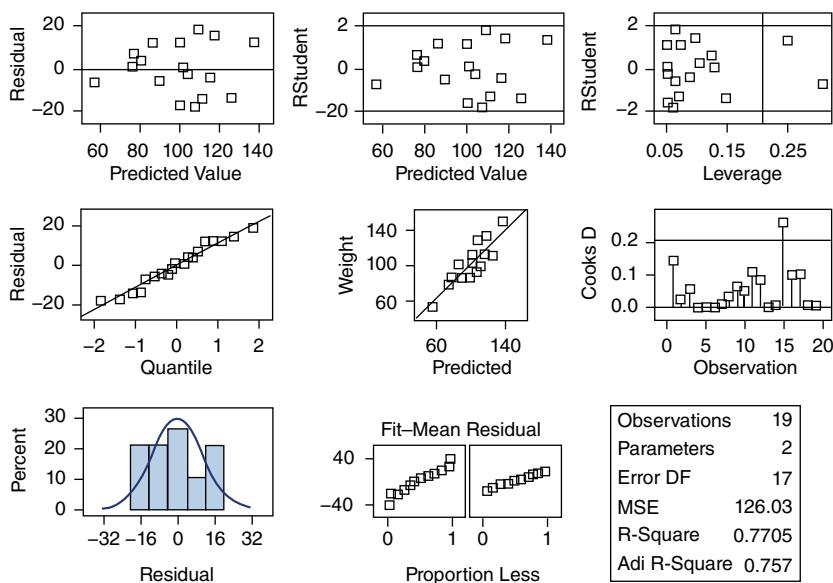


Figure 5.6 Fit Diagnostics for Weight

The studentized¹ residuals show the same pattern as the residuals, but the variables have been first studentized, which is analogous to normalizing or standardizing. For this example, because there is only one variable, the residual and studentized residual plots look identical except for the Y-axis range. If there are multiple independent variables, then the process of studentizing them will ensure that variables with larger values and larger variances do not dominate the model fitting, which could reduce model quality.

The leverage plot examines the influence that each individual point has on the regression line. Since the technique is trying to minimize the distance of the sum of squares that are at the extremes of the range of observations (either very small or very large), they have more influence on the line than the points in the center. You can think of this middle of the range like the seesaw from the playground from your youth. The farther from the center of the seesaw, the less force (weight) is needed to lift the other side. This is similar to your data. A small set of points or even a single point can alter the regression line substantially, as shown in Figure 5.7. The single outlier moves the regression line away from the very good fit of the line to the data if the outlier is not present. You can deal with outliers in several ways. One option is to remove the observation after some investigation to ensure there is some reasonable justification. Because often this is not possible or cost effective in working with big data, another option is to use a robust regression technique and winsorize² the observations.

Quantile plot (Figure 5.6 second row left plot) is used to assess quantiles of two distributions to each other; in this case the comparison is that the residuals are being compared to a normal distribution. The more points are on the line $X = Y$, the more normal the residuals are. If the points made an “S” shape,

¹ William Gosset, who wrote several papers under the pseudonym Student, developed this method.

² Winsorize is a statistical transformation to limit extreme values. It was invented by a biostatistician named Charles Winsor.

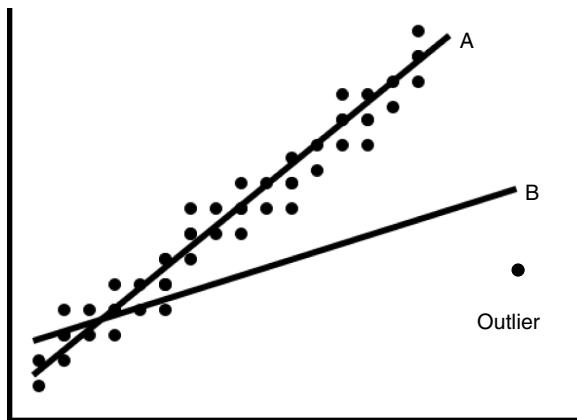


Figure 5.7 Effect of Outlier on Regression Line

that would indicate more probability in the tails. If only one side moves away from the line, that likely indicates outliers or skewness of the distribution.

Predicted versus actual plot show how well the regression line predicted the values for a set of observations with known outcomes.

Cook's D plot is also used for outlier detection. By looking at the Cook's D value, you can determine the relative influence each observation had in the fitting of the regression line.

Percentile versus residual is another view to express the same information as the QQ plot.

R-Square (R^2) is a simple transformation of taking the studentized residuals and squaring the terms. This has the productive properties of making all the terms positive and helping to scale them to be more in line with human intuition. The scale is also now conveniently bounded between 0 and 1 as opposed to -1 and 1 .

The power of regression analysis is the ability to describe the relationship between a dependent, label, or target variable³ and a set

³These are all terms to describe the same thing. The discipline of the practitioner largely determines the vocabulary used.

of independent attributes or input variables. Depending on the dependent variable and the type of analysis, many techniques can be deployed to build a regression model.

Assumptions of Regression Models

The assumptions of a linear regression model are important to note. It is also important to understand that in the practice of data mining, some if not all the assumptions are violated to some degree. There are ways to overcome these assumptions including using more advanced regression modeling techniques, which are discussed later in this section.

The standard assumptions for linear regression are:

- There is no measurement error in the observed values of the target and each input variable. Basically it is assumed that the values in the data set are correct.
- The relationship between the dependent variable and all of the independent variables are linear.
- The observations are independent.
- The independent variables are not linear combinations of each other.
- The variance of the dependent variable is constant (homoscedasticity)

For logistic regression, there is an additional assumption:

- The true conditional probabilities are a logistic function of the independent variables.

Additional Regression Techniques

Here is a list of advanced regression techniques that are useful in data mining with large data. Observations will be more extreme and relationships between variables more complex.

- Adaptive regression fits multivariate adaptive regression spline models. This is a nonparametric regression technique that combines both regression splines and model selection methods.

This regression technique can use both continuous and discrete inputs.

- Generalized additive models (GAM) are nonparametric, meaning that they do not require the assumption of linear predictors. GAMs produce smooth functions of the regression variables. This technique is most useful in fitting additive models for non-normal data. The technique can use both continuous and discrete inputs.
- Locally estimated scatterplot smoothing (LOESS) uses a local regression method to fit nonparametric models. This method is good for modeling data from an unknown distribution, and outliers are likely to be present in the data.
- Orthogonal regression uses the Gentleman-Givens computational method to perform regression on ill-conditioned data. The Gentleman-Givens method often provides more accurate parameter estimates and nonspecialized regression methods.
- Partial least squares regression, in addition to finding the minimal prediction error for a response variable, also balances that with the variation of the predictor or input variables. This balance is often achieved by using linear combinations of variables like those found in principle components.
- Proportional hazards regression uses the Cox models to fit regression models to survival data.
- Quantile regression is used to model the effects of covariates on the conditional quantiles of a response variable.
- Robust regression uses Huber M estimation and high breakdown value estimation to fit models. This regression method is good for detecting outliers and providing stable predictions when the data contains outliers.

Applications in the Big Data Era

Regression analysis is a fundamental predictive modeling method to a student of statistics, but it is potentially a new and innovative idea for those who come from other disciplines, such as computer science. I was recently watching a talk on machine learning in preparation

for a presentation. The speaker, a well-educated practitioner from a prestigious university, said: “We use a method called logistic regression for those of you that are not familiar with it. . .” This was a reminder to me that the area of data mining is very broad and people approach problems from different perspectives.

From my experience working with customers even in the big data era, regression is still the dominant method for predictive modeling. I think that can be attributed to many factors. The first is easy-to-explain output. The parameter estimates that result from a regression analysis are very approachable to people from a wide range of quantitative skill and business knowledge. Second, the results can be used to determine the key factors in the model (sometimes called driver variables or key performance indicators [KPIs]). Third, the methods are parallelized (or should be); they can be used on nearly any size problem.

Regression is still a very actively researched area, and modern techniques of lasso, robust regression, and ridge regression have all advanced the field based on principles developed over 200 years ago. I expect this field to continue to advance and create new inventions as researchers and practitioners alike look for new ways for regression to provide insight to the relationships within data. In this era where data volumes continue to increase and data is combined in more ways and at lower levels of granularity, the need for advanced regression techniques will remain in demand for a long time. Any software package you are considering for dealing with big data should do regression. If not, that is a big red flag.

GENERALIZED LINEAR MODELS

In 1972, statisticians John Nelder and Robert Wedderburn, who worked together at the Rothamsted Experimental Station in England, defined generalized linear models (GLMs) as a unified framework for probit⁴ models in pursuit of chemical dosage tolerance, contingency tables, ordinary least squares (OLS) regression, and many more. This generalized

⁴ Probit is a statistical term that describes a type of regression model that only has two possible values: for example, male or female.

description of models lifted the restrictions, which were prohibitive for certain types of problems, and offered the flexibility to accommodate response variables that are nonnormally distributed, have a mean with a restricted range, and/or have nonconstant variance—which are all violations of the assumptions for OLS regression. GLMs are characterized by three components:

1. The probability distribution of the response variable Y_i (the random component) can be any distribution from the exponential family.⁵
2. The linear model which includes the explanatory variables and the model parameters, $x'_i \beta$ (the systematic component).
3. The link function which describes the relationship between the systematic and random components.

More specifically, the usual linear component of a linear model, $x'_i \beta$, now links to the expected value μ_i of the response variable Y_i through a function g such that $g(\mu_i) = x'_i \beta$. The link function g can be any monotonic, differentiable function.⁶ For ordinary linear regression, the link is simply the identity and the response distribution is normal.

A logistic regression model, another example of a GLM, uses the logit link function for the Bernoulli-distributed response variable; in turn, this gives a probability estimate between 0 and 1 for the modeled outcome.⁷ This model would be used when the model is trying to predict churn, attrition, or any other binary event where the event of interest is not as important as the likelihood that the event will occur.

⁵ The exponential family of distribution include normal, gamma, beta, and chi-squared among many others.

⁶ “Monotonic” is a property where the values of an independent variable never decrease. “Differentiable” means that there is a derivative that exists at each point along the function.

⁷ The Bernoulli distribution is named after Jacob Bernoulli, a Swiss seventeenth-century mathematician. The distribution is one discrete trial (like a coin toss) with probability of success p and probability of failure $q = 1 - p$.

Counts can be modeled in a GLM using a Poisson or negative binomial distribution and a log link function. Examples of count models could include the number of claims someone makes against an insurance policy, the number of cars passing through a toll booth, the number of items that will be returned to a store after purchase, or any other counting-type process where the occurrence of one event is independent from another occurrence.

Example of a Probit GLM

I, like most people, do not like bugs in my home. Because North Carolina has lots of bugs, to keep them out of my house, I spray periodically the perimeter of my home and the trunks of my trees with pesticide. I have a friend whom I met when I first moved to North Carolina who worked for a national pest consumer control company. His job was to research the effectiveness of certain chemical compounds with killing pests. Consider this common example for modeling whether an insect dies from a dosage of a toxic chemical in a pesticide. Several different dosages were applied to a sample of insects, and data was collected on how many insects lived and died at the particular dosages. The plot in Figure 5.8 shows the proportion of insects that died; the Y-axis represents the proportion that died, at certain dosage amounts represented on the X-axis. While the relationship between these two variables does appear somewhat

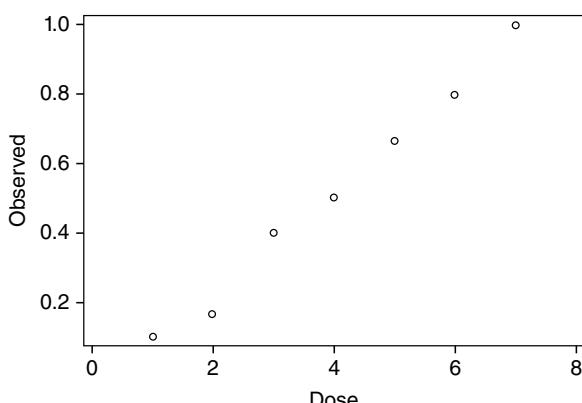


Figure 5.8 Proportion of Response by Dose

linear, a linear regression model is not suitable in this case; we are trying to predict the probability of dying at a particular dosage, which needs to be between 0 and 1 because we cannot have less than 0% of the insects die or more than 100% of the insects live, and OLS is not bounded. This probability of death P at a given dosage or log dosage, x , is equal to the probability that the insect's tolerance for the chemical T is less than x . The tolerance for all subjects (or in this case insects) at a fixed dosage or log dosage is often assumed to have a normal distribution with mean μ and standard deviation σ . Thus we have:

$$P = \Pr(T < x) = \Phi[(x - \mu)/\sigma]$$

The function Φ is the standard normal cumulative distribution function, and its inverse Φ^{-1} is the probit function. Since Φ^{-1} is a legitimate link function, this model can be expressed in the form of a GLM as:

$$\Phi^{-1}(P) = \beta_0 + \beta_1 x$$

where

$$\beta_0 = -\mu/\sigma$$

$$\beta_1 = 1/\sigma$$

One of the motivations for Nelder and Wedderburn's paper was to introduce a process for fitting GLMs because an exact estimate of the parameters was not available most of the time.⁸ They presented an iterative weighted least squares process that can be used to obtain the parameter estimates of β . The predicted probability of death \hat{P} can be estimated from the MLEs of β as:

$$\hat{P} = \Phi(\hat{\beta}_0 + \hat{\beta}_1 x)$$

Figure 5.9 shows the probit model curve (the solid line) representing the predicted probabilities that is fit to the data collected.

Unlike the regression example, in most cases for GLMs, we do not get a measure like R-Square to determine how well the model fits. Instead of R-Square, there are a set of measures to evaluate model fit.

⁸ A closed-form solution for the maximum likelihood estimates of the parameters is not available in most cases (an exception being OLS regression).

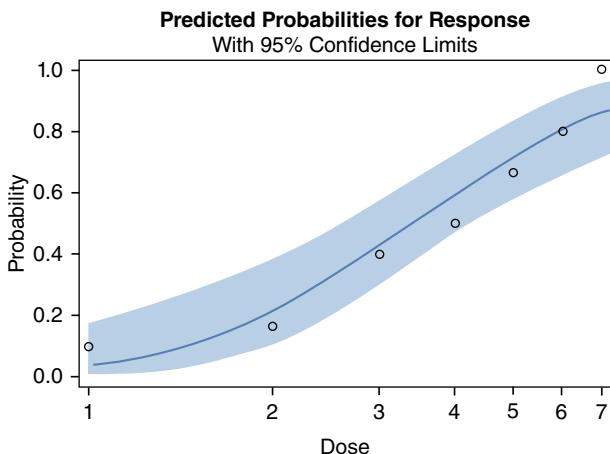


Figure 5.9 Predicted Probabilities for Response

A simple way to classify between the two groups of model fit measures: tests where small values indicate bad model fit and small values indicate good model fit.

For small values are bad:

- Pearson's chi-square statistic
- deviance⁹ or scaled deviance

Significant p -values¹⁰ for these two test statistics indicate a lack of fit.

For small values are good:

- Akaike information criterion (AIC)
- Corrected AIC (AICC)
- Bayesian information criterion (BIC)

These fit measures can also be used for evaluating the goodness of a model fit, with smaller values indicating a better fit. As for a regression analysis discussed in the Regression section, examining the residual plots can give insight into the reason for a lack of fit, as can other diagnostics mentioned, such as Cook's D for outlier detection.

⁹ Also called the likelihood ratio chi-square statistic.

¹⁰ p -values are the probability of obtaining a result at least this extreme when the null hypothesis is true.

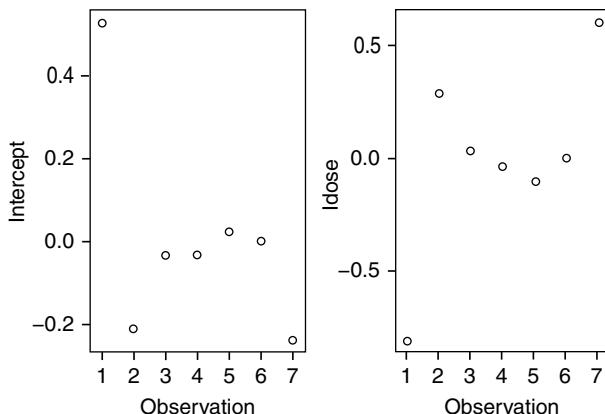


Figure 5.10 Unstandardized DFBETA Plots for Response

The DFBETA and standardized DFBETA statistics are other measures of influence of individual observations on the model fit. An example of the unstandardized DFBETA measure plotted against observation number for each of the two estimated parameters is shown in Figure 5.10. These statistics and plots can help identify observations that could be outliers and are adversely affecting the fit of the model to the data.

Applications in the Big Data Era

A main application of GLMs with respect to big data is in the insurance industry for building ratemaking models. Ratemaking is the determination of insurance premiums based on risk characteristics (that are captured in rating variables); GLMs have become the standard for fitting ratemaking models due to their ability to accommodate all rating variables simultaneously, to remove noise or random effects, to provide diagnostics for the models fit, and to allow for interactions between rating variables. The response variables modeled for ratemaking applications are typically claim counts (the number of claims an individual files) with a Poisson distribution, claim amounts with a gamma distribution, or pure premium models that use the Tweedie distribution. These distributions are all members of the exponential family, and the log link function typically

relates the response to a linear combination of rating variables, allowing these models to be formulated as GLMs. One large insurance company based in the United States uses GLMs to build ratemaking models on data that contains 150 million policies and 70 degrees of freedom.

NEURAL NETWORKS

Artificial neural networks were developed in the last half of the twentieth century. They were inspired by a simple mathematical model of a biological neuron—the core structure in the brain that enables learning in animals. In this model, described by neurophysiologist Warren McCulloch and logician Walter Pitts in 1943, a neuron “fires” (i.e., its output becomes 1) when the sum of its excitatory inputs is greater than a threshold value and no inhibitory inputs are 1. McCulloch and Pitts showed that networks of these model neurons could compute the value of most complex logical equations (equations where the variables only have the values 0 or 1). Figure 5.11 is a representation of their original concept.

Starting in the late 1950s, Frank Rosenblatt used a network of neurons similar to McCulloch-Pitts neurons, which he called a perceptron, to correctly identify digitized pictures of letters by weighting the inputs to the neurons and then optimizing those weights using the Perceptron learning algorithm. The Perceptron learning algorithm starts out with a random set of weights and then iteratively modifies those weights based on an error function of the difference between the

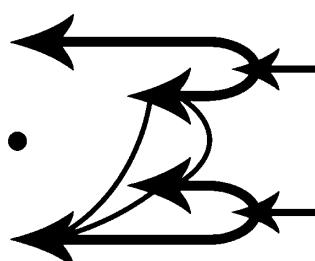


Figure 5.11 Original Drawing of a Neural Network

Source: McCulloch, W. S. and Pitts, W. H. (1943). "A logical calculus of the ideas immanent in nervous activity." *Bulletin of Mathematical Biophysics*, 5, 115-133.

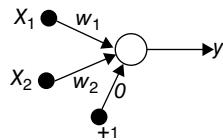


Figure 5.12 Simplified Perception Diagram

actual output of the neuron and the desired output of the neuron (1 if this neuron represents the letter that was digitized and 0 otherwise) for each neuron in the network. This type of training method is now known as supervised learning because it adjusts the weights of the network by looking at a series of examples where the desired output of the network is known. Later, when presented with a set of inputs where the correct output is not known, it is hoped that the “trained” network will output the correct result. (See Figure 5.12.)

Unfortunately, a perceptron can be trained only to classify patterns belonging to linearly separable classes—classes where a hyperplane in the input space can properly separate all patterns, as shown in Figure 5.13.

The lack of ability to model nonlinear relationships in data was pointed out in the book *Perceptrons* by Marvin Minsky and Seymour Papert in 1969. This book became controversial because its analysis of the limitations of perceptrons was interpreted to mean that these limitations would prevent even extended neural networks from solving more complicated, real-world problems. This proved to be false, but this interpretation significantly slowed research into neural networks.

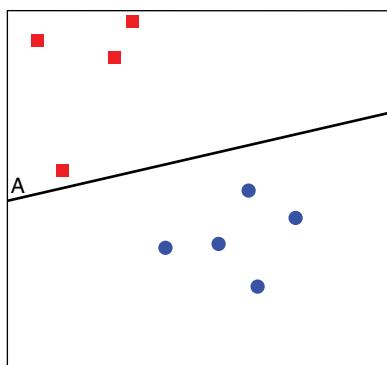


Figure 5.13 Data Properly Separated by a Hyperplane

for more than a decade as research into artificial intelligence shifted away from connectionist algorithms (which represented information as weights and connections) toward symbolic algorithms (which represented information as logical propositions and used logical inference rules to deduce results). An interesting analysis of the controversy over perceptrons is presented by Jordan B. Pollack in his 1989 review of the third edition of *Perceptrons*, “No Harm Intended.”

The breakthrough for neural networks came in the latter half of the 1980s after differentiable squashing activation functions (e.g., logistic, tanh, etc.) replaced the nondifferentiable step activation function used in perceptrons. Besides enabling neural networks to model arbitrarily complex functions, the differentiable activation functions enabled the use of the backpropagation method for computing the gradient of the error function with respect to the weights, which in turn enabled the use of the gradient descent algorithm to compute the optimal weights. This method was described in a 1986 paper titled “Learning Representations by Back-Propagating Errors” by David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams.

Finally, in 1989, in reply to the *Perceptrons* book, the paper “Multilayer Feedforward Networks are Universal Approximators” by Kurt Hornik, Maxwell Stinchcombe, and Halbert White, proved that a three-layer (input, hidden, output) neural network, where the hidden layer consisted of neurons with a squashing activation function, could “approximate any function of interest to any desired degree of accuracy.”

As we sit here in early 2014, neural networks are being used in a wide variety of fields to model everything from credit card fraud, where best to drill an oil well, what stocks to invest in, how to find faces in pictures, whether a person is likely to respond to an ad, whether a person is likely to contract a disease, and whether a drug is likely to cure it.

Despite some of the hype around neural networks, where they are described as “learning” as a human would, they actually represent a straightforward nonlinear regression problem.

In a nonlinear regression problem, a nonlinear function of the input variables is fit by optimizing the weights of a nonlinear function $y(W, X)$

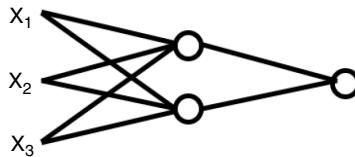


Figure 5.14 Standard Three Layer Neural Network

where

W = weight vector to be computed

X = vector of known input variables

y = known output

A neural network is simply a particular type of nonlinear function, and what is usually seen as a graphical representation is simply a diagram of that function.

The most widely used neural networks are three-layer (input, hidden, and output), feedforward networks using squashing activation functions for the neurons in the hidden layer. Recently neural networks of five or more hidden layers have found to be useful for feature detection in images and in natural language processing.

Figure 5.14 represents the output function $y(Z, W, X) = o(Z \cdot h(W \cdot X))$

where

vector X = input variables

vector W = weights between the input layer and the hidden layer neurons

function h = activation function of the hidden layer neurons

vector Z = weights between the hidden layer neurons and the output layer neurons

function o = activation function of the output layer neurons

Networks like these, generalized to include multiple hidden layers, are referred to as multilayer perceptrons (MLPs). However, the term “neural network” has been applied to a variety of other architectures, such as autoencoders, convolutional neural networks, radial basis function networks (RBF), and self-organizing maps (SOM). The rest of this chapter will focus on multilayer perceptrons.

Multilayer perceptrons are typically used for two types of problems, which usually determines the number of neurons in the output layer, the activation function used by the output layer and the error

function used to accumulate the differences between the network output and the desired output.

Regression problems are those where you are trying to predict a numerical value in response to the given input vector (like predicting the age of a person given their weight, height, and sex). Since you are trying to predict a single number, the output layer only contains a single neuron. The identity output activation function is often used for the output layer neuron and the square of the difference between the network output and the desired output is usually used for the error function.

Classification problems are those where you are trying to predict whether a given input vector belongs to one of several classes (like predicting the species of a plant given several of its measurements). Unlike regression problems, where there is a single neuron in the output layer, the output layer usually has one neuron per possible class. The softmax activation function, in combination with the cross-entropy error function, is often used for the output layer neurons because the output value can then be interpreted as the probability that a given input vector belongs to the class represented by each output neuron.

Figure 5.15 represents the two output functions

$$y_1(Z_1, W, X) = o(Z_1 \cdot h(W \cdot X)) \text{, and } y_2(Z_2, W, X) = o(Z_2 \cdot h(W \cdot X))$$

where

vector X = input variables

vector W = weights between the input layer and the hidden layer neurons

function h = activation function of the hidden layer neurons

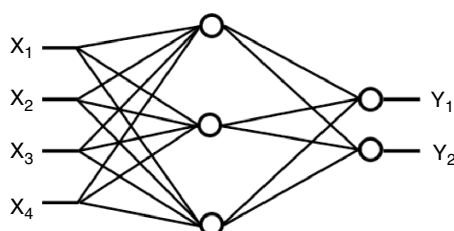


Figure 5.15 Two Output Functions

vector Z_1 = weights between the hidden layer neurons and the first output neuron

vector Z_2 = weight between the hidden layer neurons and the second output neuron

function o = activation function of the output layer neurons

In this example, each output neuron could represent one of two possible classes that each input vector can belong to. The output could be interpreted as: an input vector X belongs to class1 if output y_1 was greater than y_2 .

In order to train a neural network, you need a method to compute the weights that will minimize a highly nonlinear objective function for a set of training examples. Remember, the objective function is not the network output—it represents the network error—the total difference between what the network should be outputting and what it is outputting over all training examples.

The task of finding the minimum value of a nonlinear differentiable function is usually accomplished by using some form of either the gradient descent method or Newton's method. Both of these methods require that you repeatedly calculate the gradient of the objective function. In the context of neural networks, the gradient is the vector of partial derivatives of the objective function with respect to each weight. The value of each partial derivative in this vector represents the amount the objective function changes for a small change in weight. The key contribution of the backpropagation algorithm was a straightforward method to efficiently calculate this gradient.

The gradient descent method uses the notion that if you start at randomly chosen initial points (a randomly chosen initial set of weights) and slowly move in the direction of the negative gradient of the objective function (in the steepest downhill direction) you will eventually end up at the bottom, or at least at a place where the gradient is zero (a valley). The reason you have to move slowly is so you don't jump over the actual minimum and have to backtrack. The reason the function has to be differentiable is so the gradient exists, which it does for multilayer perceptrons with standard activation functions, but not, for instance, the step activation function. When using the gradient descent method, you have to decide how far along the negative

gradient to move at each step – the step length. One way is to move an amount proportional to the gradient. Here is the formula for the gradient descent method, where

$$W_{n+1} = W_n - \lambda \nabla f(W_n)$$

where

W_n = current vector of weights

λ = learning rate

∇f = objective function to be minimized

While moving in the direction of the negative gradient provides the steepest descent from the current position, it is not necessarily the direction that gets to the ultimate minimum the fastest.

Newton's method, when used for minimization, uses the first derivative, the gradient, and the second derivative, the curvature, to find the minimum value of a function, and is called a second order method. At each step, it computes a simple quadratic function with the same gradient and curvature as the current objective function, and then finds new weights that would minimize the quadratic function. These new weights will also result in a reduced objective function. It then repeats the process until the gradient of the objective function becomes close to zero. Here is the formula for Newton's method:

$$W_{n+1} = W_n - H^{-1} \nabla f(W_n)$$

where

H = Hessian¹¹

Second order methods generally converge to a minimum with far fewer iterations than first order methods like gradient descent, but computation of the inverse Hessian is expensive time-wise, on the order of n^3 operations for a function of n variables (weights). In order to ameliorate this problem, methods that iteratively estimate the inverse Hessian were developed, called Quasi-Newton methods. The most popular of these is probably the BFGS method (Broyden–Fletcher–Goldfarb–Shanno). Like the nonlinear conjugate gradient method, this method first computes a search direction (by multiplying the estimated

¹¹ The Hessian is a matrix of second partial derivatives of the objective function with respect to each pair of weights

inverse Hessian by the current gradient), and then uses an inexact line search along that direction to find a new set of weights, which result in a reduced gradient. However, it still takes n^2 storage locations to store the Hessian. To resolve this last issue, the Limited Memory BFGS (L-BFGS) method was developed.

Some key considerations for working with neural networks are:

- **Starting values.** When the weights of a neural network are near zero, the model is approximated by a linear regression model. Through many iterations the neural network model becomes increasingly nonlinear as the weights change and help define the unobserved nonlinear relationships. If the starting values are too close to zero, then the algorithm fails to converge to the true values. Likewise, starting with large weights usually leads to poor solutions. The challenge is to select starting values that are close enough to zero but do not get stuck there.
- **Overfitting.** One of the biggest worries when using historical data to develop a model that predicts future results is overfitting. Overfitting means the model has been trained to fit a specific set of training observations very well, but does not generalize well—does not do well at predicting the outcomes of observations not used for training. Unlike linear or logistic regression, where you’re using historical data to compute just a few model parameters, with neural networks, depending on the number of input variables and number of hidden neurons, you could be fitting your past data with hundreds or even thousands of model parameters (weights). Because neural networks are universal approximators, by adding enough hidden neurons, and running the training algorithm for enough iterations, they will eventually be able to predict the outcomes of the training observations perfectly. But what the neural network is learning during training is the noise in the specific training data set as well as the underlying process that generated the data. When presented with new data, the same noise will not be there.
- **Validation data to adjust for overfitting.** One good answer is to use cross validation during training. You split your training data into a training subset and a validation subset. The

validation subset should contain at least 25% of your available data. You must make sure that both subsets have the same distribution of data. You only give the training algorithm the training subset. After each iteration of the training algorithm, you compute the prediction error on both the training and validation subsets, using the current set of weights computed by the training algorithm. What you will typically see is that the error on the training subset (training error) and the error on the validation subset (validation error) both start downward together as the training algorithm does its work. Then, at some point, validation error starts to go up again, even as the training error continues lower. At the point the validation error increases you have achieved your best generalized model.

- **Input scaling.** The scale of the input variables affects the initial weights. It is a best practice to scale the inputs to a mean of 0 and a standard deviation of 1 before beginning to build your model. This will ensure that all input variables are getting fair and equal treatment. The scaling of the input variables also helps identify good starting points.
- **Assurance of global minimum.** Because the goal is to minimize the error is nonconvex (meaning that in hyperspace, any line between two points is not always a valid solution; think of Swiss cheese, where the air bubbles are invalid solutions), it is possible to get stuck at a local minimum and never reach the overall minimum. The ability to find the global minimum is based largely on the starting values.

Basic Example of Neural Networks

Sir Ronald Fisher gathered data on three types of *Iris*. He collected measurements of their petal length, petal width, sepal length, and sepal width. Using these four interval measures, the objective is to correctly classify from these measurements the three species of *Iris*: *virginica*, *versicolor*, and *setosa*.

The first step is to partition the data to make sure that the neural network model does not overfit so we take a stratified sample and split the data 70% for training and 30% for validation. The setup of the

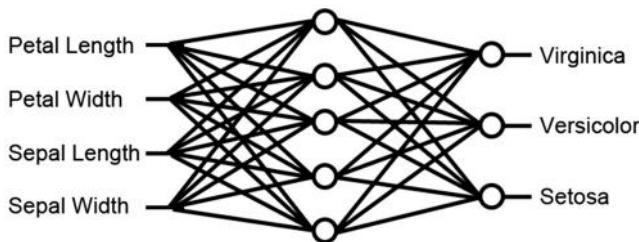


Figure 5.16 Neural Network Diagram for *Iris* Data

network is four input neurons, five hidden neurons, and three target neurons as represented in Figure 5.16.

The initial weights between the input neurons and hidden neurons are randomly initialized. The error term of the objective function is shown in Table 5.2 for 16 iterations.

Table 5.2 Average Error Function (AEF) of Training and Validation for Neural Network for *Iris* Data

Iteration	AEF Training	AEF Validation
0	0.443202	0.443281
1	0.231005	0.244472
2	0.149747	0.178680
3	0.131696	0.158875
4	0.063039	0.070936
5	0.054594	0.075916
6	0.046256	0.063721
7	0.026284	0.028231
8	0.021823	0.020531
9	0.020630	0.018305
10	0.016621	0.015093
11	0.013766	0.014756
12	0.013696	0.014804
13	0.013559	0.014864
14	0.012273	0.014848
15	0.009394	0.014826
16	0.008228	0.014819

Table 5.3 Comparison of Target and Outcome of *Iris* Validation Data Set

Target	Outcome	Count
SETOSA	SETOSA	15
VERSICOLOR	VERSICOLOR	14
VERSICOLOR	VIRGINICA	1
VIRGINICA	VIRGINICA	15

You can see from the table that the training error continues to shrink as the weights are adjusted and tuned. The validation error, however, begins to rise after the eleventh iteration (shown as bold in Table 5.2). This is because the neural network has now overfit the model to the training data and lost its general predictive power. The model that should be used to classify the *Iris* is the one produced at the eleventh iteration.

The neural network was able to classify all but one observation in the validation data correctly as shown in the third row of Table 5.3 where a virginica iris was mistaken classified as a versicolor iris. This error corresponds to a 2.2% misclassification rate for the validation data.

Recently, the term “deep learning” has become widely used to describe machine learning methods that can learn more abstract concepts, primarily in the areas of image recognition and text processing, than methods previously available. For instance, neural networks with three or more hidden layers have proven to do quite well at tasks such as recognizing handwritten digits (see Figure 5.17), or selecting images with dogs in them.

Training these types of neural networks has proven quite a challenge. The standard backpropagation algorithm begins to fail on networks with large numbers of input variables (each pixel of an image is typically a single input variable) and many hidden layers because the gradient of the total error with respect to the weights in the initial hidden layers becomes extremely small and noisy. This is known as the vanishing gradient problem. In addition, the large number of input variables, which results in very large numbers of weights that must be computed, causes the standard optimization process to be very slow.

In 2006, Geoffrey Hinton and R. R. Salakhutdinov described how they successfully trained seven hidden layer autoencoders (neural networks whose output is trained to be as close to the input as possible) on images using Restricted Boltzmann Machines. In 2010, one of Hinton’s

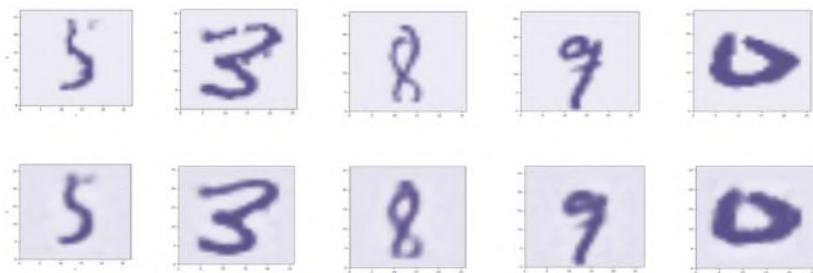


Figure 5.17 Improvement of Digit Recognition Using Deep Learning

graduate students, James Martens, developed a new Quasi-Newton method he called Hessian Free Learning,¹¹ more accurately described as Truncated Newton Hessian Free Learning, which he claims can also be used to train neural networks with many hidden layers.

DECISION AND REGRESSION TREES

Tree-based methods have long been popular in predictive modeling. Their popularity is owed, in no small part, to their simplicity and predictive power along with the small number of assumptions.

Tree-based methods of modeling have been around since the 1960s, with popular methods including CART, C4.5, and CHAID as common implementations. Decision trees were first presented by Leo Breiman in his 1984 book *Classification and Regression Trees*. The initial CART algorithm has been improved and extended in a number of ways through the years, but the overall principles remain.

Begin with a set of training data and look at the input variables and their corresponding levels to determine the possible split points. Each split point is then evaluated to determine which split is “best.” “Best” in this case is creating the purest set of subgroups. The process is then repeated with each subgroup until a stopping condition has been met. The main distinction between decision trees and regression trees is that decision trees assign a category from a finite or nominal list of choices, such as gender or color. Regression trees, in contrast, predict a continuous value for each region, such as income or age.

Consider the problem of trying to predict the gender of students from a general elective course at a university. You could build a decision

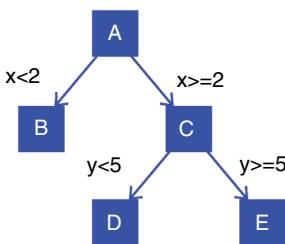


Figure 5.18 Basic Decision Tree

tree that predicts gender by dividing a representative population by height, weight, birth date, last name, or first name. Last name and birth date should have no predictive power. Height and weight, however, would likely be able to predict gender in some cases (the very tall are likely to be male and the very short likely to be female). More predictive still would be first names, as there are fewer names than heights or weights that are gender-neutral. A variable that is hard to capture but probably very predictive is hair length. This example shows that with the right attribute, it is often possible to get near-perfect separation with a single variable. In most real-world problems, that single “magic” variable does not exist, is too costly to collect, or is something that could not be known beforehand for the purposes of scoring new cases.¹²

Decision and regression trees both recursively partition the input space one variable at a time. Often a single split is used at each level, as it is equivalent to multiple splits.

Figure 5.18 shows a tree where the original data set (region A) is first split on variable y by whether it is less than 2 (region B) or greater than or equal to 2 (C). Region C is then further split into two regions, D ($y < 5$) and E ($y \geq 5$).

These same regions are shown in Figure 5.19 as partitioned input space. The partitioned input space only shows the leaves of a tree (a leaf is a nodes that have no further splits i.e. terminal node). The leaves that been split are called branches. The branches in Figure 5.19 are A and C.

Building a tree happens in two phases. First, a large tree is created by recursively splitting the input data set using a metric to determine

¹² Do not forget that the purpose of building a predictive model is to anticipate a behavior or attribute.

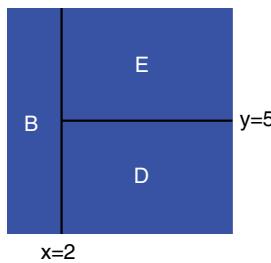


Figure 5.19 Basic Decision Tree Shown as a Partition Region

the best split. Growth continues until a stopping point is reached conditions might include: insufficient data for additional splits, additional splits do not improve model fit, or tree has reached a specified depth. After the tree has been built, it is often overfit (which makes it poor for prediction). That is, it matches the input data too well—it matches all of the small fluctuations that are present in the input data set but not in the general population. Thus, the overlarge tree is then pruned back until it is predictive in general, not just for the particular input data set. Pruning is particularly effective if a second, validation data set is used to determine how well the tree performs on general data. This comes at the cost of taking some data that would have been used to grow the tree. The C4.5 splitting method attempts to prune based on estimated error rates instead of using a separate data set.

There are a large number of methods to determine the “goodness” of a split. A few of the main ones are discussed here—misclassification rate, entropy, Gini, Chi-square, and variance. The interested reader is encouraged to consult a survey book, such as *Data Mining with Decision Trees* by Lior Rokach and Oded Maimon. As they mention, no one “goodness” method will be the best in every scenario, so there is benefit in being familiar with a number of them.

Consider the *Iris* data that was used as an example in the Neural Networks section. After the data is portioned using the same 70% for training and 30% for validation, all the split points from the four input variables are evaluated to see which gives the purest split. As you can see in the figure on the next page, if the petal width is less than 7.5 millimeters (mm), then the *Iris* is a *setosa*. If the petal width is greater than 7.5 mm or missing, then it is either *versicolor* or *virginica*. A further split

is needed to create a distinction among all three groups. All of the variables and their split points are then considered for how to refine the group for purity. In this case the petal width is again the best variable to create pure groups. This time the split point is having a petal width of less than 7.5 mm or missing (and greater than 7.5 mm because those observations have already been removed). About 90% of those *Iris* are *versicolor*. On the other side of the branch it is almost 100% *virginica*. The tree is not further split, even though there is a few flowers wrongly classified (4% of the validation); this is because there is no split that improves the classification and does not violate a stopping rule.

The English rules for classification of future *Iris* flowers appears as follows:

```
*-----*
Node = 2
*-----*
if Petal Width in mm. < 7.5
then
Tree Node Identifier = 2
Number of Observations = 34
Predicted: species=setosa = 1.00
Predicted: species=versicolor = 0.00
Predicted: species=virginica = 0.00

*-----*
Node = 4
*-----*
if Petal Width in mm. < 17.5 AND Petal Width in mm. >= 7.5 or
MISSING
then
Tree Node Identifier = 4
Number of Observations = 36
Predicted: species=setosa = 0.00
Predicted: species=versicolor = 0.92
Predicted: species=virginica = 0.08

*-----*
Node = 5
*-----*
if Petal Width in mm. >= 17.5
then
Tree Node Identifier = 5
Number of Observations = 33
Predicted: species=setosa = 0.00
Predicted: species=versicolor = 0.03
Predicted: species=virginica = 0.97
```

Misclassification rate is based on the nominal (i.e., unordered set of values, such as *Cincinnati*, *Chattanooga*, and *Spokane*) target predicted value, usually the most common value. Misclassification asks how many observations in each region (N_r observations in each region, with N_R observations in the combined region) were mispredicted (N_m), and thus is

$$M = \sum_r^R \frac{N_r}{N_R} \frac{N_m}{N_r} = \frac{1}{N_R} \sum_r^R N_m$$

Variance, used for regression trees, is similar to misclassification rate if the predicted value for the region (\hat{t}_r) is simply the average value of the target in that region. The variance is the usual variance formula, summed over the observations in each of the regions separately. If t_i is the target value in observation i :

$$V = \sum_r^R \frac{N_r}{N_R} \frac{1}{N_r} \sum_{i \in r} (t_i - \hat{t}_r)^2 = \frac{1}{N_R} \sum_r^R \sum_{i \in r} (t_i - \hat{t}_r)^2$$

Entropy, used with nominal targets, uses the “information” of a split. The goodness metric is then the gain of information (decrease of entropy). The entropy for a set of regions with N_t observations that have a value t for the target:

$$E = \sum_r^R \frac{N_r}{N_R} \sum_t^T \frac{N_t}{N_r} \log_2 \left(\frac{N_t}{N_r} \right)$$

Gini, also for nominal targets, is similar:

$$G = \sum_r^R \frac{N_r}{N_R} \sum_t^T \frac{N_t}{N_r} \left(1 - \frac{N_t}{N_r} \right)$$

Methods based on Chi-square, including CHAID, view the split as a two-way contingency table. For instance, Table 5.4 shows a three-region split with two target levels. Each cell in the table is the number of observations within each region with each target value. The p-value

Table 5.4 Observations

	Region A	Region B	Region C
Target value α	97	75	23
Target value β	3	50	175

of the resulting Chi-square sum determines the goodness of the split and also serves as a threshold to end growth.

The finished tree can be used to predict the value of a data set, understand how the data is partitioned, and determine variable importance. The tree structure makes it easy to see what factors were most important to determining the value of the target variable. The more important variables will be split more often and will generally be higher in the tree, where more data is being partitioned. The amount of data that a variable partitions determines how important that variable is. This importance can then be used to guide further analysis, such as regression.

Although a single decision tree is easy to understand, it can be sensitive to input data fluctuations despite the pruning process. For instance, say two variables are different only by 2 out of 1,000 observations and therefore partition the data almost identically. However, both variables do the best job of partitioning the data. Depending on how the data is used, one variable or the other may be selected, and the other will be ignored since it will likely not help partition the data much further. This is advantageous in that it means that correlated variables are compensated for automatically. However, which variable is selected as a split in the tree may change between runs based on very small changes in the data. Additionally, if the fit is not almost perfect, the later splits may be wildly different depending on which variable is chosen, resulting in considerably different trees.

Because of this, in addition to building a single decision tree, there are a number of methods that build upon using many decision trees. Random Forest™ uses bootstrap methods to create an ensemble of trees, one for each bootstrap sample. Additionally, the variables eligible to be used in splitting is randomly varied in order to decorrelate the variables. Once the forest of trees is created, they “vote” to determine the predicted value of input data. Additive methods, including AdaBoost and gradient boosting, use a number of very basic trees (often just a single split) together to predict the value. The sum of all of the trees’ predictions are used to determine the final prediction, using a mapping method such as a logit function for nominal targets. Unlike forest models, the trees in additive models are created in series, in such a way as to correct mispredictions of the previous trees. This is

accomplished through adjusting observation weights to make misclassified observations more important.

SUPPORT VECTOR MACHINES

I remember watching *Sesame Street* nearly every day in the morning before kindergarten. I am sure that I shouted my responses to Big Bird with as much gusto as my six-year-old body could muster. A song used often on the show titled “One of These Things” succinctly describes the purpose of support vector machines (SVMs). The song begins: “One of these things is not like the others ...”

During a *Sesame Street* episode, the audience is shown four items in each quadrant of the screen. Three of the items shown are the same and one is different. Usually the difference is the size or color of the item, depending on what lesson is being taught during that particular episode. This ability to differentiate items into two groups is the basic concept behind SVMs, which are especially useful when the differences are not obvious and there are hundreds or thousands of attributes to consider.

The original concept of SVMs is found in *Estimation of Dependences Based on Empirical Data* by Vapnik, in which the goal was to find optimal hyperplanes for linearly separable classes.¹³ The result was that the optimal hyperplanes depended on only a small subset of the training data observations, which were called support vectors. In general, there is no one SVM algorithm; rather the term “support vector machines” refers to a general way of classifying data. Let us first look into a simple case of this original support vector machine formulation.

In Figure 5.20 we have a training data set of two groups in two dimensions: -1 (dot) and 1 (square). In general, there are many lines that can separate these two classes from each other. Two such lines are illustrated in Figure 5.21.

In addition to these two lines in Figure 5.21, you can see that there are an infinite number of lines that will separate these two groups from each other. With so many solutions that all get the correct answer, it

¹³ A hyperplane is generalization of a two-dimensional plane (a line) that uses $n - 1$ dimensions to separate n dimensions into exactly two groups.

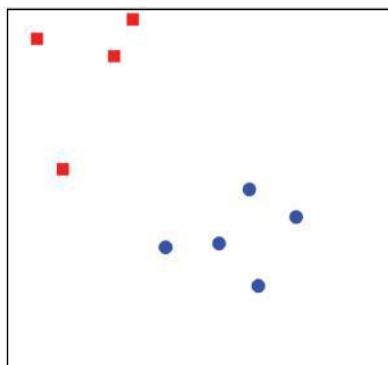


Figure 5.20 Data with Two Groups for Classification

is important to define a way to know which line is the “best” among infinite choices. Since we are interested in separating the two classes, we define the best line to be the one that maximizes the perpendicular distance from any point on either side to the line. In practice this is realized when the distance to the closest point(s) above the line is equal to the distance to the closest point(s) below the line.

You can see with the two example lines that while line A may be far away from any points in group -1 , it is very close to a point in group 1. The opposite is true for line B. Neither of these lines maximizes the distance to the closest point. Let us now look at the best line (Line C) for our example in Figure 5.22.

In this case you can convince yourself visually that the line presented maximizes the distance to any point. We can add perpendicular

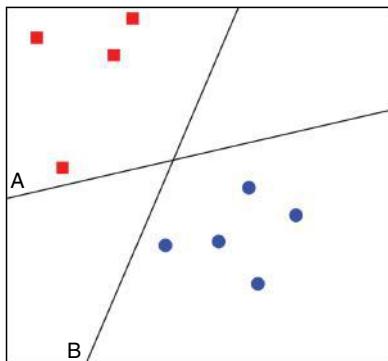


Figure 5.21 Two of Infinite Solutions to Separate into Correct Groups

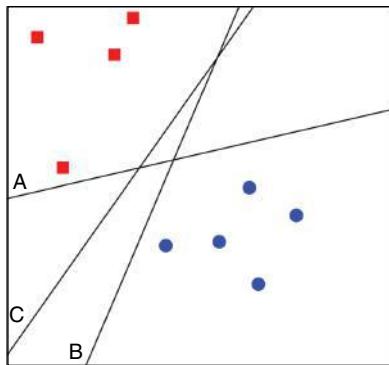


Figure 5.22 Best Separating Line

line segments representing the distance to aid in visualization. The maximization of the splitting plane provides the greatest chance to classify new observations correctly. I use a similar scheme with my children when two children want to share a treat (like a candy bar). One child determines the cut point for the candy bar, and the other gets their choice of segment. This encourages the child who is cutting the candy bar to make the two sections as even as possible. The same is true here: The algorithm is trying to find that exact equal point, normally in a high-dimensional space. (See Figure 5.23.)

The line segment from the square to the separation line has equal length to the line segment from the dot to the separation line. Finally we add two more lines, called the margin lines, to the plot. The margin line is a point of demarcation for classification. (See Figure 5.24.)

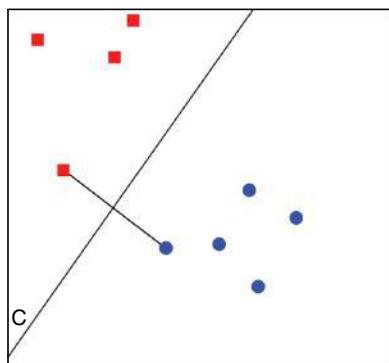


Figure 5.23 Best Separating Line with Perpendicular Reference

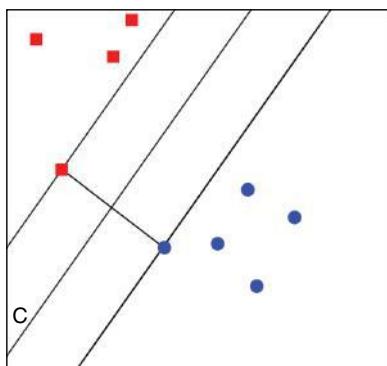


Figure 5.24 Best Separating Line with Margin Lines

As you can see, the margin lines are parallel to the separation line, and the distance from each margin line to the separation line is the same as the distance from the closest point to the separation line. Hence we call the total distance between the two margin lines the margin; often the goal of the SVM will be stated as “to maximize the margin” and make it as big as possible. Any point that lies on either margin line is considered a support vector. In our example we have only two support vectors, though in real-life applications there will likely be many support vectors. For all of you theorists, it is possible to have a problem set where every point in the data is on the margin and needed to classify points, but I have never seen this happen outside of examples designed to create this behavior.

When new data is to be classified, we look to use the separating line. A new observation is classified based on which side of the separating hyperplane it falls, so having the maximum margin will ensure minimum error in the classification of new observations. This is ultimately the question for all classification using support vector machines: Which side of the line is a new observation on? The meat of the theory for SVMs is how to make the line in a way that will minimize error on incoming data. There is also a practical consideration of taking the result of a SVM analysis and applying it to new data. The more support vectors (points on the margin) that are part of the solution, the larger the solution is to store, compute, and represent. The process is essentially to take the points on the margins, let the computer play connect-the-dots then draw a line equidistant between the two

margin lines, then plot each new observation in hyperspace and evaluate into which group it belongs.

The notation for describing a support vector machine is commonly w , the normal vector to the separation line satisfies the equation: $w \times x - b = 0$ for any point x along the separation line. This b is sometimes called the bias, and it is a measure of offset of the separation line from the origin.

In the original SVM formulation, the optimal line is found by solving an optimization problem using Lagrange multipliers subject to

$$\sum_{i=1}^n \alpha_i y_i = 0$$

where

alphas = Lagrange multipliers

y_i = the group (-1 or 1) of the training observation

The result is that vector w is a linear combination of the support vectors: x_i .

$$w = \sum_i \alpha_i y_i x_i$$

When we generalize the concepts from our two-dimensional example into three dimensions, separating and margin lines become separating and margin planes. In higher dimensions we refer to these as hyperplanes.

The discussion so far has assumed that the classes are linearly separable. What happens if our data looks like the plot in Figure 5.25?

As you see in Figure 5.25, there is no line that can completely separate the two classes. In 1995 in *Support-Vector Networks*, Corinna Cortes and Vladimir Vapnik addressed this case with the extension of the SVM to use a “soft margin.” This formulation seeks to minimize the number of errors in the two splits and maximize the margin between these splits.

In order to solve this problem, Cortes and Vapnik introduced slack variables that measure the degree of misclassification for each data point during training. In addition, a penalty function was added to the original formulation. If a linear penalty function is used, this change results in only a slight change in the optimization in that now Lagrange multipliers are restricted by a parameter C :

$$0 \leq \alpha_i \leq C$$

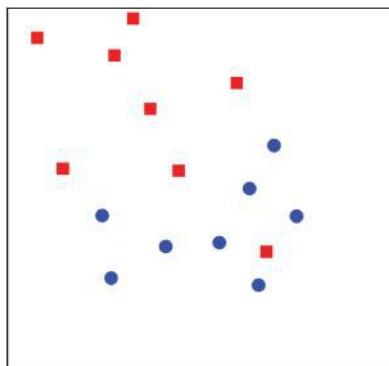


Figure 5.25 Nonlinear Class Separation

and w is still a linear combination of the support vectors x_i . If C is large, then the optimization will favor small error, while if C is small, then the optimization will favor a larger margin. It should be noted that a very small margin can lead to overfitting the data, and so while having a large C will reduce your training error, your validation error may increase.

Further work on support vector machines extended the idea of a linear separation (using a line, plane, or hyperplane) by mapping data into a higher-dimensional space. This is usually done by a method referred to as the kernel trick. It is a trick because the original data points are never explicitly calculated in the higher-dimensional space.

Where we once had the normal vector for the hyperplane satisfy $w \times x - b = 0$ for all points x on the hyperplane, we now have $k(w, x) - b = 0$ for all points x on the higher-dimensional hyperplane. Some common kernels used are:

Polynomial (nonhomogeneous): $k(x_i, x_j) = (x_i \cdot x_j + 1)^d$

Gaussian (aka radial basis function): $k(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$

Sigmoid: $k(x_i, x_j) = \tanh(\kappa x_i \cdot x_j + c)$

Using other kernels can allow for correctly classifying data such as a circle of one class surrounded by another class. However, over tuning kernels has the downside of overtraining the SVM model, especially in the case of polynomial kernels. Just think about using polynomial regression to higher and higher degrees!

Some of the major drawbacks of SVMs are:

- **Binary classification.** While work has been done to extend this to multiclass, SVM models are only natively binary classifiers. This means that the algorithm can be deployed only when a binary response is needed. This makes it unsuited to interval targets such as predicting salary, age, or sales. For nominal targets, the problem will have to be recast as a binary. I used this technique while predicting a 10-level target. I created 10 SVM models, one for 1 or not 1, 2 or not 2, and so on. I then had to calculate a probability and let each model vote on the level it should be assigned. The technique works well but requires a lot of extra processing.
- **Limited interpretability.** Unlike decision trees, which are easy to interpret, or regression models, in which the coefficients have specific meaning, it is often very hard to interpret what the numbers that come out of an SVM actually mean. So using an SVM for predictive modeling requires the trade-off of predictive power with lack of interpretation.

SVM models declare a new observation as either class A or class B without providing a probability (such as 60% class A). While there are ways of mimicking this output, they do not correspond to true probabilities.

There are not widely accepted diagnostic plots to assess the quality of an SVM solution. Instead the quality of the model prediction for an independent validation set or using a K -fold cross-validation strategy is used to ensure that the model is not overfit and will generalize to new observations.

BAYESIAN METHODS NETWORK CLASSIFICATION

Thomas Bayes was a Protestant minister who lived in England in the eighteenth century. He was considered an amateur mathematician and developed his now-famous theorem while trying to prove the existence of God through the inverse probability of causes. The field of probability was very immature at this time, mostly limited to games of chance such as poker, so Bayes developed a thought experiment (the simulations of the 1700s). The experiment began with a ball randomly

placed on a perfectly flat table, unseen by Bayes. Then another ball was randomly placed on the table, and an assistant would answer the question: Is the first ball to the right or left of the second ball? Successive balls would be randomly placed on the table. With more balls on the table, Bayes could visualize with increasing accuracy the position of the first ball as he was told the relative position of each new ball. Names were assigned to his method:

- “Prior” is the probability of the initial belief.
- “Likelihood” is the probability of another hypothesis given new objective data.
- “Posterior” is the probability of the newly revised belief.

Bayes’s ideas have until recently been largely ignored by the statistical field.¹⁴ While the Bayes method has some drawbacks, I firmly believe that people inherently act using Bayesian principles.

To see the Bayes theorem in action, consider the task of driving a car. Who is more likely to be involved in a car accident, a person between the ages of 16 and 24 or 25 and 69? This is not a trick question. As you probably guessed, teenagers are worse drivers and more likely to be involved in an accident. On the surface this is a confusing result; teenagers have a physical advantage with, on average, better eyesight, faster response time, and superior hearing yet the data shows that teenagers are four times more likely to be involved in a car accident than adults. Further, while teenagers make up only 10% of the population, they are involved in 12% of all fatal crashes. Not only are teenagers in more accidents, but the accidents involving teenagers tend to be more serious. One reason you might think of to explain this finding might be that teenagers are more likely to engage in risky behavior, like talking while driving or other multitasking events. This in fact is not true. In a study by State Farm Insurance, 65% of parents admit to talking on the phone while driving and only a small number of teenagers engaged in the same behavior. In another study, Ford found teenagers to be four times as distracted by phone conversations as adults. So what is the key to safe driving that

¹⁴ Statisticians are generally placed in one of two groups, frequentists (the majority) and Bayesians.

teenagers with their superior physical attributes are not able to do? It is the experience and Bayesian behavior we all inherently do. As we drive, we see and observe behavior so that in the future, we are much better at predicting what will happen and we are able to focus on the relevant events of the situation. When coming to a four-way stop and seeing an oncoming car, the experienced driver will be able to instantly (perhaps instinctively) tell that the driver coming from a perpendicular street is approaching with too much speed and will not stop. As we travel along the U.S. interstate system and freeways merge, we can anticipate when cars will attempt to change lanes because we have seen this happen an untold number of times before. Teenagers do not have this knowledge base or depth of experience. Therefore, their ability to predict an outcome is limited because they do not have a well-defined prior.

In Bayesian network classification, there are several different structures each varies in how much freedom there is in drawing relationships between variables.

A Bayesian network is a graphical model that consists of two parts, G, P , where G is a directed acyclic graph (DAG) whose nodes represent random variables and arcs between nodes represent conditional dependency of the random variables and P is a set of conditional probability distributions, one for each node conditional on its parents. The conditional probability distributions for each node may be prior or learned. When building Bayesian networks from prior knowledge alone, the probabilities will be prior (Bayesian). When learning the networks from data, the probabilities will be posterior (learned).

Despite the name, Bayesian networks do not necessarily imply that they rely on Bayesian statistics. Rather, they are so called because they use Bayes' rule for probabilistic inference, as explained later. It is possible to use Bayesian statistics to learn a Bayesian network, but there are also many other techniques that are more closely related to traditional statistical methods. For example, it is common to use frequentist methods to estimate the parameters of the conditional probability distributions.

Based on the topology of the structure, there are different types of networks. Let us start with the naive Bayes network.

Naive Bayes Network

We use the data set shown in Table 5.5 as an example. It is a study of the effects of pain medication treatments on elderly patients who have chronic pain. Treatment A, treatment B, and a placebo are compared. The response variable is whether the patient reported pain or not.

The data set in Table 5.5 contains four input variables: Treatment, Sex, Age, and Duration. The variable Treatment is a nominal variable that has three levels. The gender of the patients is indicated by the nominal variable Sex. The variable Age is the age of the patients, in years, when treatment began. The duration of complaint, in months, before the treatment began is indicated by the variable Duration. For convenience, the two interval variables Age and Duration are binned into two bins, respectively.

The naive Bayes model for this data is shown in Figure 5.26. In the structure, there is a link from the target to every input variable and there are no other links. It is assumed that the input variables are independent (no interactions between variables) of each other given the target.

Table 5.5 Study of Pain Medication on the Elderly

Treatment	Sex	Age	Duration	Pain
Placebo	F	< 70	< 25	No
B	M	≥ 70	< 25	No
Placebo	F	< 70	≥ 25	No
Placebo	M	< 70	≥ 25	Yes
B	F	< 70	≥ 25	No
B	F	≥ 70	< 25	No
A	F	≥ 70	< 25	No
B	F	≥ 70	≥ 25	No
B	F	≥ 70	< 25	Yes
A	M	≥ 70	< 25	Yes
A	F	< 70	≥ 25	No
A	F	< 70	< 25	Yes

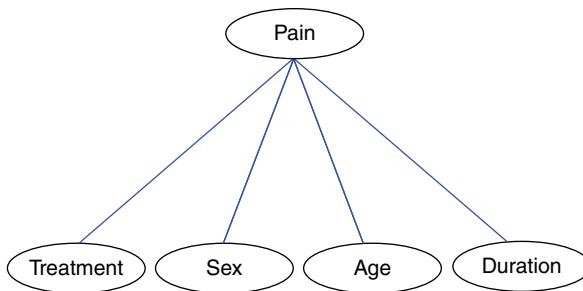


Figure 5.26 Naive Bayes Model for Pain Medication Study

Parameter Learning

Given the structure, we can learn the parameters of the conditional probability distributions, that is, $p(X_i | Y)$, where Y is the target (Pain), and X_i is an input variable (Treatment, Sex, Age, and Duration).

Using the maximum likelihood estimation, we can compute

$$p(X_i | Y) = \frac{\text{freq}(X_i, Y)}{\text{freq}(Y)}$$

where

$\text{freq}(X_i, Y)$ = frequency table of X_i and Y

$\text{freq}(Y)$ = frequency table of Y

The conditional probability distribution for Treatment is shown in Table 5.6.

We can also use a Bayesian method with Beta distribution as the prior to estimate the conditional probability distribution as

$$p(X_i | Y) = \frac{\text{freq}(X_i, Y) + 1}{\text{freq}(Y) + r_i}$$

Table 5.6 Conditional ML Probability Distribution for Treatments Given Target is Pain

Treatment	Pain=Yes	Pain>No
A	2/4	2/8
B	1/4	4/8
Placebo	1/4	2/8

Table 5.7 Conditional Beta Probability Distribution for Treatment

Treatment	Pain=Yes	Pain>No
A	3/7	3/11
B	2/7	5/11
Placebo	2/7	3/11

where

$freq(X_i, Y)$ = frequency table of X_i and Y

$freq(Y)$ = frequency table of Y

r_i = number of levels of X_i

The conditional probability distribution for Treatment is shown in Table 5.7.

The two estimations will be almost the same as the sample size increases. However, there is one advantage with the prior estimation: If an unseen observation has a variable with a new level, the Bayesian method can still estimate the probabilities, while the maximum likelihood estimation will estimate probability to be 0. Because the model is multiplicative, the resulting probability is 0, if any $p(X_i | Y) = 0$. Without the prior, $p(X_i | Y) = 0$ for a new level.

In the above example, we estimate the other conditional probability distributions using the prior shown in Tables 5.8, 5.9, and 5.10.

Table 5.8 Other Conditional Probability Distributions for Sex Given Target Is Pain

Sex	Pain=Yes	Pain>No
F	3/6	8/10
M	3/6	2/10

Table 5.9 Other Conditional Probability Distributions for Age Given Target Is Pain

Age	Pain=Yes	Pain>No
<70	3/6	5/10
≥ 70	3/6	5/10

Table 5.10 Other Conditional Probability Distributions for Duration Given Target Is Pain

Duration	Pain=Yes	Pain>No
<25	4/6	5/10
≥ 25	2/6	5/10

In the naive Bayes method, we can also handle interval variables directly assuming some distribution, typically Gaussian distribution. In this case, we need to estimate the parameters for the Gaussian distributions. Thanks to the independence assumption, the parameter estimation is straightforward using the maximum likelihood or Bayesian method.

This example can be extended to problems of extreme size.

Scoring

To score an observation with Treatment=A, Sex=F, Age<70, and Duration>= 25, we use the Bayes' rule as follows.

$$\begin{aligned}
 p(\text{Pain} = \text{Yes} | \text{Treatment} = A, \text{Sex} = F, \text{Age} < 70, \text{Duration} \geq 25) \\
 &= p(\text{Pain} = \text{Yes}) \times p(\text{Treatment} = A, \text{Sex} = F, \\
 &\quad \text{Age} < 70, \text{Duration} \geq 25 | \text{Pain} = \text{Yes}) \times K \\
 &= p(\text{Pain} = \text{Yes}) \times p(\text{Treatment} = A | \text{Pain} = \text{Yes}) \times \\
 &\quad p(\text{Sex} = F | \text{Pain} = \text{Yes}) \times p(\text{Age} < 70 | \text{Pain} = \text{Yes}) \times \\
 &\quad p(\text{Duration} \geq 25 | \text{Pain} = \text{Yes}) \times K \\
 &= \frac{5}{14} \times \frac{3}{7} \times \frac{3}{6} \times \frac{3}{6} \times \frac{2}{6} \times K \\
 &= 0.013 \times K
 \end{aligned}$$

where

$K = 1 / p(\text{Treatment} = A, \text{Sex} = F, \text{Age} < 70, \text{Duration} \geq 25)$ is a constant

Similarly we compute the conditional probability of

$p(\text{Pain} = \text{Yes} | \text{Treatment} = A, \text{Sex} = F, \text{Age} < 70, \text{Duration} \geq 25)$ as next.
 $p(\text{Pain} = \text{No} | \text{Treatment} = A, \text{Sex} = F, \text{Age} < 70, \text{Duration} \geq 25)$

$$\begin{aligned}
 &= p(\text{Pain} = \text{No}) \times p(\text{Treatment} = A, \text{Sex} = F, \text{Age} \\
 &\quad < 70, \text{Duration} \geq 25 | \text{Pain} = \text{Yes}) \times K \\
 &= p(\text{Pain} = \text{No}) \times p(\text{Treatment} = A | \text{Pain} = \text{No}) \times \\
 &\quad p(\text{Sex} = F | \text{Pain} = \text{No}) \times p(\text{Age} < 70 | \text{Pain} = \text{No}) \times \\
 &\quad p(\text{Duration} \geq 25 | \text{Pain} = \text{No}) \times K \\
 &= \frac{9}{14} \times \frac{3}{11} \times \frac{8}{10} \times \frac{5}{10} \times \frac{5}{10} \times K \\
 &= 0.035 \times K
 \end{aligned}$$

Since

$$p(Pain = No \mid Treatment = A, Sex = F, Age < 70, Duration \geq 25) > p(Pain = Yes \mid Treatment = A, Sex = F, Age < 70, Duration \geq 25),$$

we will score this observation as *Pain = No*.

Learning a Bayesian Network

Bayesian networks can be used for both supervised learning and unsupervised learning. In unsupervised learning, there is no target variable; therefore, the only network structure is DAG. In supervised learning, we need only the variables that are around the target, that is, the parents, the children, and the other parents of the children (spouses). Besides the naive Bayes, there are other types of network structures: tree-augmented naive Bayes (TAN), Bayesian network-augmented naive Bayes (BAN), parent child Bayesian network (PC), and Markov blanket Bayesian network (MB).

These network structures differ in which links are allowed between nodes. We classify them based on the three types of links (from target to input, from input to target, and between the input nodes) and whether to allow spouses of the target. The summary is shown in Table 5.11. Notice that TAN and BAN both allow links from the target to the input and between the input nodes. However, TAN has a tree structure among the input nodes, while BAN allows a Bayesian network (DAG) structure among the input nodes.

To learn these networks, we need not only to determine the parameters of the conditional probability distributions but also to learn the network structures. For TAN, one approach is to find the maximum

Table 5.11 Allowed Links between Nodes by Network Type

Network	Target -> Input	Input -> Target	Input -> Input	Spouse
Naive	Yes	No	No	No
TAN	Yes	No	Yes	No
BAN	Yes	No	Yes	No
PC	Yes	Yes	Yes	No
MB	Yes	Yes	Yes	Yes

spanning tree¹⁵ among the input nodes, while the weight of an edge is the mutual information of the two nodes. For BAN, PC, or MB, a greedy approach is typically used to find a local optimum, because the number of possible structures is superexponential to the number of nodes in the network.

In general, there are two approaches to learning the network structure: one is score based, and the other is constraint based. The score-based approach uses a score function to measure how well a structure fits the training data and tries to find the structure that has the best score. The constraint-based approach uses independence tests to determine the edges and the directions.

In the house alarm¹⁶ example shown in Figure 5.27, there are five variables: an Earthquake has occurred, a Burglary has occurred, an Alarm has sounded, John Calls and Mary Calls the authorities, each representing an event. For short, they are denoted as E, B, A, J, and M respectively. All variables are binary, with T meaning the presence of the event and F the absence of the event.

We see that the Alarm sounds for one of two reasons: either an Earthquake or a Burglary. As a result of an Alarm, the two neighbors, John and Mary, may call the owner of the house that has the Alarm. This is a parent child network structure, if Alarm is the target. The two causes of the Alarm, Earthquake or Burglary, are the parents, and John calling the authorities or Mary calling the authorities are the children to alarm.

The conditional probability distribution for each node is shown beside the node. For example, the probability of Earthquake is 0.02 and Burglary is 0.01. Suppose both Earthquake and Burglary happen, the probability of the Alarm is 0.95. If none of them happen (a false alarm), the Alarm will go off with probability of 0.001. If the Alarm

¹⁵ A spanning tree is a tree that connects all nodes in the network. A maximum spanning tree is a spanning tree that has the maximum total weight of the edges among all possible spanning trees in the network. The weight of an edge is the mutual information of the two connecting nodes.

¹⁶ Judea Pearl in, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, published a very famous example using a fire alarm. This is an adaptation of that example.

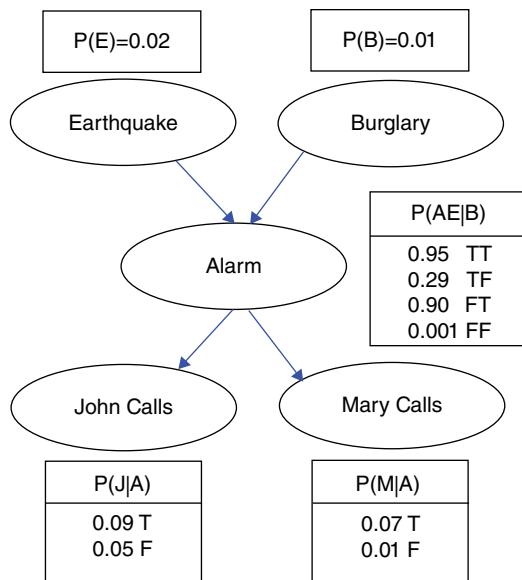


Figure 5.27 House Alarm Example

goes off, John calls with probability of 0.90 and Mary calls with probability of 0.70 respectively.

The network structure together with the conditional probability distributions completely determine the Bayesian network model, that is, the joint probability distribution of the variables is factored to the product of the conditional probability distributions. More formally, let U be the set of variables, X is a variable in U , $pa(X)$ is the set of parents of X , then $p(U) = \prod(X \mid pa(X))$.

Inference in Bayesian Networks

Once we have a Bayesian network (from a prior or learned from data), we can infer the probabilities of interest from the model. For example, in the house alarm problem, we want to determine the probability of the Alarm sounding given observations of the other variables. Suppose we observe that there is a Burglary to the house but no Earthquake, and Mary calls but John does not; what is the probability that the Alarm sounds? We can compute the probability as follows.

$$P(A \mid B = T, E = F, M = T, J = F) = \frac{P(A = T, B = T, E = F, M = T, J = F)}{P(B = T, E = F, M = T, J = F)}$$

Using the factorization of the conditional probability distributions, we compute the join distribution as follows.

$$P(A = T, B = T, E = F, M = T, J = F) = P(B = T) \times P(E = F)^*$$

$$\begin{aligned} & P(A = T \mid B = T, E = F) \times P(M = T \mid A = T) \times P(J = F \mid A = T) \\ &= 0.01 \times 0.98 \times 0.94 \times 0.70 \times 0.1 \\ &= 0.000645 \end{aligned}$$

$$P(B = T, E = F, M = T, J = F) = P(A = T, B = T, E = F, M = T, J = F) +$$

$$\begin{aligned} & P(A = F, B = T, E = F, M = T, J = F) \\ &= 0.000645 + P(B = T) \times P(E = F) \times \\ & \quad P(A = F \mid B = T, E = F) \times P(M = T \mid A = F) \times P(J = F \mid A = F) \\ &= 0.000645 + 0.01 \times 0.98 \times 0.06 \times 0.01 \times 0.95 \\ &= 0.000645 + 0.000006 \\ &= 0.000651 \end{aligned}$$

Finally, we have

$$\begin{aligned} P(A \mid B = T, E = F, M = T, J = F) &= \frac{0.000645}{0.000651} \\ &= 0.99 \end{aligned}$$

Scoring for Supervised Learning

Given a Bayesian network $\langle G, P \rangle$, let $X = (X_1, X_2, \dots, X_n)$ be the set of variables and $Y = X_n$ be the target variable. We can score an observation $(x_1, x_2, \dots, x_{n-1})$ as the level with the maximum probability given the observed input values.

$$\begin{aligned} \operatorname{argmax}_c p(T = c \mid x_1, x_2, \dots, x_{n-1}) &= \frac{P(x_1, x_2, \dots, x_{n-1}, c)}{p(x_1, x_2, \dots, x_{n-1})} \\ &= \frac{\prod_i p(x_i \mid pa(X_i))}{p(x_1, x_2, \dots, x_{n-1})} \end{aligned}$$

where

c = a level of the target

$pa(X_i)$ = the set of parents of X_i

Since the denominator is constant for all levels of the target, it is sufficient to calculate the numerator.

In the house alarm example, given that there is a Burglary to the house but no Earthquake, and Mary calls but John does not, what do we predict for the Alarm? We can compute the probabilities of the two levels (T and F) of Alarm.

$$P(A = T, B = T, E = F, M = T, J = F) = 0.000645$$

$$P(A = F, B = T, E = F, M = T, J = F) = 0.000006$$

Since

$P(A = T, B = T, E = F, M = T, J = F) > P(A = F, B = T, E = F, M = T, J = F)$, we predict that the Alarm will go off.

Bayesian network classification is a very effective technique for predictive modeling and also for variable selection. Many software packages have implementations of these different structures. One key measure to evaluation of different packages is the parallelism and ability to deal with wide data. From my experience, Bayesian network classifiers can perform well on data that is hundreds of thousands of columns and, depending on the implementation, can even work on dense data.

ENSEMBLE METHODS

An ensemble model is the combination of two or more models. Ensembles can be combined in several ways. A common and my personal preference is a voting strategy, where each model votes on the classification of an observation and then in a democratic fashion the classification with the most votes win. Another strategy is to average the predictions from the different methods to make a numeric prediction.

Boosting and bagging are accumulation model techniques that resample the training data to create new models for each sample that is drawn.

Bagging is the most straightforward of ensemble methods. Simply stated, it takes repeated unweighted samples from the data. Each observation can be included in a sample only once (without replacement within a sample), but observations are eligible to be included in all samples (sampling with replacement across samples). With bagging, it

is possible that an observation is included in each sample or an observation is included in none. Under normal conditions, neither of these cases is likely. Because the samples are unweighted, no weight is given to misclassified observations from previous samples. To implement a bagging strategy, you decide how many samples to generate and how large each sample should be. There is a trade-off in these two values. If you take a small percentage of the observations, you will need to take a larger number of samples. I usually take about a 30% sample and take seven samples so that each observation is likely to be selected into a couple of samples. After the samples have been taken and a model fit, those models are then accumulated to provide the final model. This final model is generally more stable than the single model using all the data. Bagging is particularly effective when the underlying model is a decision tree. I have not found benefit to bagging regression or neural network models.

Why does bagging work? In the case of the decision tree model, by taking a sample of the data, you will likely find different splits and a different-looking tree for each sample. None of these tree models is as good as the tree fit on all the data, but when the weaker models are combined, their predictive power is often better than the single model built on the full data. This is because excluding some observations allows weaker relationships to be exploited.

Boosting is very similar to bagging but with the major difference that in boosting, reweighting occurs after each sample model is fit to “boost” the performance of the weak model built on the sampled data. The weights of the misclassified observations are increased so that they are more likely to be selected and therefore be exploited by a subtle relationship in the data. This normally has an improved effect on the overall model when comparing boosting results with bagging results. The trade-off is that boosting must be done in a sequential manner because of the reweighting while bagging can be done in parallel, assuming you have sufficient system resources. In my personal experience, I tend to use boosting for classification models and bagging for interval prediction models.

Random forest is an ensemble tree technique that builds a large number of trees that are weak classifiers and then are used to vote in some manner to build a stable and strong classifier that is better than

the average tree created in the forest. This falls under the axiom that the whole is greater than the sum of its parts. One of the fundamental ideas in random forest is that a subset of the observations and a subset of the variables are taken. This sampling in both dimensions ensures that all variables are considered, not just the dominant few as is usually seen in decision trees. What is the right number of variables to consider in each tree? A good rule of thumb is to use is the square root of the number of candidate variables.

There are a number of tuning parameters to be considered in the training of a random forest model. Here are the high-level dimensions and my recommendations for defaults, given my experience in modeling. Specific problems might require deviations from these recommendations, but I have found success across many industries and problem types.

- Number of trees
- Variables to try
- Number of splits to consider
- Number of observations in each tree

CHAPTER 6

Segmentation

Segmentation is a collection of methods used to divide items, usually people or products, into logical mutually exclusive groups called segments. These segmentation methods include several functions for identifying the best variables to be used in defining the groups, the variables to be used in analyzing these groups, assigning observations to the clusters, analyzing/validating these clusters, and profiling them to see the common characteristics of the groups.

A well-defined segment has the following features:

- It is homogeneous with respect to a similarity measure within itself.
- It is distinct from other segments with respect to that similarity measure.

Rummikub is a tile-based game for two to four players that was invented by Ephraim Hertzano.¹ Hertzano was born in Romania and developed the game when card games were outlawed under

The data analysis for this chapter was generated using SAS Enterprise Miner software, Version 13.1 of the SAS System for Windows. Copyright © 2013 SAS Institute Inc. SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc., Cary, NC, USA.

¹ Rummikub won the Spiel des Jahres (Germany's Game of the Year) in 1980.

communist rule. The game consists of 104 numbered tiles plus two jokers. The tiles are divided into four unique color groups (usually red, blue, green, and orange) and numbered 1 to 13 so that there are two tiles for each number color combination. The game begins by each player drawing tiles from the 106 facedown tiles—I do not know the official number you are supposed to draw but in my family we draw 20.

After the tiles are drawn, each player organizes their tiles without revealing them to the other players. Once players have had a chance to organize their tiles, play begins. Dean house rules require the oldest player to go first. During a player's first turn, they must create valid sets exclusively from their tiles. A set is either a run of at least three consecutive tiles of the same color or at least three unique colored tiles with the same number. For example, tiles 4, 5, and 6 in blue is a valid run and three 11 tiles, one red, blue, and orange, is a valid set, as shown in Figure 6.1. The official rules require that the initial meld must have a certain value of the summed numbers played. The initial meld threshold was 50 before 1988, then it was dropped to 30; in my



Figure 6.1 Example of Valid Sets in Rummikub

family, kids under 5 do not have an initial threshold until they have beaten their father. If a player cannot meet the initial meld threshold, then they draw a tile and the play passes to the next younger player. After a player has completed the initial meld (hopefully on the first turn), then play continues with players laying down valid sets or runs from their tiles or reorganizing the existing tiles to place their tiles. The objective of the game is to place all of your tiles. The first person to do so is declared the winner. The jokers can be used at any time in place of any numbered tile.

The game of Rummikub is a hands-on experience in segmentation. The tiles can be part of only one group of tiles (either a set or a run) at any particular time, just as segmentations must create mutually exclusive groups. The strategy for partitioning the tiles is different among different players. Each player in my family has a slightly different system for organizing their tiles and how they choose to play their tiles through the game.

In segmentation for business or Rummikub, there is not a single “right” answer, and the best segmentation strategy can change through the course of the game just as segmentation strategies in organizations must adapt to changing market conditions to be successful. I often have three segments to my tiles initially. I have tiles lower than 5 for making sets, tiles higher than 9 for making sets, and tiles between 5 and 9 for making runs. I always hope for an additional segment of jokers, which I keep separate until I can use them to place all of my remaining tiles and win the game. I have these three segments because of the challenges of forming runs. In order to create a run with a black 1 tile, I must have the black 2 and black 3 tiles; at the beginning of the game, I have only a 3.7% ($4/106$ because there are two tiles for each color) chance of getting that combination. In contrast, to get a set of 1 tiles, I have a 5.6% chance ($6/106$ because there are eight tiles for each number, but a set cannot contain two tiles of the same color so that leaves two tiles for 1 orange, 1 red, and 1 blue). The same probability situation occurs at the upper extreme of the numbers. The middle numbers segment has better odds for runs because there are multiple numbers that can be drawn or used from the existing board to create those runs. In the case of an 8 red, I can use 6 red, 7 red, 9 red, or 10 red to create a run.

My youngest son, who has just started playing Rummikub, has a simpler strategy with just two segments, jokers and everything else. He is much stronger at putting runs together than sets so runs dominate his play. (I think this is due to the challenge of not having the same color in a set, which he sometimes forgets and his older siblings are quick to point out). This highlights another parallel between segmentation in the game and business: For most applications, a simple segmentation strategy is better than no strategy at all. If my youngest son were to have just one segment (all tiles played equally), the joker would be played very early in the game and provide him little advantage to the point of being essentially wasted.

The segmentation of the jokers from the numbered tiles points out the difference between clustering and segmentation. The terms are often used interchangeably because they are so closely related, but segmentation is a superset of clustering. Clustering is done by an algorithmic method, and it is mechanical. Segmentation often uses clustering techniques and then applies business rules like the treatment of keeping jokers in a special segment until the end of the game.

Many companies build their own segments but others prefer to buy them to aid primarily in their marketing activities. One commercial set of segments is done by PRIZM (which is currently owned by Nielsen). When I was considering moving from northern Virginia to Cary, North Carolina, I spent hours reading the descriptions of the different segments and imagining the people who lived there. PRIZM uses 66 segments to describe the people across America. Here is an example of three segments to give you an idea of the type of description and characterization given:

19 Home Sweet Home

Widely scattered across the nation's suburbs, the residents of Home Sweet Home tend to be upper-middle-class married couples living in mid-sized homes with few children. The adults in the segment, mostly between the ages of 25 and 54, have gone to college and hold

professional and white-collar jobs. With their upscale incomes and small families, these folks have fashioned comfortable lifestyles, filling their homes with toys, TV sets and pets.

07 Money & Brains

The residents of Money & Brains seem to have it all: high incomes, advanced degrees and sophisticated tastes to match their credentials. Many of these city dwellers, predominantly white with a high concentration of Asian Americans, are married couples with few children who live in fashionable homes on small, manicured lots.

51 Shotguns & Pickups

The segment known as Shotguns & Pickups came by its moniker honestly: It scores near the top of all lifestyles for owning hunting rifles and pickup trucks. These Americans tend to be young, working-class couples with large families—more than half have two or more kids—living in small homes and manufactured housing. Nearly a third of residents live in mobile homes, more than anywhere else in the nation.

These segments are designed to be informative yet concise and allow organizations to match the goods or services with those people most likely to respond positively. For certain specialized applications, each segment may have additional constraints that would need to be satisfied as well.

Since each segment is homogeneous, which means lower variance within the individual segments, it is possible to get better predictive models for each segment than you could achieve by building a predictive model for the entire population. Plainly stated, segmentation will often improve your ability to make correct predictions or classifications during predictive modeling activities, assuming each segment is large enough to build quality predictive models.

Feature films are another example of segmentation. Movies are classified into categories for many reasons. Perhaps the most important is that if they were not, the list of film choices would overwhelm most people and be too difficult to navigate effectively. We segment movies by their genre—horror, sci-fi, action, romantic comedy, and so on. Movies are also segmented by their rating and several other categories. I doubt there is a movie that appeals to the entire population, so production studios and sponsors want to understand the specific demographic for which this film is intended. Fast food chains would never want to distribute action figures from an R-rated horror film with their children's menu. Likewise, PG-rated animation films of a classic Grimm fairy tale would not be expected to be popular in the 21- to 25-year-old male demographic.

CLUSTER ANALYSIS

Clustering is the process of organizing objects into self-similar groups by discovering the boundaries between these groups algorithmically using a number of different statistical algorithms and methods. Cluster analysis does not make any distinction between dependent and independent variables. It examines the entire set of interdependent relationships to discover the similarity relationships between the objects in order to identify the clusters. Cluster analysis can also be utilized as a dimension reduction method in which the number of objects are grouped into a set of clusters, and then a reduced set of variables are used for predictive modeling. This practice helps reduce issues related to multicollinearity.

Clustering can be grouped into three categories, unsupervised, semisupervised, and supervised:

1. **Unsupervised clustering.** The objective of unsupervised clustering is to maximize the intracluster similarity and to minimize the intercluster similarity, given a similarity/dissimilarity measure. It uses a specific objective function (e.g., a function that minimizes the intraclass distances to find tight clusters). It uses a data set that has no target variable. *K*-means and hierarchical clustering are the most widely used unsupervised clustering techniques in segmentation.

2. **Semisupervised clustering.** In addition to the similarity measure that is also used in unsupervised clustering, semisupervised clustering utilizes other guiding/adjusting domain information to improve the clustering results. This domain information can be pairwise constraints (must-link or cannot-link) between the observations or target variables for some of the observations. This guiding information is used either for adjusting the similarity measure or for modifying the search for clusters in order to bias the search. In addition to the objectives for unsupervised clustering, semisupervised clustering has the objective of obtaining high consistency between the clusters and the guiding/adjusting domain information.
3. **Supervised clustering.** The goal of supervised clustering is to identify clusters that have high probability densities with respect to individual classes (class-uniform clusters). It is used when there is a target variable and a training set that includes the variables to cluster.

DISTANCE MEASURES (METRICS)

An objective way to evaluate segments is based on a similarity or dissimilarity between the objects and clusters in the data set. Many possible distance measures can be used to compute clusters. The most common are mentioned below. Two types of distance can be measured and accessed: the distance between objects and the distance between clusters. Both are important metrics depending on the clustering application.

1. Distance measures between objects

They are used to measure the distances between the objects in the data set that is used for clustering.

- **Euclidean distance.** It is basically the geometric distance between objects in the multidimensional space. It corresponds to the length of the shortest path between two objects. It is used to obtain sphere-shaped clusters.
- **City block (Manhattan) distance.** It corresponds to the sum of distances along each dimension and is less sensitive to outliers. It is used to obtain diamond-shaped clusters.

- **Cosine similarity measure.** It is calculated by measuring the cosine of angle between two objects. It is used mostly to compute the similarity between two sets of transaction data.

2. Distance measures between clusters

They are used to measure the distances between clusters. The measures that are used in hierarchical clustering are:

- **Average linkage.** It is the average distance between all the points in two clusters.
- **Single linkage.** It is the distance between nearest points in two clusters
- **Complete linkage.** It is the distance between farthest points in two clusters.

EVALUATING CLUSTERING

Since clustering is used mostly in an unsupervised way, there needs to be a measure to evaluate the quality of the clusters provided from a particular algorithm. These evaluation criteria are used mainly to find out the variability or noise in clusters, to find the optimal number of clusters for the data, to compare the clustering algorithms on the quality of their solutions, and to compare two sets of results obtained from cluster analysis.

1. **Internal evaluation criterion.** It is part of cluster analysis and is specific to the method. It is computed for the data that is used for cluster analysis. Though the objective functions for all the clustering algorithms are very similar (high intracluster similarity and low intercluster similarity), the evaluation criteria that are used in different algorithms differ one from another. For example, while sum of squares error (SSE) can be used for evaluating the k -means clustering algorithm, it cannot be used for density-based clustering algorithms.
2. **External evaluation criterion.** This criterion can be obtained using a separate data set that was not used for cluster analysis. It is used to measure how representative the clusters are with respect to a true cluster when the class labels are given or how consistent they are with respect to different clusters

when they are obtained using different parameters/methods. It can be measured in terms of purity, entropy, random index, or F-measure.

NUMBER OF CLUSTERS

For some of the unsupervised clustering algorithms, such as k -means, one of the main questions is "How many clusters are in the data?" or asked another way "What is the right number of clusters to segment my data?" There are several objective metrics such as gap and cubic clustering criteria (CCC). CCC uses a simulated reference distribution to evaluate the right number of clusters. In my work at SAS, we have recently filed a patent that extends on the CCC method by generating many reference distributions that are aligned with principal components. This method is called aligned box criteria (ABC), and it has the advantage of being more straightforward in the diagnostic interpretation.

Why was ABC not developed until now? As was discussed in Part One of this book, the computational power that has recently become commodity is a major factor. Instead of generating just one reference distribution, you can do principal component analysis for each number of clusters to be considered and generate reference distributions based on these values with hundreds to thousands of variables considered in modern segmentation problems. This is infeasible without significant computational resources.

Here is an example of unsupervised segmentation using clustering. I have generated an X and Y variable from seven different distributions. Each distribution has 1,000 points. The data is plotted in Figure 6.2 with each distribution getting its own symbol.

Without the symbols, it would be a very difficult problem for a human to segment all of these points into seven different classes. To assist practitioners a grid search method can be used. Cluster analysis is performed multiple times with a different number of clusters each time and the results are then analyzed to figure out the best choices for the number of clusters in the data. Using the ABC method, we get the diagnostic plot shown in Figure 6.3 indicating the possible candidates for the true number of clusters. This objective measure gives a starting point to determine the "right" number of segmentations for your problem.

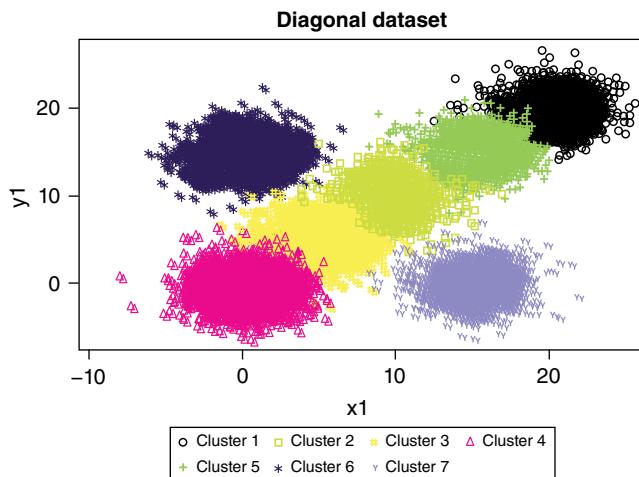


Figure 6.2 Plot of Seven Different Sets of Generated Data

How this plot is interpreted is as follows: Beginning with the minimum number of clusters, we see a decrease in the Y value. After Y starts to increase, we search for the peak values according to the Y

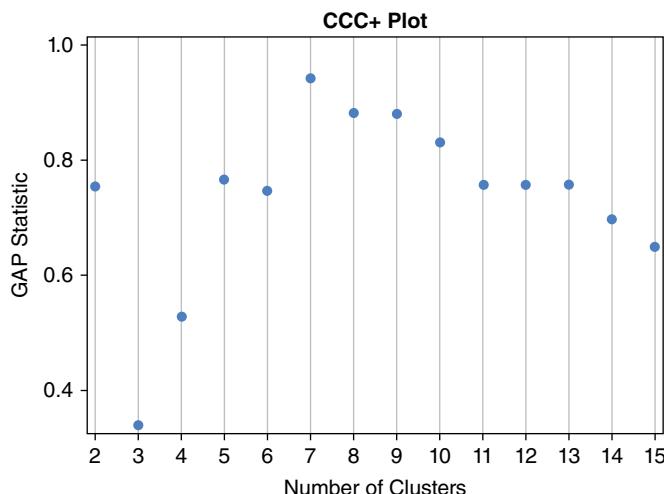


Figure 6.3 Diagnostic Plot to Assess the Best Number of Clusters

values. For example, here we see that the first peak value of Y is at 5 number of clusters and the global peak of Y is at 7 number of the clusters. Both 5 and 7 are good candidates for this cluster set. The algorithm found 3 clusters in the diagonal region of the data set for 5 number of clusters, while it found 5 clusters in the same region for 7 number of clusters.

K-MEANS ALGORITHM

K -means is one of the most widely used clustering techniques because of its simplicity and speed. It partitions the data into k clusters by assigning each object to its closest cluster centroid (the mean value of the variables for all objects in that particular cluster) based on the distance measure used. It is more robust to different types of variables. In addition, it is fast for large data sets, which are common in segmentation.

The basic algorithm for k -means works as follows:

1. Choose the number of clusters, k .
2. Select k cluster centroids (e.g., randomly chosen k objects from the data set).
3. Assign each object to the nearest cluster centroid.
4. Recompute the new cluster centroid.
5. Repeat step 3 and 4 until the convergence criterion is met (e.g., the assignment of objects to clusters no longer changes over multiple iterations) or maximum iteration is reached.

Many issues need to be considered in k -means clustering:

- The k -means algorithm requires the number of clusters k as an input. The ABC method can be used to estimate the number of clusters.
- The similarity/distance measure should be selected depending on the task.
- Clusters may converge to a local minimum. Due to this issue, the clusters that are obtained might not be the right ones. To avoid this, it might be helpful to run the algorithm with different initial cluster centroids and compare the results.

The k -means algorithm can take advantage of data parallelism. When the data objects are distributed to each processor, step 3 can be parallelized easily by doing the assignment of each object into the nearest cluster in parallel. In order to update cluster centroids at each node for every iteration, communication of cluster centroid-related information between nodes can also be added in steps 2 and 4.

HIERARCHICAL CLUSTERING

Hierarchical clustering generates clusters that are organized into a hierarchical structure. Visualizing this hierarchical structure can be used to understand the structure of clusters in the data set in addition to the clusters themselves. This technique only requires a measure of similarity between objects. It does not require you to specify the number of clusters. You can obtain any number of clusters by cutting the hierarchical structure at a proper level. There are two main categories for hierarchical clustering:

1. **Divisive clustering.** This is a top-down approach. It considers all objects in a big, single cluster first and then divides the clusters into smaller groups at each level.
2. **Agglomerative clustering.** This is a bottom-up approach. It considers each object to be an individual cluster first, and then it combines them into bigger clusters at each level according to their similarity. Clusters can be generated by minimizing the squared Euclidean distance to the center mean (Ward's method) or based on the distance between clusters (single linkage, complete linkage, average linkage methods).

PROFILING CLUSTERS

Once clusters are computed using one of the clustering techniques, an analysis should be done in order to describe each cluster. This is called profiling. This analysis can be done in terms of the internal variables that are used in the cluster analysis and the external variables that are

important for the analysis of the segments. For example, you can do cluster analysis using the age and income of the customers as internal variables. After you obtained the clusters, you can do an analysis on the demographic information of customers for each cluster. Mean, range, or standard deviation of variables in each cluster is commonly used in profiling.

CHAPTER 7

Incremental Response Modeling

A standard part of marketing campaigns in many industries is to offer coupons to encourage adoption of the new goods or service. This enticement is often essential for success because it provides some additional incentive to persuade customers to switch products if they are currently satisfied with the goods or service in question. For example, if I am a longtime buyer of laundry detergent XYZ, to get me to try a competitor's laundry product detergent ABC, I will need some incentive great enough to get me to move outside my comfort zone of buying detergent XYZ on my trips to the store. This incentive or inducement could be superior performance, but I will never know that the performance is superior unless I try the product. Another inducement could be value. Detergent ABC cleans just as well as detergent XYZ, and I get a larger quantity of detergent ABC for the same price as detergent XYZ. This strategy, like superior quality, is also predicated in me trying the product. All strategies that will successfully convert from detergent XYZ to detergent ABC require me to try the product, and the most common

way to do that is to give me a coupon. Coupons are very common for most household products and packaged foods. These coupons can, in some cases, be successful in changing buying behavior from one product to another product. Now consider this alternative case: I have been buying detergent XYZ regularly, but I am not very happy with it and I have decided to switch to detergent ABC because of a new ad I saw while watching TV. Now when I receive a coupon for detergent ABC that I had planned to buy anyway, I purchase my detergent ABC, but the manufacturer yields less profit because I purchased a product I had already planned to at a lower price than I was already willing to pay.

This is what incremental response modeling methods do: determine those people who purchased the product because of an inducement (coupon) in order to maximize the profit for the organization.

BUILDING THE RESPONSE MODEL

The results of a marketing campaign will fall into one of three groups: people who will not purchase the product regardless of the incentive, those who switched products because of the campaign, and those who were going to switch already and can now purchase the product at a reduced rate. Of these three groups, we would like to target with our marketing only those who switched just because of the coupon. There is a fourth category that I will discuss here and not refer to again. This is the group referred to as sleeping dogs; they are mostly encountered in political campaigns, not marketing campaigns. Sleeping dogs are those people who purchase your product or subscribe to your political view but by including them in a campaign they respond negatively and leave your brand.

Public radio is another example to demonstrate incremental response. Public radio, supported by listeners, has fund drives several times throughout the year to raise money to support the programming. Many listeners donate as soon as they hear the fund drive begin because they want to support the programming and feel a duty to do so. Another group probably would not donate to the station except for the appeal of a coffee mug, tote, video collection, or some other gift to reward them for their generosity that sways them to pick up the phone or go to the website and contribute. The problem for public broadcasting is it cannot discern between the two groups. If it could

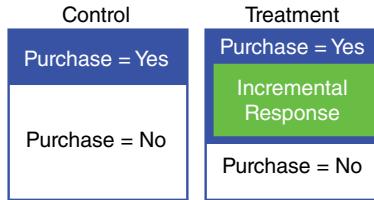


Figure 7.1 Control versus Treatment

discern between the groups, it could save the costs associated with those giveaway items and therefore reduce overall cost.

The method behind incremental response modeling is this: Begin with a treatment and control group. These groups should follow a methodology from the clinical trial literature. The treatment is a coupon. Note that you can use multiple treatments, but here we will discuss only the binary case.

Once you have divided your sample into the two assigned groups, administered either the treatment or the control, and then gathered the results, you can begin to apply this methodology. Figure 7.1 shows the difference between the control group and treatment group. The treatment group received the coupon for detergent ABC while the control group received no coupon. You can see that the coupon was effective in increasing sales (based on the size of the box labeled “Purchase=Yes”), but the top section in the treatment group represents all people who purchased the detergent. The ones in the box labeled “Incremental Response” purchased because of the coupon and the rest would have purchased regardless. Therefore, the treatment group did not maximize profit because the detergent was sold at a lower price than could have otherwise been demanded. It is rare but possible that in some cases the treatment group, those who received the coupon, could actually generate less profit than the control group, doing nothing.

MEASURING THE INCREMENTAL RESPONSE

A traditional approach to accessing the incremental response is using a differencing technique from two predictive models. Take the likelihood to purchase from the predictive model built for the treatment group:

$$P_t(y = 1 | x)$$

The likelihood to purchase for the control group is similar:

$$P_c(y = 1 | x)$$

The incremental response likelihood can be calculated as the difference

$$P_D = P_t(y = 1 | x) - P_c(y = 1 | x)$$

Then sort the resulting P_D from largest to smallest, and the top deciles are the incremental responders. Any predictive model can be employed in the differencing technique, such as the regression-based differencing model and the tree-based differencing model.

An improved method is to look only at the control group, the people who did not get the coupon, and classify each person as an outlier or not. Several techniques can be used for classifying outliers when you have only one group. A method that has good results classifying outliers is one-class support vector machines (SVMs).

Recently a new method was suggested that uses an outlier detection technique, particularly the one-class SVM. The suggested method uses the control group data to train the model and uses the treatment group as a validation set. The detected outliers are considered as incremental responses. This new method shows much better results than the differencing technique. The technique is illustrated with plots below, but more details can be found in the paper by Lee listed in the references.

In Figure 7.2, we see a graphical representation of the points from the control group that have been identified as outliers. The dots closer to the origin than the dashed line are classified as part of the negative class, and other dots up and to the right of the dashed line are classified to the positive class. The points in the negative class are considered outliers (those between the origin and the dashed line). They receive this designation because there are particular features, or a combination of many features, that identify them as different from the overall group.

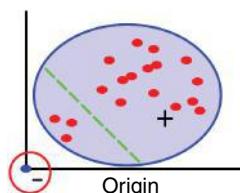


Figure 7.2 Outlier Identification of Control Group

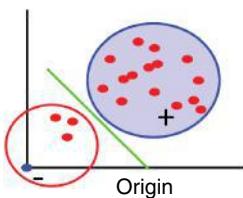


Figure 7.3 Separation of Responders and Nonresponders in the Control Group

One reason to identify some of the points in the control group as outliers is to narrow the region that the control group identifies so that we can better determine which observations are incremental responders when we apply the model to the treatment group. To apply this to our example, we would look at the people who purchased detergent ABC and then, using a one-class SVM model, identify some of those as outliers. This would leave us a set of ranges for each attribute of our customers that we can use to identify them as people who would purchase detergent ABC without a coupon, as shown in Figure 7.3.

The region of the responders for the control group as shown by the larger circle in Figure 7.3 this region can then be projected to the treatment group. Applying this region to the treatment group will identify the incremental responders, those individuals who purchased detergent ABC because of the coupon. This is illustrated graphically in Figure 7.4. This is only a representation in two dimensions. In practice, this would be in dozens, hundreds, or even thousands of dimensions.

Figure 7.4 shows the projection of the control group, those who bought detergent ABC without a coupon after the outliers were removed using a one-class SVM model and projecting the region to the treatment group, those who bought detergent ABC after being sent a coupon. To interpret this plot, examine the different parts. The first group to identify is the treatment group that falls inside the oval; these are responders who were unaffected by the coupon. This means that for those people inside the upper oval, the coupon did not influence their purchasing choice (buy detergent ABC regardless of the promotion either to buy detergent ABC or to not buy). The data points outside of the oval in the treatment response group

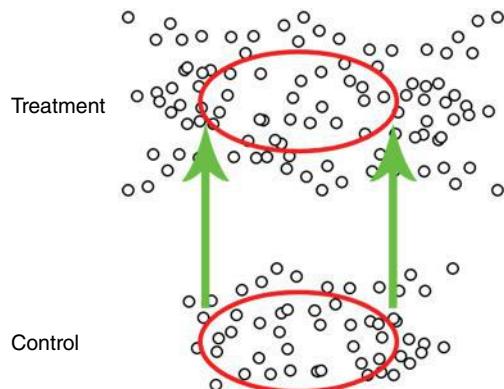


Figure 7.4 Projection of Control Group to Treatment Group

are the potential incremental responders. Those are the people who purchased because of the treatment; in this specific example, the coupon for detergent ABC. I used the word “potential” above because there is no definitive way in real life to objectively measure those people who responded only as a result of the treatment. This can be tested empirically using simulation, and that work has illustrated the effectiveness of this method. Figure 7.5 is an example of a simulation study.

Figure 7.5 shows 1,300 responders to the treatment group. This includes 300 true incremental responders. The method described above identified 296 observations as incremental responders, and 280 of

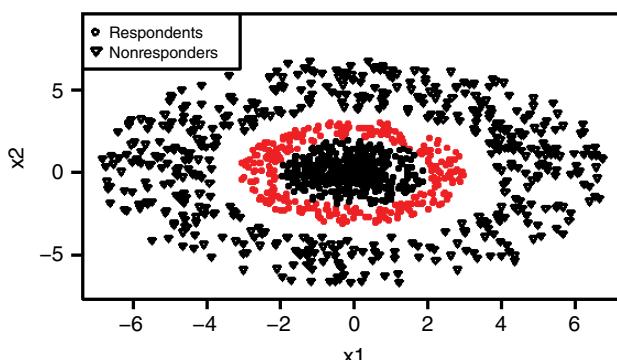


Figure 7.5 Simulation of 1,300 Responders to Coupon Offer

Table 7.1 Classification of Incremental Responders

	Correctly Classified	Falsely Classified
Responders	280	20
Nonresponders	986	16

those identified were true positives. This is all the more impressive because, as you can see, there is no simple way to use straight lines and separate the gray true incremental responders from the black nonincremental responders. This leaves 20 true responders who were not identified and 16 who were incorrectly identified. See Table 7.1 for a tabular view.

This simulation yields a 5.4% error rate, which is a significant improvement over the differencing method explained at the beginning of the chapter. Incremental response modeling holds much promise in the areas of targeted advertising and microsegmentation. The ability to select only those people who will respond only when they receive the treatment is very powerful and can contribute significantly to increased revenue. Consider the typical coupon sells the goods or service at 90% of the regular price (a 10% discount). Every correctly identified true incremental responder will raise revenue 0.9 and every correctly identified nonincremental responder (those who are not influenced by the treatment either to purchase or not) will raise revenue by 0.1 because those items will not be sold at a discount needlessly. Then add in the nominal cost of the treatment—ad campaign, postage, printing costs, channel management. We have the following revenue adjustments:

$$\text{Incremental Revenue} = 0.9r + 0.1n - \text{campaign costs}$$

where

r = incremental responders who will purchase the product if they receive the coupon but otherwise will not

n = nonresponders who will not buy the product even if they received the coupon

By taking the simulation example but increasing the error rate to nearly double at 10%, you can see the advantage of using incremental response:

$$\text{Incremental response revenue} = 333 \text{ units} = .9(270) + .1(900) - \text{fixed costs}$$

compared to:

$$\text{Control only} = 100 \text{ units} = .9(0) + .1(1000) - \text{campaign costs}$$

$$\text{Treatment only} = 270 \text{ units} = .9(300) + .1(0) - \text{campaign costs}$$

for the control and treatment-only groups. When looking at the fixed costs of each scenario, the treatment will have the largest effect because a coupon is being sent to each of the 1,300 people. This will be followed by the incremental response group, where coupons are sent only to those predicted to respond; and finally the control group, where there are no campaign costs because of the lack of campaign.

$$\text{Treatment campaign costs} > \text{Incremental response campaign costs} >$$

$$\text{Control campaign costs} = 0$$

When the campaign costs are added to the calculations, the incremental response is an even better option to either the treatment or the control group. This increasing amount of information that is made available will over time reduce the error rates, yielding even larger revenues for those organizations that leverage this powerful technique.

CHAPTER 8

Time Series Data Mining

Times series data mining is an emerging field that holds great opportunities for conversion of data into information. It is intuitively obvious to us that the world is filled with time series data—actually transactional data—such as point-of-sales (POS) data, financial (stock market) data, and Web site data. Transactional data is time-stamped data collected over time at no particular frequency. Time series data, however, is time-stamped data collected over time at a particular frequency. Some examples of time series data are: sales per month, trades per weekday, or Web site visits per hour. Businesses often want to analyze and understand transactional or time series data. Another common activity also includes building models for forecasting future behavior.

Unlike other data discussed in this book, time series data sets have a time dimension that must be considered. While so-called cross-sectional data, which can be used for building predictive models, typically features data created by or about humans, it is not unusual to find that time series data is created by devices or machines, such as sensors, in addition to human-generated data. The questions that arise from dealing with vast amounts of time series are typically different

from what have been discussed earlier. Some of these questions will be discussed in this chapter.

It is important to note that time series data is a large component to the big data now being generated. There is a great opportunity to enhance data marts used for predictive modeling with attributes derived from time series data, which can improve accuracy of models. This new availability of data and potential improvement in models is very similar to the idea of using unstructured data in the context of predictive modeling.

There are many different methods available for the analyst who wants to go ahead and take advantage of time series data mining. High-performance data mining and big data analytics will certainly boost interest and success stories of this exciting and evolving area. After all, we are not necessarily finding lots of new attributes in big data but similar attributes to what already existed, measured more frequently—in other words, the time dimension will rise in importance. In this chapter, we focus on two important areas of time series data mining: dimension reduction and pattern detection.

REDUCING DIMENSIONALITY

The main purpose of dimension reduction techniques is to come up with a condensed representation of the information provided in a time series for further analysis. Instead of looking at the original data directly, new data is created that consists of many fewer observations but contains the most important features of the original data. The point of dimension reduction is that the original data may feature too many observations or points, which make it almost impossible to deal with in an efficient and timely manner. As an example, think of a sensor on a wind turbine that constantly streams information. An engineer might not be interested in constantly monitoring this information but wants to determine seasonal or trend patterns to improve power generation or predict needed maintenance before the turbine goes offline due to a part failure. Another example in the case of sales data is that a predictive model for cross-selling opportunities might become more accurate if a modeler can include buying behavior over time instead of static statistics, such as total money spent.

An easy-to-understand example of dimension reduction is commonly called seasonal analysis and typically is applied to transactional

data. In order to reduce the original transactional data, statistics such as mean, minimum, or maximum are calculated at each point over a seasonal cycle (e.g., four quarters, 12 months, etc.). Depending on the seasonal cycle of choice, the amount of data can be reduced quite dramatically to just a few points. As you can see, this type of idea can be extended to time series data as well and to almost every time series analysis idea. Classical time series decomposition can be used in a similar fashion to derive components such as trend, cycles, or seasonality from time series data. Another easy and simple dimension reduction technique using descriptive statistics is a piecewise aggregate approximation (PAA) method, which is a binning-type dimension reduction of time series. For example, the time axis will be binned and the mean or median within each bin is calculated, and then the series of means or medians will be the reduced new time series. Again, these statistics can be used as input for further analysis, such as segmentation or predictive modeling. In the case of building predictive models for churn analysis, information like downward-trending sales patterns can be useful attributes for building more accurate models.

So far I have mainly used descriptive statistics for dimension reduction, which might filter out too much information. You may be wondering if it would be more appropriate to create new time series with fewer points instead, points that reflect the overall shape of original series. In other words, is it possible to represent the original time series as a combination of well-understood basic functions, such as straight lines or trigonometric functions? Examples of such approaches that include discrete Fourier transformation (DFT), discrete wavelet transformation (DWT), or singular value decomposition (SVD). DFT, for example, represents time series as a linear combination of sines and cosines but keeps only the first few coefficients. This allows us to represent the information of a time series with fewer points—but the main features of the time series will be kept.

DETECTING PATTERNS

The second area is pattern detection in time series data. This area can be divided into two subareas: finding patterns within a time series and finding patterns across many time series. These techniques are used

quite successfully for voice recognition using computers, where the data can be represented as time series reflecting sound waves.

Let us start with an example for finding patterns within a time series: When analyzing streaming data collected from sensors, you may want to find patterns that indicate ill behavior of a system. If such patterns can be detected, an engineer could use this information to monitor a system and prevent future failures if the pattern starts to occur. This is the underlying concept of what sometimes is called predictive maintenance: Engineers try to determine problems before they occur—outside of scheduled maintenance activities.

When faced with many time series, the idea of pattern detection can be extended to address questions like: Which time series indicate similar behavior? Which time series indicates unusual behavior? And finally, given a time series of interest, which of the many other series is most similar to this target series? Similarity analysis is the method of choice for coping with these types of questions.

Similarity analysis, as the name suggests, can be used to measure the similarity between time series. It is a very useful analysis that can be applied for clustering or classifying time series which vary in length or timing. Two very prominent application areas are fraud detection and new product forecasting.

Fraud Detection

Credit card providers are using similarity analysis to automate the detection of fraudulent behavior in financial transactions. They are interested in spotting exceptions to average behavior by comparing many detailed time series against a known pattern of abusive behavior.

New Product Forecasting

New product forecasting for consumer goods manufacturers and retailers is a constant challenge. The situations include predicting entirely new types of products, new markets for existing products (such as expanding a regional brand nationally or globally), and refinements of existing products (such as “new and improved” versions or packaging changes). All require a forecast of future sales without historic data

for the new product. However, by using techniques such as similarity analysis, an analyst can examine the demand patterns of past new products having similar attributes and identify the range of demand curves that can be used to model demand for the new product.

In a way, similarity analysis can be thought of as an extension of techniques based on distance measures. If you want to determine routes between cities, you first need to create a matrix of distances that measure the distances between all cities of interest. When transitioning to apply this concept to time series, you are faced with the challenge of dealing with the dynamic nature of time series. Two series can feature the same shape but are of different length. In this case, standard distance measures cannot be applied. A similarity measure is a metric that measures the distance between time series while taking into account the ordering. Similarity measures can be computed between input series and target series, as well as similarity measures that “slide” the target sequence with respect to the input sequence. The slides can occur by observation index (sliding-sequence similarity measures) or by seasonal index (seasonal-sliding-sequence similarity measures).

These similarity measures can be used to compare a single input sequence to several other representative target sequences. For example, given a single input sequence, you can classify the input sequence by finding the “most similar” or “closest” target sequence. Similarity measures can also be computed between several sequences to form a similarity matrix. Clustering techniques can then be applied to the similarity matrix. This technique is used in time series clustering. When dealing with sliding similarity measures (observational or seasonal index), we can compare a single target sequence to subsequence pieces of many other input sequences on a sliding basis. This situation arises in time series analogies. For example, given a single target series, you can find the times in the history of the input sequence that are similar while preserving the seasonal indices.

In this chapter, we have been dealing with questions that are dynamic in nature and for which we are typically faced with so-called time series data. We looked at ways to handle the increased complexity introduced by the time dimension, and we focused mainly on dimension reduction techniques and pattern detection in time series data.

TIME SERIES DATA MINING IN ACTION: NIKE+ FUELBAND

Nike in 2006 began to release products from a new product called Nike+, beginning with the Nike+ iPod sensor to track exercise data into popular consumer devices. This product line of activity monitors has expanded quite a bit and currently covers integration with basketball shoes, running shoes, iPhones, Xbox Kinect, and wearable devices including the FuelBand.



The Nike+ FuelBand is a wrist-worn bracelet device that contains a number of sensors to track users' activity. The monitoring of your activity causes you to be more active, which will help you achieve your fitness goals more easily. The idea of generating my own data to analyze and helping increase my fitness level was (and still is) very appealing to me, so I dropped as many subtle and not-so-subtle hints to my wife that this would make a great birthday present. I even went as far as to take my wife to the Nike Town store in midtown Manhattan to show her just how cool it was. She was tolerant but not overly impressed. I found out afterward that she had already ordered the FuelBand and was trying not to tip me off.

So for my birthday in 2012, I got a Nike+ FuelBand. I have faithfully worn it during my waking hours except for a ten-day period when I had to get a new one due to a warranty issue.¹ So to talk about how to deal

¹ Nike was great about replacing the device that was covered by warranty.

with times series data, I decided to use my activity data as captured on my Nike FuelBand. The first challenge in this was to get access to my data. Nike has a very nice Web site and iPhone app that let you see graphics and details of your activity through a day, but it does not let you answer some of the deeper questions that I wanted to understand, such as:

- Is there a difference in my weekend versus weekday activity?
- How does the season affect my activity level?
- Am I consistently as active in the morning, afternoon, and evening?
- How much sleep do I get on the average day?
- What is my average bedtime?

All of these questions and more were of great interest to me but not possible to answer without a lot of manual work or an automated way to get to the data. One evening I stumbled on the Nike+ developer's site. Nike has spent a good amount of time developing an application program interface (API) and related services so that applications can read and write data from the sensors. The company has a posted API that allows you to request data from a device that is associated with your account. For my account I have mileage from running with the Nike+ running app for iPhone and the Nike+ FuelBand. I went to the developer Web site and, using the forms there, I was able to play with the APIs and see some data. After hours of struggling, I sought out help from a friend who was able to write a script using Groovy to get the data in just an hour. This is an important idea to remember: In many cases, turning data into information is not a one-man show. It relies on a team of talented individual specialists who complement each other and round out the skills of the team. Most of the time spent on a modeling project is actually consumed with getting the data ready for analysis. Now that I had the data, it was time to explore and see what I could find. What follows is a set of graphics that show what I learned about my activity level according to the Nike+ FuelBand.

Seasonal Analysis

The most granular level of FuelBand data that can be requested through the API is recorded at the minute level. Using my data from September 2012 until October 2013, I created a data set with about

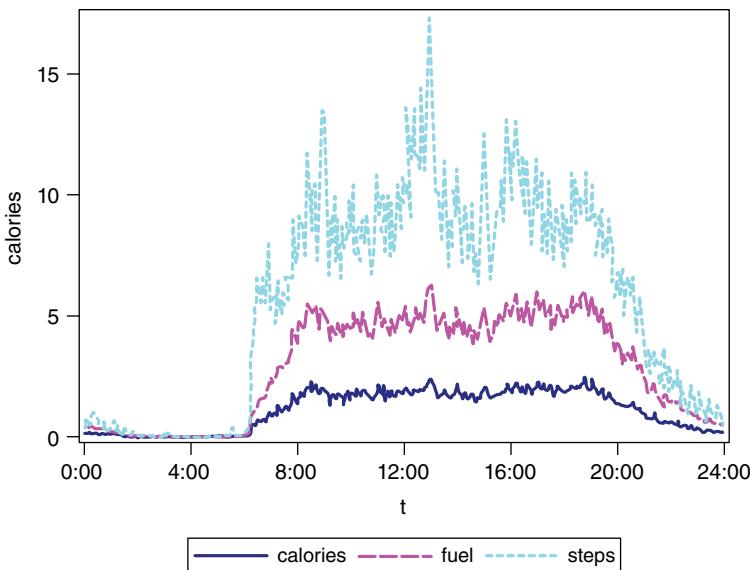


Figure 8.1 Seasonal Index of Authors Activity Levels

560,000 observations. One of the easiest ways to look into that kind of big-time data is seasonal analysis. Seasonal data can be summarized by seasonal index. For example, Figure 8.1 is obtained by hourly seasonal analysis.

From Figure 8.1, you can see that my day is rather routine. I usually become active shortly after 6:00 AM and ramp up my activity level until just before 9:00 AM, when I generally arrive at work. I then keep the same activity level through the business day and into the evening. Around 7:00 PM, I begin to ramp down my activity levels until I retire for the evening, which appears to happen around 12:00 to 1:00 AM. This is a very accurate picture of my typical day at a macro level. I have an alarm set for 6:30 AM on weekdays, and with four young children I am up by 7:00 AM most weekend mornings. I have a predictable morning routine that used to involve a five-mile run. The workday usually includes a break for lunch and brief walk, which show up as a small spike around 1:00 PM. My afternoon ramps down at 7:00 PM because that is when the bedtime routine starts for my children and my activity level drops accordingly.

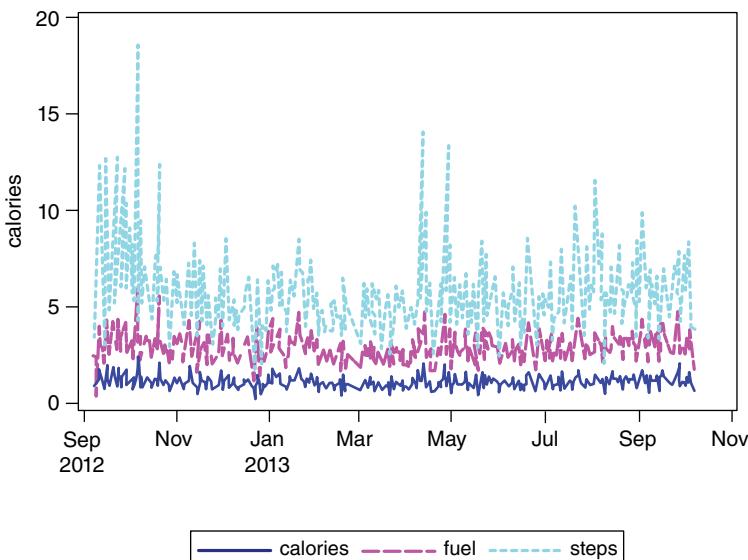


Figure 8.2 Minute-to-Minute Activity Levels Over 13 Months

Trend Analysis

Figure 8.2 shows my activity levels as a trend over 13 months. You can see the initial increase in activity as my activity level was gamified by being able to track it. There is also the typical spike in activity level that often occurs at the beginning of a new year and the commitment to be more active. There is also another spike in early May, as the weather began to warm up and my children participated in soccer, football, and other leagues that got me on the field and more active. In addition, the days were longer and allowed for more outside time. The trend stays high through the summer.

Similarity Analysis

Let us look at the data in a different way. Since the FuelBand is a continuous biosensor, an abnormal daily pattern can be detected (if any exists). This is the similarity search problem between target and input series that I mentioned earlier. I used the first two months of data to make a query (target) series, which is an hourly sequence averaged over two months. In other words, the query sequence is my average

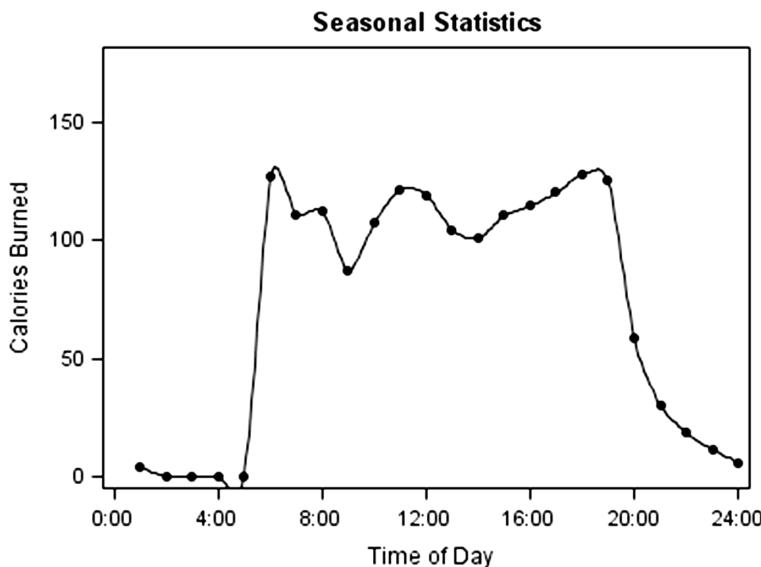


Figure 8.3 Average Pattern of Calories Burned over Two-Month Period

hourly pattern in a day. For example, Figure 8.3 shows my hourly pattern of calories burned averaged from 09SEP2012 to 9NOV2012.

Using the SIMULARITY procedure in SAS/ETS® to find my abnormal day based on calorie burn data.

Using dynamic time warping methods to measure the distance between two sequences, I found the five most abnormal days based on the initial patterns over the first two months of data compared to about 300 daily series from 10 Nov. 2012 to. This has many applications in areas such as customer traffic to a retail store or monitoring of network traffic for a company network.

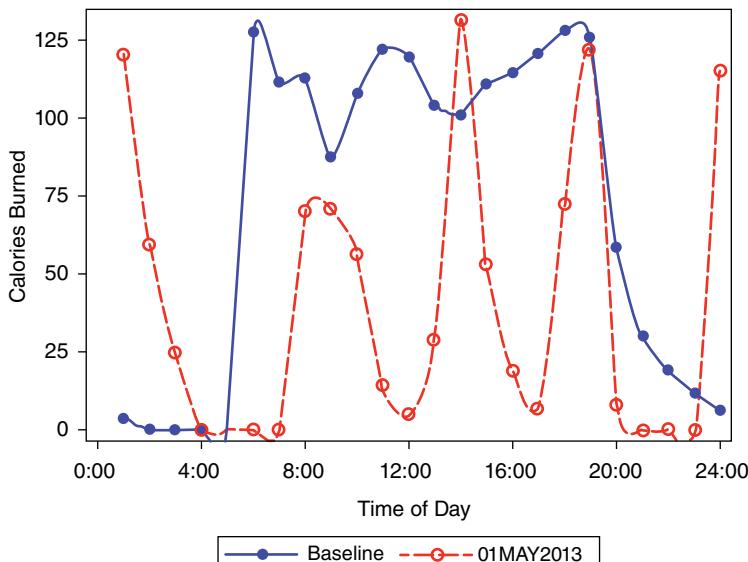
From Table 8.1, you see that May 1, 2013, and April 29, 2013, were the two most abnormal days. This abnormal measure does not indicate I was not necessarily much more or less active but that the pattern of my activity was the most different from the routine I had established in the first two months. These two dates correspond to a conference, SAS Global Forum, which is the premier conference for SAS users. During the conference, I attended talks, met with customers, and demonstrated new product offerings in the software demo area.

Table 8.1 Top Five Days with the Largest Deviation in Daily Activity

Obs	Day Sequence	Distance to Baseline
1	ActDate_5_1_2013	30.5282
2	ActDate_4_29_2013	25.4976
3	ActDate_1_13_2013	18.6811
4	ActDate_8_22_2013	17.6915
5	ActDate_6_26_2013	16.6947

As you can see from Figures 8.4 and 8.5, my conference lifestyle is very different from my daily routine. I get up later in the morning. I am overall less active, but my activity level is higher later in the day.

For certain types of analysis, the abnormal behavior is of the most interest; for others, the similarity to a pattern is of the most interest. An example of abnormal patterns is the monitoring of chronically ill or elderly patients. With this type of similarity analysis, home-bound seniors could be monitored remotely and then have health care staff dispatched when patterns of activity deviate from an individual's norm.

**Figure 8.4** Comparison of Baseline Series and 01May2013

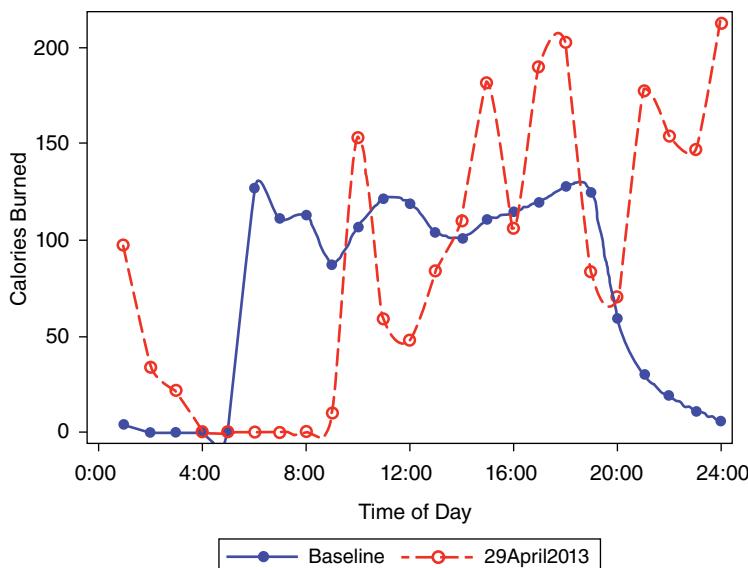


Figure 8.5 Comparison of Baseline Series and 29April2013

This method of similarity analysis is useful because each patient would have their own unique routine activity pattern that could be compared to their current day's activities. An example of pattern similarity is running a marathon. If you are trying to qualify for the Boston Marathon, you will want to see a pattern typical of other qualifiers when looking at your individual mile splits.² Figure 8.6 shows the two most similar days from my FuelBand data.

I have seen a very strong trend to create wearable devices that collect consistent amounts of detailed data. A few examples include Progressive Insurance's Snapshot™ device, which collects telemetric data such as the speed, GPS location, rate of acceleration, and angle

²The Boston Marathon is the only marathon in the United States for which every participant must have met a qualifying time. The qualifying times are determined by your age and gender. I hope to qualify one day.

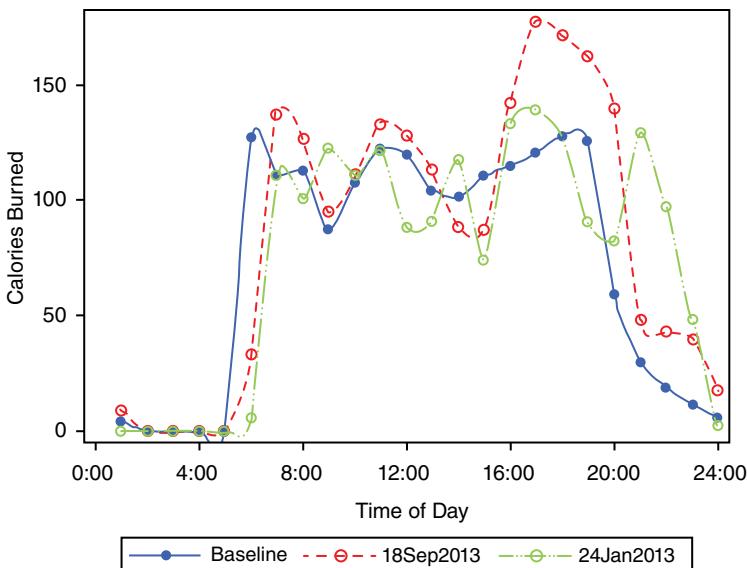


Figure 8.6 Two Most Similar Days to the Baseline

of velocity every 10 seconds while the car is running; or using smartphone GPS data to determine if a person might have Parkinson's disease, as the Michael J. Fox Foundation tried to discover in a Kaggle contest.³

³I am a huge fan of Kaggle and joined in the first weeks after the site came online. It has proven to be a great chance to practice my skills and ensure that the software I develop has the features to solve a large diversity of data mining problems. For more information on this and other data mining contests, see <http://www.kaggle.com>.

CHAPTER 9

Recommendation Systems

If we have data, let's look at data. If all we have are opinions, let's go with mine.

—Jim Barksdale, former Netscape CEO

WHAT ARE RECOMMENDATION SYSTEMS?

Recommendation systems, also known as recommendation engines, are a type of information system whose purpose is to suggest, or recommend, items or actions to users. The recommendations may consist of retail items (movies, books, etc.) or actions, such as following other users in a social network. Typically, the suggestions are a small subset selected from a large collection, according to criteria such as preferences previously expressed by users. Other possible criteria include age, gender, and location.

The most common strategies followed by recommendation systems are the following.

- **Content-based filtering** gathers auxiliary information (e.g., user demographics, music genre, keywords, answers to a questionnaire) to generate a profile for each user or item. Users are

matched to items based on their profiles. Example: Pandora's Music Genome Project.

- **Collaborative filtering** is based on past user behavior. Each user's rating, purchasing, or viewing history allows the system to establish associations between users with similar behavior and between items of interest to the same users. Example: Netflix.

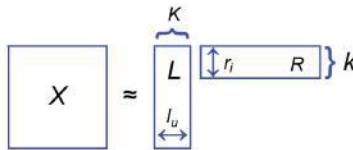
Collaborative filtering is perhaps the most popular of these strategies, due to its domain-free nature. Among collaborative filtering systems, one can further distinguish between neighborhood-based methods, based on user-user or item-item distances, and latent factor or reduced-dimension models, which automatically discover a small number of descriptive **factors** for users and items. Low-rank matrix factorization is the best-known example of reduced-dimension models and is among the most flexible and successful methods underlying recommendation systems.¹ There are many variants of matrix factorization, including probabilistic and Bayesian versions. Restricted Boltzmann machines, a type of deep learning neural network, are another state-of-the-art approach.

WHERE ARE THEY USED?

The scope of application for recommendation systems is vast. Any business that sells a wide variety of items to large numbers of people is a potential user of recommendation systems. They are employed by retailers, such as Amazon, Netflix, and Target; movie and music Web sites, such as Netflix and last.fm; and social networks like Facebook and Twitter. They can also be found in grocery stores, such as Tesco.

Besides the aforementioned examples in retail, media, and social networking, recommendation systems are employed by general-purpose Web sites, such as Yahoo! and Google, in order to select the best advertisements to serve to each user, based on browsing history and other information. Another application is **next-best offer** in marketing decision making, used for instance by Schwan's Foods for increasing sales of frozen food items.

¹The rank of a matrix is the number of linearly independent columns the matrix contains.

**Figure 9.1** Factor Matrix

HOW DO THEY WORK?

We will describe the basic mathematical setting underlying most recommendation systems. Let the ratings be arranged as a matrix X with N_u rows and N_i columns, where N_u is the number of users and N_i is the number of items to be rated. The element x_{ui} of X is the rating given by user u to item i . Typically, we know only the values for a few elements of X . Let D denote the known ratings stored in the form of tuples $(u_1, i_1, x_{u_1 i_1}) \dots (u_n, i_n, x_{u_n i_n})$, as illustrated in Figure 9.1. We assume there are n known ratings.

Baseline Model

A basic, “baseline” model is given by

$$x_{ui} = b_0 + b_u + b_i$$

where b_0, b_u and b_i are a global, user, and item bias terms, respectively. The b_0 term corresponds to the overall average rating, while b_u is the amount by which user u deviates, on average, from b_0 . The same applies to b_i for item i . The goal is to estimate b_0, b_u , and b_i for all u and i , given D .

The bias terms can be estimated by solving the least squares optimization problem

$$\min_{b_0, b_u, b_i} \sum_{(u,i) \in D} (x_{ui} - b_0 - b_u - b_i)^2 + \lambda \left(\sum_u b_u^2 + \sum_i b_i^2 \right)$$

with b_0 set to the average of all known x_{ui} . The first term is the sum of squared errors between the observed ratings and the ratings predicted

Table 9.1 Data Set D of Known Ratings

U_1	i_1	$X_{u_1 i_1}$
	\vdots	
U_n	i_n	$X_{u_n i_n}$

by the model. The second term is a regularization penalty designed to discourage overly large biases, which are associated with overfitting. The parameter λ controls the amount of regularization.

Low-Rank Matrix Factorization

A more flexible model is given by low-rank matrix factorization. Consider the following inner product:

$$x_{ui} = l_u \times r_i$$

where l_u and r_i are vectors of dimension K . The aforementioned model can also be expressed by the matrix product.

$$X = LR$$

where

rows of L (the “left” factor matrix) = vectors l_u for all users

columns of the “right” factor matrix R = vectors r_i for all items

This is depicted in Figure 9.1. In this model, X has rank K , which is typically much lower than either N_u or N_i , hence the name low-rank matrix factorization.

L and R can be estimated by solving the optimization problem:

$$\min_{l_u, r_i} \sum_{(u,i) \in D} (x_{ui} - l_u \cdot r_i)^2 + \lambda \left(\sum_u \|l_u\|^2 + \sum_i \|r_i\|^2 \right)$$

which is similar to the baseline model. The regularization term avoids overfitting by penalizing the norms of the factors. Two popular methods for solving this problem are stochastic gradient descent and alternating least squares.

Stochastic Gradient Descent

Stochastic gradient descent starts from an initial guess for each l_u, r_i and proceeds by updating in the direction of the negative gradient of the objective, according to

$$l_u \leftarrow l_u - \eta(e_{ui}r_i - \lambda l_u)$$

$$r_i \leftarrow r_i - \eta(e_{ui}l_u - \lambda r_i)$$

where

$e_{ui} \triangleq x_{ui} - l_u \cdot r_i$ = prediction error for the rating associated with the pair (u,i)

η = user-defined learning step size

The updates are made one rating at a time, with each pair (u,i) selected uniformly at random, without replacement, from the data set D . This is the reason behind the name “stochastic.” Once a full pass through D (i.e., an epoch) is completed, the algorithm begins a new pass through the same examples, in a different random order. This is repeated until convergence, which typically requires multiple epochs.

Stochastic gradient descent does not require the full data set to be stored in memory, which is an important advantage when D is very large.

Alternating Least Squares

Another well-known method is alternating least squares, which proceeds in two steps. First, it solves Equation 1 with respect to the left factors l_u by fixing the right factors r_i ; then it solves for the r_i with the l_u fixed. Each of these steps can be tackled by a standard least squares solver. Define n_u as the number of ratings from user u and $R[u]$ as the restriction of R to the items rated by u . Let also the vector x_u denote the ratings given by u , in the same order as in $R[u]$. Then the estimate of l_u is found by solving

$$\begin{bmatrix} R[u] \\ \sqrt{\lambda n_u} I_K \end{bmatrix} l_u = \begin{bmatrix} x_u \\ 0 \end{bmatrix}$$

where I_K is the $K \times K$ identity matrix. The estimate of r_i is analogous.

Alternating least squares has the advantage of being simpler to parallelize than stochastic gradient descent. It has the disadvantage of larger memory requirements. The entire data set needs to be stored in memory, which can be an issue for large D .

Restricted Boltzmann Machines

There exist many other recommendation approaches besides low-rank matrix factorization. In this section we address restricted Boltzmann machines (RBMs), due to their growing popularity and the fact that

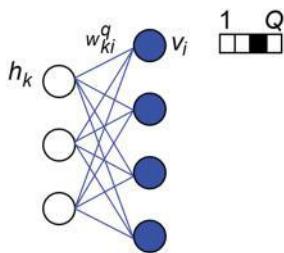


Figure 9.2 Restricted Boltzmann Machine for One User

they represent a distinctive and highly competitive approach. An RBM is a two-layer neural network with stochastic units. The name “restricted” comes from the fact that the units must be arranged in a bipartite graph, as shown in Figure 9.2. Units in the visible layer are connected only to units in the hidden layer, and vice versa. The connections are undirected, meaning that the network can operate in both directions, with the hidden units exciting the visible units and vice versa.

In the recommendation setting, there exists a separate RBM for each user u . Assuming that user u has rated m items, there are m visible units in the corresponding network.

The output of the k -th hidden unit is typically binary and is denoted h_k with $k = 1, \dots, K$. The output of the i -th visible unit is v_i with $i = 1, \dots, m$. For recommendation, v_i is usually ordinal with discrete values between 1 and Q . The q -th value of v_i , denoted v_i^q , has a certain probability of being activated (i.e., the network estimates a probability distribution over all possible values of v_i). The connection between the k -th hidden unit and the q -th value of the i -th visible unit is associated with a weight w_{ki}^q . To avoid clutter, bias terms are not shown in Figure 9.2. The dependence on user u is also omitted.

The units are stochastic in the sense that the output obeys a probability distribution conditioned on the input rather than being a deterministic function of the input. The hidden units are binary with

$$p(h_k = 1 | \mathbf{v}) = \sigma\left(b_k + \sum_{i=1}^m \sum_{q=1}^Q v_i^q w_{ki}^q\right)$$

where

b_k = a bias term

$\sigma(x) = 1/(1 + e^{-x})$ = a sigmoid function

The ordinal visible units follow a softmax rule:

$$p(v_i^q = 1 | h) = \frac{\exp(b_i^q + \sum_{k=1}^K h_k w_{ki}^q)}{\sum_{l=1}^Q \exp(b_l^l + \sum_{k=1}^K h_k w_{ki}^l)}$$

with bias term b_i^q . All user-specific networks share connection weights and bias terms when multiple users rate the same item, but both hidden and visible units have distinct states for each user.

Contrastive Divergence

The parameters of an RBM are the weights and biases. These are learned by maximizing the marginal log-likelihood of the visible units, given by

$$p(V) = \sum_h \frac{\exp(-E(V, h))}{\sum_{V', h'} \exp(-E(V', h'))}$$

with the term $E(V, h)$ representing the “energy” for the network configuration. This has the expression

$$E(V, h) = -\sum_{i=1}^m \sum_{k=1}^K \sum_{q=1}^Q w_{ki}^q h_k v_i^q + \sum_{i=1}^m \log Z_i - \sum_{i=1}^m \sum_{q=1}^Q v_i^q b_i^q - \sum_{k=1}^K h_k b_k$$

where Z_i is a normalizing constant. Learning is performed by gradient ascent of $\log p(V)$. The updates for the weights are

$$w_{ki}^q \leftarrow w_{ki}^q + \epsilon \frac{\partial \log p(V)}{\partial w_{ki}^q}$$

$$\frac{\partial \log p(V)}{\partial w_{ki}^q} = \langle v_i^q h_j \rangle_{data} - \langle v_i^q h_j \rangle_{model}$$

where $\langle \cdot \rangle$ denotes the expectation operator. The biases follow similar updates. The term $\langle v_i^q h_j \rangle_{data}$ is equal to the frequency with which the binary quantities h_j and v_i^q are simultaneously on when the network is being driven by the data set D , meaning that the visible units v_i^q are clamped to the values in the training set. The term $\langle v_i^q h_j \rangle_{model}$ is

the expectation with respect to the distribution of v_i^q defined by the learned model, and it is far harder to calculate. For this reason, an approximation is used. In the Monte Carlo–based contrastive divergence method, the approximation is

$$\langle v_i^q h_j \rangle_T \approx \langle v_i^q h_j \rangle_{\text{model}}$$

where $\langle v_i^q h_j \rangle_T$ denotes the expectation over T steps of a Gibbs sampler.

As seen, for example, in the Netflix competition, RBMs tend to perform well for cases where matrix factorization has difficulties, and vice versa. For this reason, a successful approach consists of utilizing both a matrix factorization recommender and an RBM in tandem, thus providing a combined prediction.

ASSESSING RECOMMENDATION QUALITY

The best metric for evaluating a recommender system depends on the problem at hand. In the majority of applications, the root mean square error (RMSE) is the measure of choice. The RMSE is defined as

$$RMSE = \sqrt{\frac{1}{n} \sum_{(u,i) \in D} (x_{ui} - l_u r_i)^2}$$

and it measures the numerical discrepancy between the actual and predicted value for x_{ui} . Moreover, the RMSE is directly related to the data fit term in Equation 1, which means that it is the metric being optimized in most matrix factorization schemes. A similar relationship exists with the log-likelihood objective optimized in RBMs.

In some situations, the rank ordering of the recommendations may matter more than the specific predicted values for x_{ui} . For such cases, when a list of the top r recommendations is desired, it may be preferable to use a measure derived from information retrieval, such as the mean average precision at r , denoted $MAP@r$. This is defined for a set of N_u users as

$$MAP@r = \frac{1}{N_u} \sum_{u=1}^{N_u} AP_u @ r$$

$$AP_u @ r = \frac{1}{\# \text{relevant}_u} \sum_{k=1}^r P_u @ k \times I_{u,k}$$

in which $AP_u @ r$ is the average precision at r for user u . The precision at k , written as $P_u @ k$, is the fraction of items in the top k results that are relevant for user u . The indicator variable $I_{u,k}$ is equal to 1 if item k is relevant for user u , and 0 otherwise. The total number of relevant items for this user is denoted $\#relevant_u$.

The $MAP@r$ is difficult to optimize directly. For this reason, approximations and upper bounds are often used. It is known that optimizing for the RMSE and then sorting the items in descending order of predicted x_{ui} is equivalent to minimizing one such upper bound.

RECOMMENDATIONS IN ACTION: SAS LIBRARY

The library at SAS keeps records of users, books, book requests, and renewals. As an application example, we applied low-rank matrix factorization in order to predict whether a given user will request or renew a particular book. The data set contains 8180 records pertaining to 2212 users and 6763 books. The number of renewals ranges from 0 to 23. We have added 1 to the number of renewals, in order to differentiate between books that have only been requested once and missing data. Hence, the “rating” in the postprocessed data set ranges from 1 to 24.

Note that the observed entries constitute an extremely sparse subset of the unknown full matrix X ; only $\frac{8180}{2212 \times 6763} \approx 0.05\%$ of X is observed.

We utilized $K = 20$ factors and ran stochastic gradient descent (following Equation 2) for 40 epochs using a randomly selected fraction of the data set training. The remainder was held out in order to assess testing error. The computation took only a few seconds on a standard desktop computer. The resulting RMSE is plotted in Figure 9.3.

Rather than trying to predict the number of renewals for a user/book pair, which is not necessarily a useful task per se, it may be of interest to recommend a list of r books to a given user u . In order to do so, it is possible to compute estimates $\hat{x}_{ui} = l_u \cdot r_i$ for all books i and sort the list of \hat{x}_{ui} in descending order. The top r entries correspond to the recommended books. Note that we do not report the $MAP@r$ metric, since that would require a reliable mechanism for inferring whether each recommended book is relevant for each user.

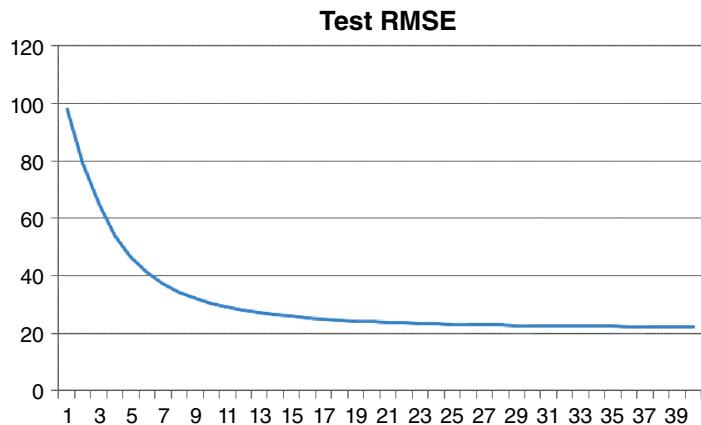


Figure 9.3 Root Mean Square Error on a Held-Out Test Set from SAS Library Data

As a concrete example, we examine a specific user in the data set. The random subset of entries utilized for training is shown in Table 9.2. The top 10 recommendations generated for this user by the matrix factorization model are shown in Table 9.3. It is apparent that the model has captured the user's interest in linear regression and econometric themes.

Table 9.2 Training Entries for a Specific User

Book ID	Text
660	Logistic regression examples using the SAS system.
3890	The complete guide to option pricing formulas / Espen Gaarder Haug.
6304	The geometry of multivariate statistics / Thomas D. Wickens.
4691	Generalized additive models / T.J. Hastie, R.J. Tibshirani.
4674	Survival analysis : techniques for censored and truncated data / John P. Klein, Melvin L. Moeschberger.
20101	Practical methods for design and analysis of complex surveys / Risto Lehtonen, Erkki Pahkinen.
21353	Bayesian ideas and data analysis an introduction for scientists and statisticians / Ronald Christensen, Wesley Johnson, Adam Branscum, Timothy Hanson.
13287	Matrices with applications in statistics / Franklin A. Graybill.
6214	Generalized linear models / P. McCullagh, J.A. Nelder.

13628	Kendall's advanced theory of statistics. Vol. 1, Distribution theory / Alan Stuart, J. Keith Ord.
3476	Sampling of populations : methods and applications / Paul S. Levy, Stanley Lemeshow.
4675	Data : a collection of problems from many fields for the student and research worker / D.F. Andrews, A.M. Herzberg.
18210	Generalized linear models for insurance data / Piet de Jong, Gillian Z. Heller.
18971	Computer intensive statistical methods : validation model selection and bootstrap / J.S. Urban Hjorth.
18924	Applied linear statistical models : regression, analysis of variance, and experimental designs / John Neter, William Wasserman
19765	Sampling : design and analysis / Sharon L. Lohr.
19765	Sampling : design and analysis / Sharon L. Lohr.
20649	Bayesian methods : a social and behavioral sciences approach / Jeff Gill.
23286	Analysis of health surveys / Edward L. Korn, Barry I. Graubard.
21148	Design and analysis of experiments with SAS / John Lawson.

Table 9.3 Top 10 Recommendations for the Specific User

Book ID	Text
6139	Econometric analysis / William H. Greene.
4165	The SAS system information delivery for the pharmaceutical industry.
5182	Estimation and inference in econometrics / Russell Davidson, James G. MacKinnon.
3951	Communication skills to inspire confidence / Barrie Hopson, Mike Scally.
19741	Statistical programming in SAS / A. John Bailer.
1011	Generalized linear models : a Bayesian perspective / edited by Dipak K. Dey, Sujit K. Ghosh, Bani K. Mallick.
6444	The Dilbert principle cubicle's-eye view of bosses, meetings, management fads & other workplace afflictions / Scott Adams.
23234	Multiple comparisons and multiple tests using SAS / Peter H. Westfall, Randall D. Tobias, Russell D. Wolfinger.
23305	Generalized linear models and extensions / James Hardin, Joseph Hilbe.
20712	Mathematical statistics / Jun Shao.

CHAPTER 10

Text Analytics

Unstructured data, of which text data is a major part, is one of the three major sources for the data volume explosion that has occurred in the last dozen years.¹ Nearly all of your communication is now in a digital format from email to tweets and blogs.² I was even able recently to change my phone plan, purchase a different high-speed Internet service, and correct a billing problem all via instant message chat session. Even when the communication happens via phone, it is likely being converted to a text format for storage and further potential analysis.

Text data, no matter the origin, presents challenges to process it and convert it from a raw form to make it suitable for modeling.

Working with text, or unstructured data, is one of the two major reasons that this area offers such a competitive advantage in many

*The data analysis for this chapter was generated using SAS Text Miner software, Version 13.1 of the SAS System for Windows. Copyright © 2013 SAS Institute Inc. SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc., Cary, NC, USA.

¹The other two sources are machine-to-machine data and images (which includes videos).

²This includes voice calls because they are digital and can be programmatically converted to text.

markets. It is sometimes very difficult to manipulate the data into a usable form. The recommended steps for getting data ready include:

- **Identify the problem you are trying to solve.** This may sound simple, but many projects fail because they do not have a clear scope and outcome and then they drift out of control until they are never heard from again.
- **Identify the potential sources of your data.** If this is purely text analytics, then sources will be unstructured. If this is predictive modeling, then this will likely include both structured and unstructured sources.

INFORMATION RETRIEVAL

Information retrieval is a needed action in almost any text analytics activity. Several activities are common tools in performing information retrieval:

- **File crawling.** File crawling is the process of recursively moving through files in order to extract the useful information for a later purpose. A common example is the archive folders created by Microsoft Outlook or folders for documentation or memos. The file types that have been identified, by either the crawler or the user, are noted and tagged for later information retrieval activities like text extraction.
- **Web crawling.** Web crawling is similar to file crawling except the activity takes place on the Web. Many people are familiar with the term “bots,” software applications that perform automated and repetitive tasks. Scraping text from Web pages is one application of a bot. The Internet, because of its open standard protocol, makes conversion of Web pages to text straightforward. With so much of the human-generated data being text, Web crawling is an essential operation for getting information about what customers are thinking for competitive intelligence, managing brand reputation, and assessing community awareness to organizational initiatives.
- **Text extraction.** Text extraction is normally the step directly following Web or file crawling. It normally involves separating

the text from the formatting of a document or Web page. An example would be the extraction of plain text from a Portable Document Format (PDF) file. Text extraction removes all of the formatting, font, and size type of information and leaves only the plain text for analysis.

- **Index and search.** In this age of the Internet, index and search are frequent occurrences. The information retrieval services must build indexes that can be used to facilitate effective search. In 2008, Google on its official blog said it had hit the milestone of identifying 1 trillion unique URLs at one time. With a quality index in place, users can then search not only for a single term but also perform a facilitated search that includes rules like Boolean logic.

CONTENT CATEGORIZATION

Content categorization is the process of evaluating documents and suggesting or assigning categories based on their content. This can be done with several methods, but the most common is the bag-of-words method. It works much as it sounds. The words from a document are figuratively all placed into a bag and then they are drawn out in sets. The likelihood of words occurring together helps in classifying documents into categories based on the co-occurrence of words. Consider words like “river,” “stream,” and “bay”; they are likely to be found together in documents about water, but they might also be found in documents published by the Environmental Protection Agency (EPA) about pollution and contamination levels.

There are two main methods within content categorization: text topic identification and text clustering. In text topic creation, the words that make a document unique are compared to the words from other documents, and a similarity score is calculated. As more documents are measured and determined to be similar, topics emerge organically, without predefined direction or rules. Every document is given a score for every text topic, and then a cutoff value is applied to identify which documents belong to a particular topic. Documents can be found in multiple topics. Referring back to the water words in the previous paragraph, if there was topic on boats and EPA pollution, that document

might be found in both topics because of the unique words. Text clustering, in contrast, limits documents to only one cluster. The clustering techniques are similar to those discussed in Chapter 6 on segmentation with the prestep of needing to convert the text to condensed numeric representation. This is often done using singular value decomposition (SVD). The SVD is similar to principal components in that it represents a compact numeric representation for a set of data.

Another key component for content categorization to be effective is the dictionary and thesaurus available to reduce the considered words to those that are relevant for differentiation. This means words that are common to all the documents will be eliminated or have reduced weight for classifying topics and clustering. This can also include words commonly found in the English language, such as “and,” “to,” “the,” and so on.

In most organizations, it is necessary to obtain information about, and from, unstructured data that is created internally and externally. Taxonomists might develop the categories and concepts while subject matter experts write rules.

The process of categorizing content includes large, complex, and hierarchical taxonomies. Content categorization uses advanced linguistic technologies to identify metadata in, and about, your documents improve the business value of information technology and the corporate data that it manages. Content categorization is the automated process of classification and extraction of entities from input documents.

You will improve results on information retrieval and organization costs when all of the information created by, or within, your organization can be classified and easily retrieved.

TEXT MINING

Text mining is a discipline that combines data mining and text analytics to use unstructured or textual data along with structured data for the purposes of exploration, discovery, and predictive modeling or classification. Given a large body of textual data, it is desirable to know what main ideas are present. Being able to sift through large volumes of data requires the ability to successfully parse and filter text in order to reveal the most meaningful and significant content contained

therein. One of the major obstacles to using unstructured data is summarizing it in a form that is usable for topic discovery and predictive modeling. Like most other data mining processes, textual data can require additional preparation to be effective in subsequent analysis.

Due to the many ways words can be represented (e.g., inflection), parsing such content requires the ability to be language aware. While English words are commonly separated by spaces, a language such as Mandarin does not use spaces in this way and thus impacts how tokenization is performed. Identifying different parts of speech plays a significant role in text mining as knowing how a word is used impacts further processing. For example, the term “bank” can mean a financial institution or a riverbed, depending on the context, but can also refer to the notion of wagering. Parsing techniques such as the use of synonyms can aid in treating different words as equivalent terms while stemming can reduce all forms of a term to its root (e.g., “improves,” “improved,” and “improving” reduces to “improve”).

Filtering textual data eliminates content that is not applicable by reducing the total number of terms and documents used for analysis, all while preserving what is considered to be the most valuable and relevant content. Data can be reduced to capture a subset of documents that are relevant to a particular issue, which can significantly reduce the time and computational resources required for later analysis. Term and frequency weightings may be used to vary the importance of terms based on how frequently the terms occur in individual documents as well as across the entire collection of documents. A term may occur frequently in a given document while also occurring just as frequently in all of the other documents. Thus, this term is not a good discriminator in differentiating between documents and can be weighted accordingly. Other filters, such as spell correction, may also be required.

Learning the main themes of what a body of text is about is known as topic discovery. Through the use of various mathematical techniques (e.g., singular value decomposition [SVD], latent Dirichlet allocation [LDA]), the most significant and distinctive features of the text can be surfaced and enable seeing what terms and phrases best describe what is happening in the text. Using the same techniques, predictive modeling can take these same features and produce classifiers that can

be used on unforeseen data that can accurately predict the target outcome. Using these data-driven, statistical-based models, classifiers can be trained, tested, and validated from any collection of text.

TEXT ANALYTICS IN ACTION: LET'S PLAY JEOPARDY!

If you have gotten to this point in the book, you probably have a good a strong impression that I am a really big nerd.³ One of the nerdy behaviors I enjoy is trivia, and there is no better trivia game than watching *Jeopardy!* To those who are not familiar with the game, it is a quiz show format with three contestants and host Alex Trebek. The game takes place in a 30-minute TV time slot and has three sections, Jeopardy!, Double Jeopardy!, and Final Jeopardy!. Inexplicably my wife developed a real interest in *Jeopardy!* during her second pregnancy, which gave me a chance to indulge daily in a favorite pastime.

The Jeopardy! round presents the contestants with five distinct categories and five clues with each clue being assigned a dollar value beginning at \$200 up to \$1000.⁴ (See Figure 10.1.) Each day there

Statistics History	Less is Moore	Fischer of Men	"Box"ing	Nonfiction Writers
<u>100</u>	<u>100</u>	<u>100</u>	<u>100</u>	<u>100</u>
<u>200</u>	<u>200</u>	<u>200</u>	<u>200</u>	<u>200</u>
<u>300</u>	<u>300</u>	<u>300</u>	<u>300</u>	<u>300</u>
<u>400</u>	<u>400</u>	<u>400</u>	<u>400</u>	<u>400</u>
<u>500</u>	<u>500</u>	<u>500</u>	<u>500</u>	<u>500</u>

Figure 10.1 *Jeopardy!* Game Board

³ This is a badge I can wear proudly now that nerds are cool.

⁴ The value of the individual clues has changed over time.

are three contestants, including two new contestants and the winner from the game the previous day. The winner is on the far left from the TV viewers' perspective, and two new players are in the center and far right. The round begins with the returning player selecting the first clue.

Now, one twist on *Jeopardy!* that distinguishes it from other quiz programs is that instead of giving the contestants a question, the contestants are given an answer and they must give the question to receive credit. For example, if the category is City Name Origins, the information presented to the contestants might be "It's the place in England where the oxen forded the river Thames." After the question is read, contestants race to press a hand-held buzzer. The fastest contestant to buzz in gets the opportunity to answer. However, remember that the response must be phrased as a question for the contestant to get points. For this question, the contestant must say, "What is Oxford?" If the contestant answers correctly, the value of the clue is credited to his or her total. That contestant then has control of the board and gets to select the next clue.

If the contestant answers the question incorrectly, the value of the clue is deducted from his or her total (amounts can go negative) and the remaining contestants have an opportunity to answer. Contestants only have one attempt per question, and they are allowed five seconds to respond once they have buzzed in to respond to the clue. If no contestant answers the question, the contestant who selected that clue retains control of the board. One additional twist to the game is that in the Jeopardy! and Double Jeopardy! rounds, certain clues are a "daily double."⁵ This clue can be answered only by the contestant who selected it, and that contestant can wager any sum, up to either the current value of his or her account or \$500, whichever is greater.⁶ If contestants answer correctly, the amount they wagered is added to their total. If they answer incorrectly, the amount is deducted from their total, at which point they can have a negative account balance.

⁵ One clue in the Jeopardy round and two in the Double Jeopardy round.

⁶ The daily double has changed the outcome of many *Jeopardy!* games.

The Double Jeopardy! round is played just like the Jeopardy! round with the exception that clue values are doubled and there are two daily doubles.

The final round, called Final Jeopardy!, includes only a single question. Before the question is read, contestants are told the category, and then they must decide their wager based on their earnings to this point in the game. (Players with negative winnings after Double Jeopardy! are excluded from Final Jeopardy!).) The wager is not revealed to the audience or the other contestants at this point. After the wager is written, the question is revealed to the contestants, and they are given 15 seconds to record their response. After time expires, the response and wagers are revealed and the contestants' account balances are adjusted according to if they responded correctly to the clue. The contestant with the most money is named the winner and banks that money along with the invitation to return the next day and play again. The nonwinners are given a nominal parting gift.

Ken Jennings holds a number of *Jeopardy!* records including the most consecutive wins (74) and has earned over \$3 million on the show. Jennings and others maintain the J! Archive Web site (www.j-archive.com). This Web site has an archive of all the clues, responses, and contestants ever to play *Jeopardy!* Because of the show's long run, that amounts to about 250,000 questions.

It is because of the popularity of the game show, its approachability, and its data volume that I wanted to showcase it here to discuss text analytics methods and the power of this capability. Jeff Stewart, a friend of mine, is a former *Jeopardy!* contestant (he won a car!), and we began to engage in a discussion of how to prepare for the game show. He mentioned the fact that to maximize your preparation, you need to know which topics require deep knowledge (U.S. Presidents) and which ones require only rudimentary knowledge (Chinese philosophers). This sparked my curiosity to try to quantify what topics give you the best value for your time.

This problem, like almost all real-world analytics problems, starts with getting the data into a usable form. The text analytics suite from SAS was used for all of this analysis, but I have written the steps so that they could be generically followed independent of the software used.

Information Retrieval Steps

The first step is to collect the data from the source. In this case I used the Web crawler to recursively download the data from each game stored at J! Archive. J! Archive has a single page for each game, and that page includes a wealth of information including all the clues, their point values, who answered correctly and incorrectly, what order the questions were selected, and more.

With the data downloaded, I needed to identify the different parts of the HTML files and create metadata (data about the data) for later analysis. In the case of the *Jeopardy!* questions, the attributes I tagged were game ID, question text, subject, dollar value, answer, and so on.

In text analytics, “document” is a generic term to mean a single entity. During the analysis, “document” was at times used to refer to a game and other times to a clue.

After the entities had been tagged and extracted from the HTML files, they were assembled in an XML form in chronological order across all games. In the next code snippet, you can see how the details for a single clue are represented and the custom tags used to identify the different parts of the data.

Excerpt from XML file of Text Parsing from J! Archive

```
<clue>
    <game_id>329</game_id>
    <show_number>4782</show_number>
    <text>Sure, you studied presidents & state birds, but
        how the Beastie Boys & their "No Sleep Till" here</text>
    <value>$1000</value>
    <order_number>15</order_number>
    <correct_response>Brooklyn</correct_response>
    <is_daily_double>False</is_daily_double>
    <daily_double>-1.000000</daily_double>
    <dialogue/>
    <is_triple_stumper>False</is_triple_stumper>
    <row_index>2</row_index>
    <category>U.S. GEOGRAPHY IN SONG</category>
    -<answers>
        -<correct>
            <name>Ken</name>
        </correct>
    </answers>
</clue>
```

These steps of preparing the data were common to all the analysis questions for this example.

Discovering Topics in *Jeopardy!* Clues

Primary themes or ideas existing within a given body of text can be automatically discovered using the text analytics techniques discussed in this section. Through automatic term and document association and by using SVD techniques, the most meaningful and significant features emerge and describe the overall topics occurring in the corpus.⁷

The J! Archive data set has a number of text fields that can be used for topic discovery, but the most useful in this data is the clue text. The clue text is the most meaningful for distinguishing topics occurring over all *Jeopardy!* games. In general, the largest field yields the most textual information.

Building on the data processing information from the Information Retrieval section, topics can now be learned. As a starting point, I requested that 25 topics be created from the clues. The top four categories along with the most common terms used to define each category and how many clues were found under that topic are shown in Table 10.1.

If you have seen the show, these topics are not surprising, but the information could not have been captured by looking at the clue categories. In *Jeopardy!* the clue categories generally provide insight into the correct response. As an example here is a clue: “On ‘50s TV he played the man with badge 714.” This might be obvious to some as a *Dragnet* reference but to many it might not be. Using the category of “Jack’s in the box,” you have the additional information that the person’s name includes “Jack”; the answer in this case is of course Jack Webb. Another example is the clue “This *NSYNC song hit the top 5 in 2000; see ya!” Unless you are a major fan of boy bands, this song title is probably not on the tip of your tongue, but coupled with the category of “Say that 3 times” I think a number of people can correctly answer “Bye Bye Bye.” Other examples of categories that provide meaning for the clue response but not the topic are “When you’re having fun, “Watcha doin’,” “Rhyme hole,” and so on. (See Table 10.2.)

⁷ A corpus is a large and structured set of texts.

Table 10.1 Most Common Topics from *Jeopardy!* Clues

Topic	Common Terms	Number of Clues
Music	Song	11,264
	Hit	
	Hear	
	Group	
	Sing	
Literature	Write	10,976
	Novel	
	Book	
	Author	
	Play	
Water Geography	Large	10,144
	World	
	Island	
	Mile	
	Lake	
Television	Play	9,743
	TV	
	Show	
	Game	
	Movie	

Knowing the most popular topics asked on *Jeopardy!* is an important advantage as you study and prepare to play the game. The number of questions asked in any one game comes from a very wide body of knowledge. If you encounter a category in which you have limited knowledge, it could end your winning streak (or stop it from getting

Table 10.2 Top-Scoring Questions from the Music Topic

Clue	Category
Frank Sinatra despised this 1966 hit, calling it the worst song he'd ever heard..."Doo-be-doo-be-doo"	Song Standards
In song it's "A Name I Call Myself"	2-Letter Words
Hop-happy song: 99 B O B O T W	<i>Jeopardy!</i> Ditloids
In a 1978 song he was "Slip Slidin' Away"	Sliding
Steve Miller had a No. 2 hit with this 1977 song	Song Birds

started). Using a text topic approach, you have information on which topics are the most likely to be asked so that you can use your study time efficiently. This has common application in business as well. The ability to quickly and accurately identify the topic of a customer question either through a customer call center or electronic communication will improve an organization's chances to address the customer's specific concern and therefore improve retention and customer profitability.

These topics were used in a very raw form to show the ability of text analytics techniques to organize information. In practice, examination and refinement of the topics would likely be needed.

Topics from Clues Having Incorrect or Missing Answers

Building on the previous sections of preparing the data and identifying the most common topics, it would be useful to know if there are certain topics that contestants are not prepared for and then provide a strategic advantage in your ability to answer questions other contestants cannot. Alternatively, we might discover that contestants answer incorrectly in a somewhat uniform pattern and no advantage can be gained.

To address this question, the unstructured data of the clue must be combined with structured information about the contestants and their responses to the clues. An incorrect answer was defined as a clue when any contestant provided an incorrect answer or where all three contestants failed to provide any answer. There are around 31,000 questions (about 18%) answered incorrectly by at least one contestant.

Repeating the same analysis to identify clue topics, using this subset of questions allows the comparison to see if contestants are ill prepared for certain topics or if there are holes in the contestants' knowledge of the most popular clue topics. The two most popular topics of the questions with a wrong answer are shown in Table 10.3. The most common topic for missed questions is clues that might have multiple answers and the contestant needs to know the dates of events to select the correct response. As an example, "This single-named Renaissance artist who died in 1520 was regarded in the 19th century as the greatest painter ever." There are four single-named Renaissance painters that most people know (largely because of the Teenage Mutant Ninja Turtles), but contestants must know which one died in 1520, and if there is more than one, they must also know which was named the greatest painter ever in the 19th century.

Table 10.3 Most Popular Topics for Questions with a Wrong Answer

Topic	Common Terms	Number of Clues
Century Events	Century	2,505
	19th	
	Work	
	Famous	
	Born	
Music	Song	2,297
	Hit	
	Hear	
	Group	
	First	

This question is difficult because while Raphael is the only one of the four to die in 1520, Leonardo da Vinci died in 1519 so without a high degree of certainty it would be easy to respond incorrectly.

The percentage of questions missed varies from topic to topic but with the exception of Century Events, they are all within a few percentage points of the overall missed question rate, and there is no statistical significance. So if you're preparing for *Jeopardy!*, know your dates and study the most popular topics to give yourself the best advantage.

Discovering New Topics from Clues

In preparing to compete on *Jeopardy!*, a very important question is whether there are clues (or topics) from the early episodes that are as relevant as clues from later shows. Said another way, are the topics asked about on *Jeopardy!* changing over time, or are they stable? This type of question is also very important to organizations that deal with text because we know that fashion trends, political views, economic factors, and pop culture change over time and being able to identify new topics trending in future unforeseen data gives organizations the ability to detect these shifts. The baseline topics can be applied for discriminating what topics are continuing to be captured while allowing for new topics to emerge that are not accounted for by the existing topics. This model is ideal for yielding new topics that occur chronologically and can be scaled to account for new topics occurring at time $t + 1, t + 2, \dots, t + N$.

Baseline topics are either manually assembled by the user through providing topic names, terms, roles, and weights or are automatically learned topics that have been modified subsequently. For the *Jeopardy!* topics, all have been automatically learned. Once baseline topics have been created, they can be applied to new data for identifying what is continuing to trend while allowing for new topics to be learned automatically.

Applying this to the J!-Archive data, the baseline data was taken from the years 1995 through 2000. The top five topics from those years are shown in Table 10.4.

These five topics were then applied to data from 2005 to 2010. Making the periods nonconsecutive ensured that a topic could not have appeared at the end of one period and then faded early in the next period. Holding the topics from the first period constant and also allowing new topics to be formed preserves the ability to identify these new topics and their relative size compared to the original topics identified in the first time period. The result of the analysis show that new topics are emerging. In Table 10.5 you can see how the popularity ranking of the topics changed. The new most popular topic was Women in History with questions like “Born in 1961, this famous woman was the daughter of the Eighth Earl of Spencer” (Princess Diana).

From the results in Table 10.5, you can see that topics do change over time but not drastically. From this analysis we find that more weight should be given to topics that are from more recent games than those from 20 years ago. By following this methodology, topic trending can be performed for a variety of types of sequentially ordered data. Then exploration, discovery, and monitoring can be performed to transform this stream of unstructured information into business value for your organization.

Contestant Analysis: Fantasy *Jeopardy!*

As a final look at the power of text analytics, the information of *Jeopardy!* clues will be combined with structured data to predict the success of *Jeopardy!* contestants. The contestants' previous performances on the show and the clues that they responded to are used to predict how the contestants would perform on questions from games in which

Table 10.4 Top Five Topics from 1995 to 2000

Topic	Common Terms	Number of Clues
Island Geography	Large	2,114
	Island	
	World	
	Group	
	Mile	
Music	Hit	1,817
	Song	
	Group	
	Hear	
	Sing	
Literature	Novel	1,738
	Author	
	Publish	
	Base	
	Book	
Screen	Play	1,538
	TV	
	Series	
	Actor	
	Movie	
Famous Firsts	First	1,468
	Woman	
	World	
	Book	
	Publish	

Table 10.5 Topic Popularity Ranking for Two Time Periods

Topic	1995–2000 Rank	2005–2010 Rank
Island Geography	1	2
Music	2	5
Literature	3	9
Screen	4	13
Famous Firsts	5	8

they did not participate. Ken Jennings participated in 75 games in his continuous winning streak in 2004. He correctly responded to 2,729 questions (91% correct) while incorrectly responding to 256. Additional data was included outside of Ken's winning streak, including data from the fifth round of the Tournament of Champions,⁸ which aired on May 23, 24, and 25, 2005, and two days from the IBM Challenge, which aired February 15 and 16, 2011.

To predict if contestants will correctly respond to a clue they were never exposed to during the show, the questions they did answer, either correctly or incorrectly, must be analyzed. The clues that a contestant responded to were put into a training set. The clue text was then filtered, parsed, and clustered, and singular value decompositions were made in a number of dimensions. With the binary target variable of the contestants' responses and the SVDs from their training, many predictive modeling techniques (including all the ones discussed in Chapter 5) were used to build models with the objective to minimize the misclassification.

After building a model for Ken Jennings, Brad Rutter, and Jerome Vered (three of the players with the most longevity in the game's history), I could then predict their expected earnings for an arbitrary episode of the show that none of them participated in.

I was able to predict with very high accuracy whether the men would respond correctly or not to an arbitrary clue. Then using the questions from a show that aired when none of the contestants appeared, as shown in Table 10.6, I scored the questions using the model and calculated the expected earnings based on the probability of a correct response and then adjusted the earnings based on the percent-

Table 10.6 Projected Earnings from Three Contestants

Contestant	Projected Earnings		
	6/6/02	10/25/07	7/7/08
Ken Jennings	\$27,619	\$27,132	\$27,384
Brad Rutter	\$22,532	\$24,602	\$30,615
Jerome Vered	\$14,061	\$19,908	\$14,122

⁸The Tournament of Champions is an invitation event for past winners to compete against each other

Table 10.7 Overall Response Profile

Contestant	Correct Responses	Incorrect Responses	Correct Response	Response Rate
Ken Jennings	2,723	256	91%	62%
Brad Rutter	524	50	91%	78%
Jerome Vered	402	40	91%	43%

age of questions the contestants responded to.⁹ Table 10.7 shows the comparison of the three players with respect to the number of questions and their success at responding correctly.

It is interesting to see in Table 10.7 that these three successful contestants all had a 91% correct response rate but had such different response rates. While there is no way to know how any of these contestants would actually have performed on the show for a game they did not participate in, there are some comparisons that reinforce my projections. Ken Jennings's average daily winnings during his 74-game win streak was \$33,727 and his average final Jeopardy! wager was \$7,698.¹⁰ If you take his average final Jeopardy! wager and add it to his projected earnings, you get a sum that is within 5% of his average payout.

While this example of comparing *Jeopardy!* contestants from different periods against each other likely will never become as popular as discussion of the greatest quarterback of all time, the ability to take text and combine it with structured data to get better predictive models is still relatively new for most organizations. I hope that this example has helped you begin to think how you can use text mining in your organization to improve your data-driven decisions and use of analytics.

⁹ The expected earnings did not take into account daily doubles so for comparison assume that no one found the daily doubles.

¹⁰ Ken's highest game was \$75,000 and his lowest game was \$8,799.

PART
THREE

**Success Stories
of Putting It All
Together**

True genius resides in the capacity for evaluation of uncertain, hazardous, and conflicting information.

—Winston Churchill

Parts One and Two explained the proper tools with which to manage data and the tools for analyzing it. Part Two also discussed the techniques that give you the best opportunities to uncover new information and confirm your hypotheses about your business. In this part, I have compiled a collection of cases that illustrate companies that have been able to use this information to find business value. In some cases, this value has amounted to millions of dollars added or millions of dollars saved, but in each case analytics were successfully applied to well-stored and well-prepared data that improved the business.

I have worked with each of these companies to varying degrees. In a few cases, I served in an advisory role or gave details about the software they were using to help them efficiently achieve their business objective. In others, I was more involved and actually performed the analysis myself to help prove out the business value.

These cases come from companies in different industries and countries. In most cases, they are large multinational corporations that faced and overcame major challenges in using their data effectively. My hope is that you can use these cases as a blueprint for applying process and experience to your business challenge to make rapid headway and achieve results. Taking raw data and turning that data into valuable information is a precious skill that is highly esteemed in the marketplace and should be trained, cultivated, and acquired whenever possible.

QUALITIES OF SUCCESSFUL PROJECTS

All of the successful projects have a number of items in common. The commonalities were not similar industries, or organizational structure, or even language. First among the common elements were an executive sponsor who valued this information and appreciated that analytics could be used to gain insight from data being collected.

The business value was easy to explain to people of all levels at the company.

A commonality that I did not expect before I complied all of these cases is that the data was already being collected. I did not encounter a situation where an executive sponsor wanted the answer to a question that was not available from analyzing the existing data. This recognition brought me to appreciate that sometimes advantageous choices in the past pay dividends into the future.

CHAPTER 11

Case Study of a Large U.S.-Based Financial Services Company

The U.S. banking industry, because of government regulations and its utilization by almost all citizens, has by necessity lots of data. Some of the data is kept to comply with regulatory requirements, and some is kept to help the institution provide quality service and access to beneficial financial instruments to its customers. To sift efficiently and effectively through all the data being kept and distill useful information from that data poses a challenge for banks.

The bank under consideration serves millions and millions of customers. It frequently engages in marketing campaigns to offer both current and potential customers new relevant products and services. This case study is a marketing campaign to identify likely people to respond to an offer for a new credit card. This scenario is repeated frequently at this and many other banks and also applies to many other

companies that offer products to customers. The ability to effectively predict which potential customers will respond to a marketing offer is very valuable for several reasons. The first is that it builds customer confidence by not sending junk mail or spam. The next reason is it allows the bank to reduce its expenses by not printing and mailing unwanted materials or hiring call center staff to make unwanted phone calls. The last reason is that getting customers signed up for the services they want generates revenue for the bank.

Credit cards generate revenue for banks through several methods. This revenue is categorized as two types: cardholder fees and vendor fees. Cardholder fees include annual fees, late fees, over-limit fees, interest paid fees, and so on. Vendor fees are transaction fees paid by vendors for each use of the card in exchange for guarantee of payment. As you would expect, there are significant costs associated with campaign marketing activities so being able to more accurately predict a potential customer's response to the credit card offer is very important. Better model accuracy leads to lower costs, better brand reputation, and improved customer loyalty. For this case study, the primary objective is better model accuracy.

The bank also had a second objective: improved operational processing efficiency and responsiveness. This bank has an IT infrastructure that is very common in the industry. It has several large enterprise data warehouses (EDWs) that store data about all current customers, past customers, and identified potential customers. The type of information stored in the EDWs ranges from customer account information, such as your mailing address or date of birth, to information provided by third parties, such as your credit score. There is also transactional information including each purchase you have made with a bank-issued credit card or call record for when you spoke to a customer service representative or used the automated system to pay your bill or check your account balance. The set of EDWs are, for this case, a set of massively parallel processing (MPP) databases.

TRADITIONAL MARKETING CAMPAIGN PROCESS

The marketing campaign follows a set of serialized steps that are well established and operationally defined. They are described next.

Gather the data. A campaign begins once a proposal is approved from the business executive. At that point a request is sent from the marketing department to its IT business partner. The request is to acquire data from the EDWs for all current and potential customers that adhere to certain criteria for the purpose of building a response model. The IT department gathers the data using several tools (most are Structured Query Language (SQL) based) as they extract the relevant information and aggregate it to the account or household level. Since marketing is done at a household level, information must be rolled up or aggregated to that level of detail.

At any given time, there are many credit card campaigns going. Since much of the same information about customers is needed in each campaign, it is operationally more efficient to create a standard base table for modeling on a regular interval—monthly in this case—so that many teams are able to use the same base table and then customize only the portion of data specific to a particular campaign. Having multiple teams use the same data provides for consistency, reduces the number of copies of the data and thus saves storage space, and creates an economy of scale for the creation of the base modeling table. The economy of scale comes from having one central group do the data preparation work once. Many model-building groups take advantage of the central data preparation instead of numerous different staff members doing essentially the same task for different marketing campaigns. The other economy of scale comes from the specialization that comes when you break up the data preparation from model building so that you do not require individuals to be experts in both domains. The consistency comes in through the treatment of the variables. One example is binning of continuous variables. Binning is a very common technique to simplify computations and to make interpretation easier. Age is a common variable that has high predictive power in likelihood to respond to certain marketing campaigns and also has special treatment for certain ages of potential clients.

Here is an example to further explain why binning variables can be a useful practice and consolidating that task leads to operational efficiency and better control:

The U.S. government in May of 2009 passed the Credit Card Accountability Responsibility and Disclosure Act of 2009, or Credit

CARD Act of 2009 for short. This act defines, and largely limits, the type of marketing activities that are permitted to people under 21 years of age. If the age variable were left as a continuous factor, it would require postprocessing to eliminate those potential customers from the model and at the same time introduce bias (because the people under 21 were used to fit the model but are not able to receive marketing offers). If the age variable is binned in this way—under 18, 18–20, 21–29, 30–34, and so on—we can simply filter prospects that fall into the ineligible bins.

After the IT department has completed its process of preparing the data, the modeling teams are notified.

Creating the model. In order to create a predictive model, you must identify the customers' attributes you know that help predict the outcome. The input data (containing the customer attributes or independent variables) will be a mix of interval, nominal, and binary variables. The number of customer attributes that will be considered initially to predict the binary response target of "respond to offer" or "no response" will be hundreds to sometimes thousands of variables. The modeler begins by reviewing the relationship of the response target and the customer attributes. This comparison will be done using correlation statistics and/or simple plots, such as histograms or bar charts. An investigation of missing values often is performed at this point and a plan put into place to address them (or a conscious choice to not address them).

Once the modeler has some initial assessment of the relationships between candidate variables and the target variable, the modeler will then split the data into mutually exclusive partitions. One partition will be used for building the model. This is often referred to as the training set. The other partition, the validation set, is for checking the generalized fit of the model. The validation set is used to protect against overfitting. Overfitting the model will lead to poor model generalization, a lower response rate from customers, and ultimately less revenue for the bank. After the data has been divided into partitions and initial exploration has been completed, the variable selection and model fitting begins in earnest. Building a predictive model is a highly iterative process with the goal being to get the simplest model (least number of customer attributes) that best predicts those who will respond to the

offer without overfitting. The reasons this process is iterative are because many statistical and machine learning techniques can be used to fit a model, and each technique has a number of parameters that can be tuned in the model-building phase. There is no silver bullet technique that will always give the best model, although some techniques are better than others.

The final phase in this step is the assessment of all the candidate models to pick a best model. Many measures can be used to determine the “best” model. Some of these measures are averaged squared error (ASE), misclassification rate, and profit. It often happens that a technique will dominate the various measures, but situations exist where one modeling technique will appear best with one measure and another technique will appear best for another measure. Along the way, some candidate models are discarded, but almost always there are at least a few techniques that need to be evaluated in this final step. At this point, supervisory team members or managers are involved in the review process to make the final model selection.

Scoring the model. The marketing department is then given the model logic so that it can be applied to the entire potential customer table. Selecting a tool with integration between the modeling and the scoring steps can save the organization considerable time and money. With proper integration, the model logic can be applied to all customers in minutes; if the model logic has to be converted from one technology to another and then tested and verified, it can be weeks because of the manual involvement.

Rank your potential customers. The potential customers would then be ranked by their likelihood to respond or by the profitability they would bring. This ranking is as simple as creating a new column in the potential customer table. Once this column is populated with a customer’s likelihood to respond or predicted profit, the table is then sorted in descending order so that the most likely respondents are on top.

Operationalize your campaign. The operational aspects of addressing, sending customized email, or placing outbound calls to potential customers does not begin until the budget for that campaign has been exhausted. A last step and best practice is to review the results

of the marketing campaign and access how effective the model was in meeting its objectives.

The process just outlined has challenges when it is implemented in real-world situations. For example, during the steps of data assembly from the EDWs, model creation, and model application and scoring, the underlying customer data was changing. Customers were applying for new credit cards, missing car payments, changing their contact policies, moving to new states, and other possibilities from a near-limitless list of things that happen. These events that are external to the marketing campaign require human reconciliation, which is costly and slows down the process. The typical turnaround on a marketing campaign from the initial data pull to offers being sent to customers is a month.

The model lift, a metric to describe the predictive power compared to a random baseline, was between 1.5 and 2 at the second decile (20% mark). This means the selected final model was 1.5 to 2 times better than a random model when looking at the first fifth of the data. To put it another way, if you have 1000 potential prospects and 50 are going to respond to your campaign, if you take a 20% random sample, you would get, on average, 10 responders in the group of 200. After using the model to help you select who will be in the campaign, you expect to get 15 to 20 responders in the same 200 people.

The model lift in this case was restricted by the time window allowed for the modelers to work. As mentioned previously, the model-building process is very iterative, and modelers get better results as they have time to iterate through the model-building process. It is well known that the more time you give to building your model, the better the model will be.

HIGH-PERFORMANCE MARKETING SOLUTION

The financial institution adopted a big data computing appliance approach using a MPP database. This had two advantages.

1. It has a much larger aggregate memory and computational capacity than any single machine could provide. With more memory and computational power, the bank could use more

historical data and more candidate variables than previously, which allowed it to find patterns that were not discovered in the samples of the data it used previously.

2. It minimized data movement. With the decreased data movement and the additional computational power available, the bank was able to build the same models in minutes instead of many hours as it had been done before. One of the most dramatic improvements was for a model that had taken 19 hours to run now took 3 minutes to run. With this additional time, the modelers at the bank were able to try more statistical and machine learning techniques to search for a more effective model to predict who would respond to the campaign. They were also able to tune the models more. The use of additional techniques and greater refinement of the parameters resulted in a model with better lift—it was better at identifying those people who would respond to the marketing campaign.

VALUE PROPOSITION FOR CHANGE

The value proposition for this bank can be described in several ways: accuracy and precision, productivity for employees, more efficient processes, and increased revenue or return on investment.

The bank took advantage of the combination of the increased number of modeling techniques it could try, the increased time it had to tune each technique, and the increased data it could use for creating the model, and it was able to move the model lift from 1.6 to 2.5. This is a significant increase in model accuracy. Referring back to our 1000-customer example, now instead of finding 10 responders with no model or 16 with the traditional modeling approach, with this new high-performance appliance, the bank was able to identify 25 responders in the 200 people selected in the top 20%. This technology increased model accuracy.

With the decrease in model-building runtimes, the bank's analysts were able to complete their work in a shorter time, which allowed them to work on modeling marketing campaigns they otherwise would have not been able to do. The time-per-model-run went from

one day to one hour; that translates into less staff or more models being produced without increasing staffing. This increased employee productivity.

Because of the reduced time to create the model, the amount of reconciliation and human intervention needed at the end of the campaign preparation phase was reduced, which had an additive effect of reducing the time to produce a marketing campaign and the human expense to create it. This made the process more efficient.

Because this process was more efficient, took less time, was more accurate, and took less staff, it is better in every measurable way. By computing the savings of staff time, the lifetime value of each acquired customer, and the improved brand reputation through better public perception about marketing efforts (i.e., people feel the bank sends less spam), the bank was able to quickly realize a positive return on investment measured in months.

CHAPTER 12

Case Study of a Major Health Care Provider

The health care industry is seeing an increase in the data it has available and the expectation from customers and shareholders that this data be used to improve customer satisfaction, improve patient care, and return a higher value to shareholders.

The increase in data comes from several sources. The first is from electronic medical records. In years past, the information available to health care providers would be a claim form (either paper or electronic) that was processed to reimburse physicians for their services. Then prescription drugs were added so that some initial analysis could be made and help save patients from potentially harmful drug interactions. But in this era of electronic medical records, information far beyond the billing codes can be processed and analyzed. Physician notes, radiology reports, and transcribed phone calls to on-call nurse hotlines are all now potentially available. A major challenge to data of this type is that it is largely unstructured, which poses potentially significant challenges in productively using this data for improved decision making.

In fact, it is estimated that nearly 80% of all data is unstructured, and medical records can be some of the hardest to work with. Due to its own set of shorthand and abbreviations, extra care must be taken in organizing this information.

This case study is illustrative of the win-win opportunities that exist for both businesses and consumers. When businesses harness the data at their disposal and turn that into actionable information, customers get a better product—in this case better health benefits and proactive resolution of concerns—and businesses can reduce their expenses and increase their revenue, leading to substantially increased profits. This health care provider administers a large number of Medicare Advantage plans with many participants across the United States.

Medicare Advantage is a health insurance plan that is offered by private providers to add benefits, or provide an advantage, over the standard Medicare plan. Medicare is a standard health benefit package that covers older adults and those with disabilities since 1965. Medicare Advantage has more than 41 million participants, just over one in eight people living in the United States. The U.S. government pays the private provider a set monthly fee per member. Members may have a premium to pay out of pocket depending on the plan type and coverage they elect. Medicare Advantage has been an ongoing program since 1997 with a revision in 2003. Under the Patient Protection and Affordable Care Act of 2010 (or as it was referred to in the most recent U.S. presidential election, Obamacare), the Medicare Advantage plans have recently seen considerable change.

The changes made during the act of 2010 emphasized the quality of the plan and making data about quality more available and transparent to participants and the public at large. Each plan receives a star rating from one to five stars. One star is the lowest rating, and five stars is the best rating. A three-star plan is average. The Centers for Medicare and Medicaid Services (CMS), a government agency within the Health and Human Services Department, is responsible for assigning the ratings. The star ratings are aggregates of five different ratings systems: CAHPS (Consumer Assessment of Healthcare Providers and Systems), HEDIS (Healthcare Effectiveness Data and Information Set), HOS (Health Outcomes Survey), IRE (Independent Review Entity),

and CMS. Brief explanations of some of the individual rating systems are included next.

CAHPS

This program is a multiyear initiative of the Agency for Healthcare Research and Quality to support and promote the assessment of consumers' experiences with health care. First launched in October 1995, it aims to address a range of health care services and meet the information needs of health care consumers, purchasers, health plans, providers, and policy makers. The goals of the CAHPS program are twofold:

1. Develop standardized patient surveys that can be used to compare results across sponsors and over time.
2. Generate tools and resources that sponsors can use to produce understandable and usable comparative information for both consumers and health care providers.

The various CAHPS surveys ask consumers and patients to report on and evaluate their experiences with health care. These surveys cover topics that are important to consumers and focus on aspects of quality that consumers are best qualified to assess, such as the communication skills of providers and ease of access to health care services.

HEDIS

HEDIS consists of 80 measures across five domains of care. HEDIS measures address a broad range of important health issues. Among them are the following:

- Asthma medication use
- Persistence of beta-blocker treatment after a heart attack
- Controlling high blood pressure
- Comprehensive diabetes care
- Breast cancer screening
- Antidepressant medication management
- Childhood and adolescent immunization status
- Childhood and adult weight/body mass index assessment

HEDIS is designed to provide purchasers and consumers with the information they need to reliably compare the performance of health care plans.

HOS

The goal of the Medicare HOS program is to gather valid and reliable health status data in Medicare-managed care for use in quality improvement activities, plan accountability, public reporting, and improving health. Managed care plans with Medicare Advantage contracts must participate.

IRE

IRE is an appeals process for those who feel they have been unfairly treated or denied services owed them under their plan benefits.

In using these assessment instruments, the Medicare health plans are rated in five different categories

1. Stay healthy: screenings, test, and vaccines
2. Managing chronic conditions
3. Member experience with the health plan
4. Member complaints, problems getting services, and improvement in the health plan's performance
5. Health plan customer service

As you can see, members' feelings about their health plans are very important components to the star rating. The star rating is also critical to reimbursement of the plan for services it provides. Here is how the process works.

For each county in the United States, a benchmark value is established for the Medicaid and Medicare monthly cost of care. This benchmark varies from county to county due to a number of factors. Medicare Advantage plans then bid on the cost to provide coverage for Medicare parts A and B benefits (the original coverage) for the eligible participants of that county. If the bid amount is under the benchmark value (which it usually is), then the plan providers receive a percentage of the difference between the bid price and the benchmark value

back from the county in the form of a rebate. This provides incentives to the plans to provide the required standard of care at the lowest price—all plans are required to cover all the tenants of Medicare part A and B. The rebate must be used to provide extra benefits to the enrollee.

The rebate is the first place that the star rating affects revenue. The rebate amount a plan receives is modified by the star rating, where five-star plans receive a larger rebate than a three- or four-star plan with the same bid. Here is an example: If the benchmark for Mesa County, Colorado, is \$300 per month and plan XYZ bids \$280, it receives 67% of the difference back in a rebate $(\$300 - \$280) \times .67 = \$13.40$ per participant. This rebate is adjusted based on the star rating of plan XYZ. If it is a five-star plan, then the rebate is 73% (or a rebate of \$14.60) instead of \$13.60. In addition to increasing the rate of the rebate reimbursement, the benchmark value is adjusted so the adjusted benchmark for Mesa County for a five-star plan is \$315 ($\300×1.05), which results in a total rebate of \$25.55. The difference between the original rebate and the adjusted rebate is termed the bonus. For this example, plan XZY will receive a bonus of \$12.15 per member. Here, having a five-star plan almost doubled the profit per member as compared to a three-star plan.

The bonus is paid for with funds designated in the Health Reform Law and also from funds established for a demonstration period. These two sources of funding total about \$3.1 billion in bonus allocation for 2012.

The reward for having a five-star plan increases during the period of 2012 to 2014 as the bonus percentage increases in each year as this new quality component is phased in. In addition to additional revenue, the CMS is further adding emphasis on customer satisfaction by allowing customers to move from their current plan, if it is rated less than five stars, to a five-star plan at any time. Before, customers normally could change plans only during the annual open season.

This health care provider has plans rated above average. Given the potential revenue that could be generated from having a five-star plan, it made good business sense to invest in using the unstructured data it was already storing to help determine customer satisfaction and intervene as needed. The early investment in producing five-star plans is a virtuous cycle that can lead to billions of dollars in additional revenue over the course of several years.

This company already has a significant investment in analytical talent and a number of predictive models to help inform decision makers, but there was a wealth of unstructured data, most in the form of text, which was not being used. This was seen as a great opportunity to use the textual information recorded by the customer support representatives (CSRs—the operators who answer the phone when customers call with questions) to aid in identifying those customers who were unhappy or needed some additional assistance in resolving their issues or problems.

All parties can be proud of their success in addressing this problem. I have aging family members who participate in Medicare Advantage, and I want them to be happy with their health care experience. The company wants to have happy participants because it engenders goodwill, improves customer loyalty, and reduces the overhead costs to administer the plan (less correspondence to send and less call center staff, etc.), which in turn allows the company to offer better benefits at the same rate or equivalent benefits at a lower rate. The third reason is that the reimbursement rate from the U.S. government is determined in part by the customer satisfaction rating that each plan gets.

Almost every plan member is satisfied with this plan provider, so identifying dissatisfied customers is difficult because they are rare (which was discussed in Chapter 4) but one that is crucial as the plan administrator wants all plan members to be pleased with their coverage.

More satisfied patients stay with their current plan. Satisfied patients encourage primary care and specialist physicians to accept plan terms.

The business challenge faced was that in the current environment, only a small portion of the records could be modeled. A great source of information—the transcripts of phone calls with the call center—was not available in the modeling process.

The business problem was to improve customer satisfaction. To do this, more records had to be used to improve model accuracy and more attributes, namely CSR notes taken during a call, needed to be considered in the modeling process. The implemented solution was an MPP database that was used as a distributed in-memory modeling appliance. The training data set was 169 million records with each record being about

5 kilobytes in length. This length consisted of hundreds of fields but also several large text fields. The process followed five steps:

1. **Gather the data.** This consists of federating data from several enterprise data warehouse (EDW) systems. In addition to the EDW, there is data from transactional systems, MS Access databases, Excel spreadsheets, flat files, and several other sources. This represents some of the true issues with solving data mining problems. This pain point is easier than it would have been in previous attempts because of the ample storage space provided by the MPP database and the parallelism in the loads. For this proof of concept, the data used is all of the relevant structured data that could be gathered and the text fields that contain the CSR notes from call centers throughout the country. The IT department in conjunction with analytical staff performs this function of gathering data together for analysis.
2. **Load the data.** With a single table containing all the modeling features created, that table was loaded in and distributed into the MPP database. The size of the data file was about 800 gigabytes. Using the distributed loading capabilities, the table loaded in about 20 minutes. This is a significant improvement over previous attempts using different technology.
3. **Model the data.** There are a number of reasons that customers might call the health care administrator: billing issues, access to health care—the doctor they want to see is out of network or none of the in-network providers have timely appointments. Customers also call because of medication issues—the medication that their doctor prescribed is not available in a nonformulary (non-name brand, such as statin versus Lipitor®). For each of the general reasons that customers call, models need to be created. Each class of model might have several subcategories that have models associated with them. The predictive models have some shared attributes but unique attributes as well.

To make use of the text data, a singular value decomposition (SVD) is then performed. For those unfamiliar with SVDs,

they are a factorization of a matrix analog to an eigenvalue or principal component. The SVDs combined with the structured variables from the EDW were used to create models to predict the likelihood that a customer would be dissatisfied. The models do not decay very quickly compared to models in other industries, and they often remain valid for many months. The biggest reason models decay is due to changes in plan benefits. Each year participants have an open season during which they can switch plans. Plan administrators must notify customers of any plan changes in advance of the open season so that customers can make an informed choice. The models are sensitive to these plan changes, and significant effort is spent to identify similar situations in the historical data and estimate the correlation and potential impact. The reduced time required to build models has led to a more tactical approach to model building, because the decay point of the model is known in advance and the likely drivers are also known. The analytics team is able prepare for the event and then take early information from the first few days of customer calls to build a relevant model. This avoids the lengthy process of determining surrogates in the historical data and then running simulations to predict customer behavior. The team is able to monitor the response and then make adjustments. This leads to better models and a more efficient use of the modeling team's time.

The health care provider remarked during this process that without this high-performance appliance, it would have never attempted to model a problem of this size; it just would not have been feasible. A number of modeling techniques mainly in the family of logistic regression and neural networks were considered, and multiple iterations to each were attempted to get the highest-quality model. These iterations, because they were running in a distributed in-memory appliance, were done in minutes where days would have been required previously. This reduction in time for building each model iteration allowed for many more iterations of each model and the opportunity to consider additional modeling techniques. This

combination of considering more techniques and having more chances to refine each technique will always lead to better models.

4. **Score the models.** With the models built, each call record is scored by each model to create a probability score for a customer being dissatisfied. The patient's scores from each model are then sorted, summed, and ranked. The summing is done because a customer may be scored as dissatisfied by multiple models, and that aggregate score is relevant.
5. **Apply to the business.** Two of the most powerful features that have taken root in this organization are (1) the predictive power of the models and their ability to identify situations where intervention is warranted and (2) the ability to group search terms to present all the relevant data to the business user. For example, there are more than 20 different spellings for the term "grievance" (I spelled it wrong in several drafts myself) in the CSR notes. Historically, business users would have to repeat their search multiple times for each variation (and hope that they guessed them all). Then business users would be required to make aggregate graphs based on what they saw or try to decide which spelling best approximated what they saw for the entire population. With this high-performance appliance, search terms can be collapsed and managed at a global level to give users the complete information they need.

The high-performance appliance is used primarily to provide insight into the massive amount of data that this health care provider is responsible for and also to test business analysts' hypotheses on why certain situations are occurring with customers.

This proof of concept was so successful that funding was immediately secured to make the project permanent. The end goal is to have every customer call transcribed and every customer touch point scored. Additionally each call would be scored for sentiment and classified into categories based on the contents of the call. All of this is possible now when it was only wishful thinking a few years ago.

This predictive modeling process to predict patient satisfaction has been able to help the health care provider better focus its efforts on those patients who need additional help. This increased effort and attention ensures patients better medical care and hopefully improved medical outcomes. It also allows the health care provider to spend human capital resources where they are needed most and reduce costs while returning a better value product for customers and higher earnings for shareholders. In the future, this single program has the ability to affect millions and millions of dollars in profit annually through achieving and maintaining a five-star rating.

CHAPTER 13

Case Study of a Technology Manufacturer

The manufacturing line is a place where granular analysis and big data come together. Being able to determine quickly if a device is functioning properly and isolate the defective part, misaligned machine, or ill-behaving process is critical to the bottom line. There were hundreds of millions of TVs, cell phones, tablet computers, and other electronic devices produced last year, and the projections point to continued increases for the next several years.

FINDING DEFECTIVE DEVICES

To visit the manufacturing facility of the latest and greatest electronic manufacturers is to almost enter a different world. People wearing what looks like space suits, robots moving in perfect synchronization, all as machines roar and product moves through the many steps from raw material to parts and then finally to an inspected flawless finished product.

When a failure is detected in the manufacturing process, there is an immediate need to identify the source of the problem, its potential size, and how to correct it. Doing this requires analyzing thousands of potential variables and millions to even billions of data points. This data, in addition to being very large in size, is also very time critical. There is a finite window of time after a problem is discovered to stop the defective batch of product from leaving the factory and then also to address the root cause before more time and material are wasted. This efficient treatment of defective product also helps to maintain good brand reputation, worker morale, and a better bottom line.

HOW THEY REDUCED COST

The typical path of resolution for this type of modeling for this customer is outlined in the next six steps.

1. The first step is the inspection process for the product. Each product has a specified inspection process that is required to ensure quality control guidelines are met and the product is free from defects. For example, determining if a batch of silicon wafers meets quality control guidelines is different from determining quality control acceptance for a computer monitor. In the case of the wafer, the result is essentially a binary state: The wafer works or it does not. If it does not work, it is useless scrap because there is not a mechanism to repair the wafer. In the case of the computer monitor, it could be that there is a dead pixel, or a warped assembly, or discoloration in the plastic, or dozens of other reasons the monitor does not pass quality control guidelines. Some of these product defects can be remediated by repeating the assembly process or swapping out a defective part; others cannot be remediated, and the monitor must be written off as waste. In these complex devices, humans are still much better at visual inspection to determine defects in the display (for more details see the Neural Networks section in Chapter 5) than teaching a computer to see. The human eye is very developed at seeing patterns and noticing anomalies. In this manufacturing environment, the human quality control

team members can find more reasons for rejection than the computer system by an order of magnitude.

If a product batch does not meet the quality control threshold, then an investigation is initiated. Because the investigation slows or stops the ability to manufacture additional items, an investigation is not done on a random basis but only after a batch of product has failed quality control inspection. This investigation is expensive in terms of time, quantity, and disruption, but it is essential to prevent the assembly of faulty products.

2. Once a batch of wafers, for example, has failed quality control, an extract of data from the enterprise data warehouse (EDW) is requested. This data is sensor data over the last few days from a number of different source systems. Some of the data is stored in traditional massively parallel process (MPP) databases; other data is stored in Hadoop and still other in neither Hadoop nor MPP databases. This is a complex process that involves joining a number of tables stored in a relational schema of third normal form and denormalizing them to a rectangular data table along with matching them to MPP tables. The process to manufacture a single wafer can include several thousand steps and result in more than 20,000 data points along the assembly line process. When this data is transformed and combined with millions and millions of wafers manufactured every day, you have truly big data. The assembled rectangular data table is of a size that would defy the laws of physics to move for processing in the allotted time window of a few hours.
3. The data is then clustered. With such wide data (more than 10,000 columns) it is clear that not all will be responsible for the defects. The columns can be removed and clustered based on several criteria. The first is if the column is unary, or only has one value. With only one reading, the column holds no useful information in determining the source of the defects. The second is multicollinearity. Multicollinearity is a statistical concept for when two or more input variables, the measurements from the sensors in this case, are very highly correlated. Multicollinearity does not affect the final predictive power of the model, but it does needlessly increase the computational cost

of building the model because the model considers more terms than it really needs. Multicollinearity can lead to parameter estimates that are highly influenced by other input variables and thus are less reliable. This clustering step removes the useless variables and those that do not add additional predictive power because of their relationship to other input variables.

4. An equipment path analysis is now performed to try to identify the potential source of the problem. This analysis can be done using several techniques, but decision trees are often a popular choice for their high interpretability and simple assumptions. The problem historically has been the difficulty of using the needed volume of data in the required time window.

Decision trees at a basic level attempt to split a set of data into the two purest groups by evaluating all the candidate variables and all the candidate split points. The calculation of the best split must be communicated, summarized, and then broadcast (in the case of a distributed environment) before the next split point can be determined. Because of this communication after each step, additional care in constructing efficient techniques to build decision trees is required in a parallel and distributed environment.

After the best split is found, it must be evaluated to see if the split adds to the overall explanatory power of the tree. At some point in building almost every decision tree, the algorithm reaches a point that additional splits do not improve the results but needlessly complicate the tree model. This falls under the principle of parsimony. Parsimony is a principle that if two models (decision trees in this case) have equal explanatory or predictive power, then the simpler model is better. If the equipment path analysis yields suspect equipment, it is investigated. Historically, the decision tree has proven to be very effective at uncovering the systemic failure—a machine that is out of alignment or that has an inaccurate thermometer. It does not deal well with detecting interaction causes of defects. These are often termed random events because they are not attributable to a single cause and therefore cannot be easily diagnosed and fixed. An example of this might be that

in step 341, a particle of dust interfered with encoding for item 123. That failed encoding caused a delay for item 113 on step 297, which resulted in an excess heat registered to sector A4 of the wafer. Because of the excess heat in sector A4, the chemical bath in step 632 did not adhere fully, which then caused a . . . (you get the idea). This type of complex reaction is not visible using decision trees. So other techniques specific to time series data—functional dimension analysis (FDA) or symbolic aggregate approximation (SAX) (see Chapter 8 for more details)—are being evaluated. This should remind us that even though dramatic improvements were achieved, there is still more research and work to be done to improve the process.

5. A root cause analysis is also performed. This is to identify the problem in procedure or process that leads to the error and the least common set of variables that need to be assessed to prevent the problem. With all the measurements, attributes, and metadata, taking every data artifact into account is time consuming and impractical. The least common or vital few analyses are done using a variable importance measure to determine what additional check can be added to prevent a repeat of this failure.
6. The results from both steps 3 and 4 are sorted, ranked, and analyzed for frequency. After this information is summarized, a report is issued with detailed timeline, problem, recommendations, and next steps.

In this case, the customer used the distributed file system of Hadoop along with in-memory computational methods to solve the problem within the business parameters. This process had been developed, refined, and improved over several years, but until the adoption of the latest distributed in-memory analytical techniques, a problem of this size was considered infeasible. After adopting this new infrastructure, the time to compute a correlation matrix of the needed size went from hours down to just a few minutes, and similar improvements were seen in the other analysis steps of an investigation. The customers are very pleased with the opportunity they now have to gain competitive advantage in high-tech manufacturing and further improve their process and their position in the market.

CHAPTER 14

Case Study of Online Brand Management

In this modern information age, business success is closely tied to your success on the Internet. This is especially true of manufacturers. Online brand management is a challenging, fast-paced environment where ideas or perceptions that once took weeks can evolve in hours. The popular microblogging site Twitter reported in the fall of 2012 that it was now seeing 500 million tweets per day on the site. A quick search for this large company revealed thousands of tweets from their customers. With each five-second refresh, more than 20 new tweets would arrive almost around the clock. It is impossible for any human or any department of humans to keep up with all the customer comments about a company from just this one microblogging service. Now add updates from other popular social media sites like Facebook, LinkedIn, and others, and staying on top of consumers' thoughts on your product can appear to be an impossible task. The amount of information that must be processed to keep an accurate view of the brand's value and then be

prepared to promote, defend, or anticipate changes in the brand can seem to be a daunting task.

A number of analytical techniques can be used to help tame this task. Here we will talk about Twitter, but these six techniques could be used for almost any human-generated feed of data.

1. Identify the language in which the user is commenting. Language detection is an important first step because it can help determine the size and scope of the issue or praise. By and large, Twitter users only use the site in one language with a few using two (almost always their mother tongue plus English). Identifying the language also helps to identify a location if the GPS locations are not available. (In my experience, this is of minor benefit since most tweets are made from smartphones, which include the location in their metadata.)
2. Look for sentiment. Sentiment analysis is a fantastic tool in managing a brand because it helps you focus in on those customers who are unhappy, the aspects of the product that cause the most problems, or those users who can be of help as advocates for your brand. The true power of sentiment analysis is that it not only informs you how customers feel about your product but it tells you how strongly they feel it. It also sheds light on if they feel this particular way about the entire product or just some aspects of the product. For example, consider an LCD TV display. This is a complex product with many individual features to like or dislike in addition to an overall sentiment. By first identifying the language, we can see that there are a higher-than-normal set of negative posts from Brazil, which might lead to an investigation of the translation quality of the user's manual or it might point out a product flaw that when the TV is left on for a large number of hours (like during the World Cup), the color saturation changes and it is harder to discern details in the content.
3. Categorize the content. There is some debate that I think boils down to personal preference about doing sentiment analysis before content categorization or vice versa. My personal preference for this order is that it allows me to use sentiment as a

- filter for the most emergent issues, upset customers, and then categorize those feedback items.
4. Apply business rules to the situation. Once we have identified the sentiment of particular users and the categories that trouble or please them, we must now use the established business process for resolving their concern or leveraging the positive feedback in future activities. If current business processes do not include details on how to react in these types of situations, then those business processes will need to be established, but that is a separate discussion that will not be dealt with in this book.
An example of a standard business process for positive feedback is to retweet or like the comment so that users following you will see the positive comment and hopefully increase their brand loyalty and think more positively about your brand. In the case of negative comments, the options might be to send a private message to the customer, send them a list of the nearest retail stores so that they may exchange the product, or send a link to the frequently asked question (FAQ) page where this issue is addressed. The action does not need to be grand, but to be effective, it should be targeted to the customer's specific issue and it should be promptly after the complaint is registered. If this is done successfully, you will secure a more faithful following from this user, and the user is likely to influence those in their network of influence toward more positive sentiment about the brand.
 5. Create models that can be used as a feedback loop. With the details that have been provided by the user, we can join that information with the structured data that already exists and create models that solve a number of useful purposes. We can incorporate positive feedback into updated sales forecasts and as triggers to suppliers for more parts. Negative feedback can be studied for product quality and issues with manufacturing processes or customer service training or usability design. As is often the case, in this step we can learn more from our critics than our advocates. While no one wants negative feedback, it is often the best source for product improvement.

6. Monitor the process. Once this process of identifying sentiment and moving comments into categories and following business rules for dealing with the feedback are in place, there are opportunities for continual improvement in this area just like any other. Key metrics for this area are the number of communications issued, average time to respond, and number of escalated concerns. It is also a worthwhile endeavor to look for correlation of your brand success and your online efforts. I doubt a causal link can be established in most cases, but showing efficacy in your efforts is the best way to keep funding and show the additional value your work brings to the company.

During this case study the focus is on the things that you can do to monitor your brand and product portfolio. The same ideas can be employed in trying to gain market share from your competitors. There are few times people are more willing to change habits (using a particular brand is a habit) than at the point directly after they have just expressed frustration with that habit. Your mileage will vary on the point depending on the goods or services your brand sells. Services are usually easier to displace than durable goods. But always being on the lookout for opportunities to take market share from competitors is a tried and true way for business to thrive.

CHAPTER 15

Case Study of Mobile Application Recommendations

With the rapid growth of smartphones over the past five years, a new market for smartphone applications has emerged and with it stiff competition for mind share. All vendors in this space desire the ability to make recommendations on the applications (apps) that users will like. The goal is to ensure that recommendations are not seen as spam (unwanted solicitations) but instead as great advice, thus moving from simply suggesting content to a highly prized role as a trusted advisor. The business problem in this case is very straightforward: Given the apps users have on their mobile phones, what apps are they likely to use? This problem poses several challenges, the first being the size of the data. With hundreds of millions of cell phone users and each one being almost unique in app purchases, finding good recommendations is difficult. The other issue is the level of detail about the apps. For this business challenge, the data was a set of binary variables. The binary state could be defined a number of different ways: Did they have the application installed? Had they ever used the application? Had the application been used in the last time period?

Table 15.1 Sparse Customer and Item Data Sheet

Customer	Item1	Item2	Item3	Item4...	ItemN
Customer1	1				1
Customer2			1		
Customer3				1	
Customer4...	1				
CustomerN		1			

Regardless of how you frame the problem, the output is a probability to purchase the app (the purchase price may be free) based on a set of binary input variables.

This problem is divided up into several stages, each requiring different skill sets. The first stage is data collection and aggregation. Traditional predictive modeling through data mining generally requires that input data be rectangular, and missing values pose severe problems for some algorithms. In this case, having a list of potential applications to recommend and all possible apps that could have been purchased as variables and all the customers as rows in the data (see Table 15.1) creates a very sparse data set. (There are lots of missing values.) Normally imputation would be an approach to deal with these missing values, but because the data is so sparse and the variables are binary, imputation is very difficult without creating bias and skewing results. This high level of missing values makes neural networks and logistic regression poor techniques for this type of problem. Decision trees can better handle missing values, but they tend to be unstable. The techniques of factorization machines or stochastic gradient descent generally behave better for these types of problems.

The second problem is data size. Because making recommendations while considering only a subset of the data is suboptimal, the quality of the recommendations is called into question. Therefore, sufficient computing hardware, especially main memory, must be allocated for the problem size along with software that is designed to perform in a parallel and distributed environment. The parallel nature allows all the CPU cores on the machine to be utilized, and the distribution property allows for multiple machines to work together solving the problem. This distributed nature also allows for expansion of your

hardware as the problem size increases—this often happens after a successful implementation: New, larger problems emerge that, if they can be solved, will provide additional business value and are therefore very desirable.

The parallel and distributed design of the software also allows for a critical component: speed. It is not useful to solve a problem in a day if the answer is needed in a few seconds (what most people are willing to wait) and the answer cannot be precomputed. This problem is often overcome by doing offline training. The recommendation model is trained and refined using historical data and then deployed so that new events—a user coming to an app store, for example—is scored and apps are recommended. This pattern of offline training and model scoring solves the problem of quick recommendation but introduces the concern about the useful life of a model, or model decay. Having the ability to quickly train, deploy, and retrain is the ideal situation for recommendations that need to be near real time or faster. Alternatively, a service-based architecture could be used where the model is trained and then held in memory. When a recommendation is needed, an application program interface (API) is employed with the new record, and a recommendation is made using a content-based filtering or collaborative filtering method. These methods produce a small subset of the entire collection of apps available in the virtual store based on criteria and then update the overall table so that a future recommendation request will use all available information.

For this case study there were about 200 apps to be considered. The first step was to look for clusters that would help reduce the problem space. This clustering reduced the number of input variables from 200 down to a few dozen. After the clustering was completed, the cluster variables were added to the list of potential input variables. Other techniques included singular value decomposition, and principal component analysis to find relationships between apps. After the enrichment of the data with the additional features were created, variable selection techniques, both supervised and unsupervised, were used to eliminate variables that did not provide useful information. This was followed by multiple iterations of different tree models. The trees were boosted and ensembles were created in a variety of ways to

produce powerful and stable models. The models were evaluated against a holdout sample that had been partitioned earlier—not randomly, as is typical, but by a time window. To randomly sample would have biased the recommendations because users were in both training and validation data, but with a time partition the model could be tested under real-world conditions, because once the model is developed and deployed, it must be updated with an active learning paradigm or retrained before excessive decay.

CHAPTER 16

Case Study of a High- Tech Product Manufacturer

Semiconductor wafer fabrication is very competitive. Companies compete on cost, quality, and delivery time to market. In the age of digital information, a large amount of data (e.g., process data, equipment data, and lots of historical data) has been automatically or semiautomatically collected, recorded, and accumulated for monitoring the process, diagnosing faults, and managing the manufacturing process. Decision makers may potentially use the information buried in the raw data to assist their decisions through data mining for possibly identifying the specific patterns of the data.

However, in the high-tech industry of semiconductor manufacturing, many factors that are interrelated affect the yield of fabricated wafers. Engineers who rely on specific domain knowledge cannot find possible root causes of defects rapidly and effectively. (See Figure 16.1.)

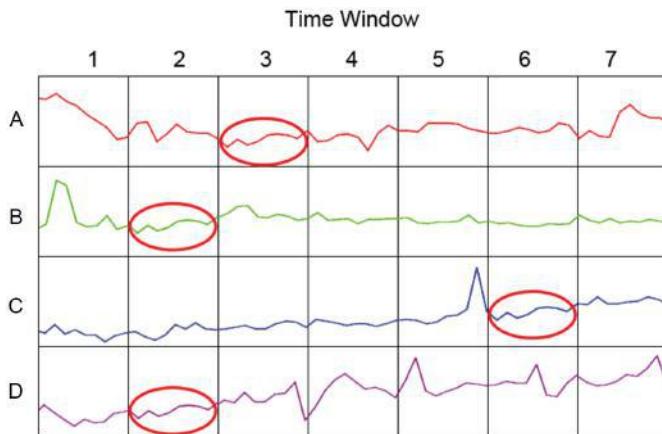


Figure 16.1 Time Series Similarity

In addition, when conducting similarity analysis for the time series data, the company also needs to identify and report separately the similarity according to the definable time windows; users are allowed to adjust and define these time windows when necessary. For example, in Figure 16.1, the similarity of time window 1, 5, 6 in time series A has a stronger similarity pattern than the others. Companies can decompose these time series similarities in different time windows for different machines.

HANDLING THE MISSING DATA

The collected semiconductor data often include missing and inconsistent data. Depending on the sources of the data, some of the yields in some processes are captured and measured only by sampling (for particular LOT ID). And some of the yields measured with particular LOTs which will be combined or split to other LOTs in the manufacturing processes.

Data preprocessing is required to improve the quality of the data and facilitate efficient data mining tasks. In particular, missing values are replaced or deleted; and in addition to some statistical way of missing data imputation, business rules are also adopted for handling the

missing data. Some of the new variables combining machine number with date are generated.

The causal relationships between the machines of specific processes and the yield rate are examined and investigated by engineers. Different statistical methods are used to identify possible root causes (e.g., the specific machine where the failure occurred). Semiconductor companies have been using decision trees to identify the root causes of the equipment path and other vital variables. By running the >2000 times including >5million devices data, the target variable is the equipment condition (good or not good), based on the equipment path (e.g., equipment ID, Step ID. . .).

In addition to the decision tree models for root cause analysis, the company has been exploring and validating different techniques for identification of root causes, including partial least square regression and gradient boosting. In the latest proof of concept, it requested consideration of the time dimension; the time series similarity matrix was used to investigate the pattern of processes (and machines) that leads to low yields. Clustering to automatically choose the best number of cluster groups can be done using cubic clustering criterion to estimate the number of clusters using Ward's minimum variance method, k -means, or other methods based on minimizing the within-cluster sum of squares.

Requirements focused on analyzing/processing "events in motion" for the machine/equipment-processed data. Continuous queries were made on data in motion (with incrementally updated results) and with the high volumes (more than 100,000 events/sec) stream processing was used to identify events via the time series similarity or "kernel library" events.

APPLICATION BEYOND MANUFACTURING

After successful results are realized in the manufacturing division for the company, methods and techniques will be rolled out to other parts of the company to allow them to take advantage of the big data analytic architecture. Some examples of planned projects are:

- Usage pattern analysis with log data from manufactured electronic devices

- Setup strategy in business to consumer device with big data
- Social media analysis
- Brand reputation
- Monitoring new trends in technology
- Call center performance
- Retail market quality
- Claim risk monitoring

CHAPTER 17

Looking to the Future

As I look down the road to where I see big data, data mining, and machine learning in the future, I see many opportunities and some serious challenges. The biggest challenge that will need to be addressed is the balance between data privacy and reproducible research. I was asked to address this topic in a question-and-answer session during an invited talk at the Joint Statistical Meeting for the American Statistical Society in 2013. It is a difficult topic with no clear path forward. How we balance user privacy and reproducible research results is a decision that will impact all of us both as consumers and also as data users. It requires us as a society to define what our right to privacy actually includes. This balance of competing interests includes questions of legality as well as technology. Regardless of the outcome, there will be significant impacts on organizations in these ways: to comply with research standards, adhere to legislation, and fund enforcement of the laws.

REPRODUCIBLE RESEARCH

As my fourth-grade daughter can repeat from memory, the scientific method is: formulate a hypothesis, then test your hypothesis through an experiment, and finally analyze your results to determine if you can reject the hypothesis. A key outcome of a scientifically performed study is the reproducibility of the results. Unfortunately, this basic expectation has become increasingly rare. *The Economist* in October 2013 detailed the difficulty in getting reproducible results and suggests explanations for why it has become so hard to reproduce research results. One example cited that less than half of 238 biomedical papers published in 84 different journals could be reproduced. In another study, Amgen, a U.S. drug company, tried to replicate 53 landmark studies in the science of cancer; it was able to reproduce just 6. The U.S. government and many other developed nations around the world spend a significant amount of money to fund research to advance science in a number of disciplines. For example, the National Institutes of Health, which is part of the U.S. Department of Health and Human Services, spent over \$25 billion in research funding in fiscal year 2012. What is our return on investment if some high percentage of that research is not actually advancing the various fields of study?

Tighter controls and accountability need to be put into place so that people and organizations are held accountable for their published research findings so that advances in areas of public good can build on top of a firm foundation, not quicksand.

PRIVACY WITH PUBLIC DATA SETS

The ability to identify people's behavior even before they might be aware of it does leave me with mixed emotion. Consider the story that *The New York Times Magazine* did on Target and its success in identifying women who are pregnant in order to take advantage of the opportunity to change purchasing habits. On one hand, I am proud to be part of a group that is advancing science and practice to the point that machines can detect patterns and positively identify these otherwise unobservable traits. On the other hand, I value my privacy and I am cautious about the information I volunteer to organizations, fearing

that it will be shared with others I did not authorize. I look forward to the day when junk mail is no longer junk but exactly the ads I wanted. That day has certainly not yet arrived.

Arvind Narayanan is an associate professor in the Department of Computer Science at Princeton University. In a paper written in 2008, he outlined two strategies for taking data sets that contain microdata and combining that data with publicly available data to deanonymize a large data set.¹ In his example he uses the Netflix Prize data set, but the same could be done for a data set with DNA sequencing of a cancer drug study or any other large data set.

The first strategy is to match a few known items about a particular individual with the anonymous data. With this match, the entire set of attributes about the individual is revealed. By matching the ratings made on the publicly available IMDB ratings Web site and the Netflix data, individuals could be identified. Once a user was identified by matching between the two sources, the Netflix data revealed their full viewing history. Now you might be familiar with the Netflix Prize and think: Netflix only used a sample of the records, so how can you be sure you have the right person? When the information was matched, it produced a result that was 28 standard deviations away from the mean compared to no more than 2 standard deviations for every other customer.

Another interesting finding is that it required relatively little information about the movies watched to accurately identify an individual. This has to do with the sparse nature of the data; people watch relatively few movies compared to the size of the database. It turned out that knowing just six movies that a person watched allowed for a 99% match rate when the date of viewing was known within 14 days. This is data that could be easily overheard in the corporate lunchroom.

These research findings show how little thought most people have given to the privacy concerns in the big data era. It illustrates the point that additional information and computing power make it possible to deanonymize large data sets just like playing Sudoku, where there are a few known elements to start and through reason and deduction

¹ Microdata is information about specific people. Often it does not include unique identifiers but purchases, health histories, or viewing preferences.

more cell values can be identified, which in turn helps to identify even more cell values until all the cell values are known.

Contrast this lack of reproducible results with the danger of releasing large sets of big data that contain personal information. This will be a lively debate and will reduce the amount of crowd-sourced research and scientific advancement that can take place.

THE INTERNET OF THINGS

The idea of the Internet of Things was first presented by Kevin Ashton in 1999. Kevin Ashton, co-founder and executive director of the Auto-ID Center, is credited with coining the term the “Internet of Things.” Ashton said he first used the term in a presentation in 1999 he made to Procter & Gamble on how to link the company’s RFID supply chain technology to the Internet. Ashton was a marketer for Procter & Gamble (P&G) in the cosmetics area and was very frustrated that each time he surveyed products in stores, a number of items were out of stock. After many presentations to management at P&G, he was appointed to MIT as the executive director of the Auto-ID Center. The center was given \$20 million in funding to reduce the cost of radio-frequency identification (RFID) tags. The funding came from many of the major retail brands, including Coke, Home Depot, Wal-Mart, Kraft, P&G, and so on. With reduced costs of RFID and by creating a situation where suppliers were required to use RFID for every pallet and shipment, machines were beginning to generate data that was sent to other machines (M2M communication). This idea that machines talk to machines without human involvement has also contributed to the occurrence of the big data era. The power of the RFID tag is to always know where a shipment is located. This reduces lost product due to theft, spoilage of perishable products like produce, and automated inventory replenishment systems. The power of RFID tagging is that retailers can now make truly personalized offers based on the items you have in your cart while you are still shopping.

As the Internet of Things continues to develop and mature, data volumes will continue to proliferate and more and more machines will be talking to one another on our behalf. Data generated by humans will fall to a small percentage in the next ten years. Capturing the

relevant machine data and using it effectively in your organization will provide the biggest competitive advantage in the data mining space over the coming decade.

In the next ten years, the number of smart meters will increase, and smart cities will be in the planning process. There are already commercial appliances like the Nest thermostat that use machine learning algorithms to optimize your energy usage. The Nest is controllable via smartphone app or the Internet, and it provides an element of gamification. This trend will grow and grow, and I am excited to let computers help me make efficient choices in areas of energy, water, and time conservation.

SOFTWARE DEVELOPMENT IN THE FUTURE

Software will continue to be developed in two parallel tracks. One track is software created by commercial vendors that will offer routine updates, technical support, and quality control assurance. The other track is the open source community where work is done mostly by volunteers and a few paid staff. These two tracks will continue, but increasingly the commercial software will need to improve integration with open source software. I remember when the Linux operating system was rising in popularity, and there were those who predicted that Microsoft would be extinct in a few years. There was also the camp that felt that Linux was a hobbyist project and no long-term value would ever come of it. Looking back now, everyone can see that neither group was right or wrong. Today Microsoft still holds a 90% market share over the desktop market and Linux controls 25% of the server market. Clearly neither group is going away. Application software will face the same challenges. Users want a more responsive software experience that gives them easy access (no complicated installs), high availability to needed computing power, and is easy to maintain. The familiarity with instant Web search, wireless connectivity, and the adaptation to the online life has made us all impatient. Software makers will need to respond by ensuring that software enables our style of work rather than frustrating it. Several years ago when I became heavily involved in distributed computing at SAS, I was amazed at how quickly I became accustomed to quick response times. I was working on a modeling problem for a large U.S. bank, and the software was taking about

five hours to build a model. When I ported the modeling activity to a distributed computing environment, the model fitting took about three minutes. At first, I was in awe and got giddy each time the models fit in the three- to four-minute range. After a few days, however, if the model took three minutes forty five seconds to complete, I became anxious and started to wonder what was wrong with the system or what other person had consumed the resources that I felt possessive of. This has happened to an entire generation of users. An organization that fails to build performance into its product will become extinct.

Open source projects, especially those supported by the Apache Foundation, will become the standard for software. Commercial vendors in the business-to-business market will continue to exist and thrive because organizations need the regular updates, reliability, risk mitigation, and support that commercial software organizations offer. You will continue to purchase software from vendors just as you do today, but the nature of how those organizations develop software will change to incorporate more open source software. I am not suggesting that all vendors will be like Red Hat, which distributes and extends the open source operating system Linux, but instead open source tools will be the standard within these organizations. The consolidation to certain standards benefits the employees because their skills are transferable from one company to another. Employers benefit because a strong talent pool exists with the needed skills. In this market consolidation, open source will be king. The price, access, and ability to improve are all attractive qualities that ensure that open source software will dominate the software markets to a much larger degree than it currently does. Open source software will be used routinely in more aspects of business operations than it is today, but commercial vendors will still provide products and generate revenues because of their ability to optimize the software and establish more control over the user experience, which leads to a more effortless user experience and higher customer productivity.

FUTURE DEVELOPMENT OF ALGORITHMS

The science of modeling will march forward at an even faster rate than it has already. New algorithms will be developed that in special situations will perform better than stable algorithms that have been around for

many years. With the exception of the occasional breakthrough, big data practitioners will need to be well versed in the traditional methods and be capable of evaluating new challengers in objective ways. One thing is certain: The number of algorithms claiming huge advances will increase much faster than the ones that actually move the industry forward. If you refer back to Chapter 5, most of those ideas are really quite old. The newest algorithm inventions will build from these great ideas and show incremental improvement for the next five to ten years. At that time there will be a revolutionary idea that will make a huge step forward in the area of prediction. My personal research interest is in the ensemble tree and deep learning areas. The tools to prototype and express advanced techniques will also continue to improve. DARPA, in early 2013, announced that it would be promoting a new programming paradigm for managing uncertain information. This paradigm will be called Probabilistic Programming for Advanced Machine Learning (PPAML).

As the science of predictive algorithms continues to advance, better predictions will be available. Machines will become better and better at finding patterns in data and skilled software engineers will find more efficient ways to implement these algorithms, the need for human interpretation and oversight will not go away. In the labor-intensive industry of automobile assembly where robots are used heavily in the process. It is humans who have to maintain the equipment and inspect it for faults. No matter how advanced, the computer will not be able to replace the human mind for problem solving and imagination. Computers are not capable of thought; they are capable of information recall. Let us consider Watson, the question-and-answer computer built by IBM, that defeated the best of the best in *Jeopardy!*. Watson is a modern marvel and combines the ideas of big data and predictive analytics as well as any machine I am aware of. Watson was named after the IBM founder, Thomas J. Watson. The system is a cluster of ninety IBM Power 750 servers with a total of 2,880 processor cores and memory storage up to 16 terabytes of RAM. In addition to its impressive size, it is also fast. It can process 500 gigabytes of information or the equivalent of 1 million books per second. With 80 teraflops,² it would place in the top 100 of supercomputers in the world. Watson needed so much memory

² A teraflop is 10^{12} floating operations per second. Flop is a measure of computing speed.

because in order to match human information retrieval times for *Jeopardy!*, hard disk drives would be too slow. Watson uses a number of algorithms to deduce meaning from the questions posed to it in a natural language format. This is a huge step forward for computers that can understand nuance and context. Watson is loaded with dictionaries, novels, taxonomies, ontologies, newspapers, magazines, blogs, and so on—pretty much any electronic media that can be loaded.

As part of an in-depth feature from *The New York Times* in advance of Watson playing on *Jeopardy!*, the computer's processing was described as follows:

The great shift in artificial intelligence began in the last 10 years, when computer scientists began using statistics to analyze huge piles of documents, like books and news stories. They wrote algorithms that could take any subject and automatically learn what types of words are, statistically speaking, most (and least) associated with it. Using this method, you could put hundreds of articles and books and movie reviews discussing Sherlock Holmes into the computer, and it would calculate that the words “deerstalker hat” and “Professor Moriarty” and “opium” are frequently correlated with one another, but not with, say, the Super Bowl. So at that point you could present the computer with a question that didn’t mention Sherlock Holmes by name, but if the machine detected certain associated words, it could conclude that Holmes was the probable subject—and it could also identify hundreds of other concepts and words that weren’t present but that were likely to be related to Holmes, like “Baker Street” and “chemistry.”

In theory, this sort of statistical computation has been possible for decades, but it was impractical. Computers weren’t fast enough, memory wasn’t expansive enough and in any case there was no easy way to put millions of documents into a computer. All that changed in the early ‘00s. Computer power became drastically cheaper, and the amount of online text exploded as millions of people wrote blogs and wikis about anything and everything; news organizations and academic journals also began putting all their works in digital format. What’s more, question-answering experts spent the previous couple

of decades creating several linguistic tools that helped computers puzzle through language—like rhyming dictionaries, bulky synonym finders and “classifiers” that recognized the parts of speech.¹

The development of algorithms will continue as more success is found in data mining and leveraging analytics within organizations. The law of diminishing returns will apply to all but a select few breakthroughs in the next ten years. This means that most developments will be only slightly better than existing methods.

IN CONCLUSION

The term “big data” may become so overused that it loses its meaning, but the evolution of hardware, software and data mining techniques and the demand for working on large, complex analytical problems is here to stay.

About the Author

Jared Dean (Cary, NC) is a Director of Research and Development at SAS Institute. He is responsible for the development of SAS's worldwide data mining solutions. This includes customer engagements, new feature development, technical support, sales support, and product integration. Prior to joining SAS, Dean worked as a Mathematical Statistician for the U.S. Census Bureau. He holds an MS degree in computational statistics from George Mason University.

When not working, he spends time with his wife and four children. Their favorite family activities include travel, sports, enjoying good food, cheering for the Broncos, and playing games at home.

The author can be reached via these means:

jared.dean@gmail.com

@jaredldean



Appendix

NIKE+ FUELBAND SCRIPT TO RETRIEVE INFORMATION

```
import groovyx.net.http.ContentType
import groovyx.net.http.RESTClient

def accessToken = '17269d534424f87dbfc99c6ac1ab6f4'
def nikePlusClient = new RESTClient('https://api.nike.com/me/sport/')

def activities = []

def fetchAgain = true
def offset = 1
def count = 10
while (fetchAgain) {
    // grab the top-level activities list
    println "Fetching top-level activities for offset: ${offset}, count: ${count}"
    def resp = nikePlusClient.get(
        path:'activities',
        contentType : ContentType.JSON,
        headers: [
            appid : 'fuelband',
            Accept : 'application/json'
        ],
        query:[
            access_token : accessToken,
            offset : offset,
            count : count
        ]
    )

    assert resp.status == 200

    resp.data.data.each() {
        println "Fetching details for: ${it.activityId}"
        def activityResp = nikePlusClient.get(
            path: "activities/${it.activityId}",
            contentType : ContentType.JSON,
            headers: [
                appid : 'fuelband',
                Accept : 'application/json'
            ],

```

```
query: [
    accessToken : accessToken
]
)

assert activityResp.status == 200

// add this data for this activity
activities.add([
activityResp.data.activityId,
activityResp.data.startTime,
activityResp.data.activityType,
activityResp.data.metricSummary.fuel,
activityResp.data.metricSummary.distance,
activityResp.data.metricSummary.duration,
activityResp.data.metricSummary.steps
])
}

// determine if we need to fetch another page
if (null != resp.data.paging.next) {
fetchAgain = true
offset += count
} else {
fetchAgain = false
}
}

// write activities to a file
new File("C:\\nike_plus.csv").withWriter { out ->
// write the header
out.writeLine("activityId,startTime,activityType,fuel,distance
,duration,steps")

// write each activity
activities.each {
out.writeLine("${it[0]},${it[1]},${it[2]},${it[3]},${it[4]},${
it[5]},${it[6]}")
}
}
```

References

INTRODUCTION

- The Official Klout Blog. "What does Klout measure?" <http://blog.klout.com/2011/12/what-does-klout-measure/>
- IEEE. "IEEE 802.15™: WIRELESS PERSONAL AREA NETWORKS (PANS)." <http://standards.ieee.org/about/get/802/802.15.html>
- <http://www.quora.com/Big-Data/Is-Big-Data-just-a-fad> *Big Data: The Next Frontier for Competition.* (May 2011). Retrieved November 13, 2013 from http://www.mckinsey.com/features/big_data.
- Davenport, Thomas H., and Jeanne G. Harris. *Competing on Analytics: The New Science of Winning.* Cambridge, MA: Harvard Business School Publishing, 2007.
- GNC Staff. "Tracking the Evolution of Big Data, a Timeline.". (2013, May 28). Retrieved June 27, 2013, from <http://gcn.com/articles/2013/05/30/gcn30-timeline-big-data.aspx>
- Hastings, D. "The Challenger Disaster." (2003). Retrieved July 4, 2013, from <http://www.core.org.cn/NR/rdonlyres/80850EB6-27D1-4A10-B422-E6383570D5F1/0/challenger.pdf>. National Cancer Institute. (2013, February 8). "What Is Cancer?" Retrieved November 19, 2013 from <http://www.cancer.gov/cancertopics/cancerlibrary/what-is-cancer> "Tamoxifen." (2013, January 1). Retrieved November 19, 2013 from <http://www.macmillan.org.uk/Cancerinformation/Cancertreatment/Treatmenttypes/Hormonaltherapies/Individualhormonaltherapies/Tamoxifen.aspx>
- Macmillan. "Tamoxifen." (2013, January 1). Retrieved November 19, 2013 from www.macmillan.org.uk/Cancerinformation/Cancertreatment/Treatmenttypes/Hormonaltherapies/Individualhormonaltherapies/Tamoxifen.aspx
- McKinsey & Company. "Big Data: The Next Frontier for Competition." (2011, May). Retrieved November 13, 2013 from www.mckinsey.com/features/big_data
- National Cancer Institute. "What Is Cancer?" (2013, February 8). Retrieved November 19, 2013 from www.cancer.gov/cancertopics/cancerlibrary/what-is-cancer
- TIOBE Software. Tiobe Index. (2013, September 30). Retrieved October 6, 2013, from <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>
- United States Golf Register. (2013) Retrieved June 26, 2013, from <http://www.usgolfregister.org/faq.asp>.

CHAPTER 2

- Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." (2004). OSDI. Communications of the ACM, 50th anniversary issue, 1958–2008 Volume 51 Issue 1, January 2008.
- White, Tom. *Hadoop: The Definitive Guide*, 2nd ed. O'Reilly, 2012.

CHAPTER 3

- Sony Computer Entertainment Europe Research & Development Division. "Pitfall of Object Oriented Programming." http://harmful.cat-v.org/software/OO_programming/_pdf/Pitfalls_of_Object_Oriented_Programming_GCAP_09.pdf
- <http://www.statisticbrain.com/average-historic-price-of-ram>
- Dean , Jeffrey , and Sanjay Ghemawat . "MapReduce: Simplified Data Processing on Large Clusters." (2004, December). OSDI '04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA. Retrieved research.google.com/archive/mapreduce.html
- Eleftheriou, E., R. Haas, J. Jelitto, M. Lantz, and H. Pozidis. (2010, April). IBM Research. Retrieved from http://sites.computer.org/debnull/A10dec/ELE_Bulletin_Dec.pdf
- Hurwitz, Justin, and Wu-chun Feng. "Initial End-to-End Performance Evaluation of 10-Gigabit Ethernet." (2003, August). Proceedings of IEEE Hot Interconnects: 11th Symposium on High-Performance Interconnects. Retrieved from <http://public.lanl.gov/radiant/pubs/10GigE/hoti03.pdf>

CHAPTER 4

- Kelley, B. (2013, June 26). *What Are the Odds of Making a Hole-In-One?* Retrieved from About.com: <http://golf.about.com/od/faqs/f/holeinoneodds.htm>.
- USGA. (2013). *United States Golf Register*. Retrieved June 26, 2013, from <http://www.usgolfregister.org/faq.asp>.
- SAS Institute Inc. (2013). *SAS® Enterprise Miner™ 13.1: Reference Help*. Cary, NC: SAS Institute Inc.

CHAPTER 5

- Bache, K., and M. Lichman. *UCI Machine Learning Repository*. Irvine, CA: University of California, School of Information and Computer Science, 2013. <http://archive.ics.uci.edu/ml>.

- Breiman, Leo, Jerome Friedman, Charles J. Stone, and R. A. Olshen. *Classification and Regression Trees*. New York: Chapman & Hall, 1984.
- Cortes, Corinna, and Vladimir Vapnik. "Support-Vector Networks," *Machine Learning* 20 (1995): 273–297.
- Ford Motor Company. "Reducing Driver Distractions" (2012, August). Available at http://www.at.ford.com/SiteCollectionImages/2011_NA/Article%20Images/Oct_2011/Driver%20Distractions.pdf.
- Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer-Verlag, 2001.
- Hornik, Kur', Maxwell Stinchcombe, and Halber White. "Multilayer Feedforward Networks Are Universal Approximators Neural Networks," *Neural Networks* 2 (1989): 359–366.
- Miglautsch, John R. "Thoughts on RFM Scoring," *Journal of Database Marketing* 8 (August 2000): 67-72.
- McCulloch, W. S., and W. H. Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics* 5 (1943): 115–133.
- McGrayne, S. B. *The Theory That Would Not Die: How Bayes' Rule Cracked the Enigma Code, Hunted Down Russian Submarines, and Emerged Triumphant from Two Centuries of Controversy*. New Haven, CT: Yale University Press, 2011.
- Nelder, J. A., and R.W.M. Wedderburn. "Generalized Linear Models," *Journal of the Royal Statistical Society A* 135, no. 3 (1972): 370-384.
- Neter, J. E. *Applied Linear Statistical Models*. Boston: McGraw-Hill, 1996.
- Pearl, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, San Francisco: Morgan Kaufmann, 1988.
- Pollack, J. B. "No Harm Intended: A Review of the 'Perceptrons,' Expanded Edition." *Journal of Mathematical Psychology* 33, no. 3 (1989): 358–365.
- Poole, Michael A., and Patrick N. O'Farrell. "The Assumptions of the Linear Regression Model," *Transactions of the Institute of British Geographers* 52 (1971): 145–158.
- Rosenblatt, F. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psychological Review* 65, no. 6 (1958): 386–408.
- Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-Propagating Errors," *Nature* 323 (1986): 533–536.
- SAS Institute Inc. *SAS/STAT® 13.1 User's Guide*. Cary, NC: SAS Institute Inc., 2013.
- State Farm. *Driving: Through the Eyes of Teens*. (2007.) Available at http://www.cvsa.org/programs/documents/osd/StateFarm_TeenDrivingReport.pdf.

Vapnik, Vladimir Naumovich. *Estimation of Dependences Based on Empirical Data*, 2nd ed. New York: Springer, 2006.

Werner, G., and C. Modlin. *Basic Ratemaking*, 4th ed. Arlington, VA: Casualty Actuarial Society, 2010.

CHAPTER 6

Huang, Minzhang, Hai Zhao, and Bao-Liang Lu. "Pruning Training Samples Using a Supervised Clustering Algorithm," in *Advances in Neural Networks—ISNN 2010*, ed. Liqing Zhang, Bao-Liang Lu, and James Kwok. Berlin: Springer-Verlag, 2010.

Zeidat, Nidal, Christoph F. Eick, and Zhenghong Zhao. "Supervised Clustering: Algorithms and Application." (2006, June 28). University of Houston, Technical Report UH-CS-06-10.

CHAPTER 7

Lee, Taiyeong, Ruiwen Zhang, Yongqiao Xiao, and Jared Dean. "Incremental Response Modeling Based on Novelty Detection via One-Class Support Vector Machine" to be presented at the ISBIS 2014 and SLDM Conference in Durham, NC June 2014. Available at <http://www.stat.duke.edu/~banks/dcc/>.

CHAPTER 8

Al-Naymat, Ghazi. *New Methods for Mining Sequential and Time Series Data*. Saarbrücken, Germany: VDM Verlag Dr. Mueller Aktiengesellschaft & Co. KG, 2010.

Leonard, Michael, Jennifer Sloan, Taiyeong Lee, and Bruce Elsheimer. "An Introduction to Similarity Analysis Using SAS®." SAS Institute Inc., Cary, NC, 2010.

Mitsa, Theophano. *Temporal Data Mining*. Boca Raton, FL: Chapman & Hall/CRC Data Mining and Knowledge Discovery Series, 2010.

CHAPTER 9

"'Next Best Offer' Analytics: Ebay, Schwan Foods, Netflix and Others." <http://practicalanalytics.wordpress.com/2012/01/05/Analytics-case-study-schwan-foods/>

Basu, Chumki, Haym Hirsh, and William Cohen. "Recommendation as Classification: Using Social and Content-Based Information in Recommendation,"

- Proceedings of the Fifteenth National Conference on Artificial Intelligence.* Palo Alto, CA: AAAI Press, 1998, pp. 714–720.
- Bell, Robert M., and Yehuda Koren. “Scalable Collaborative Filtering with Jointly Derived Neighborhood Interpolation Weights,” *IEEE International Conference on Data Mining*. (2007). pp. 43–52.
- Bell, Robert M., Yehuda Koren, and Chris Volinsky. “The BellKor Solution to the Netflix Grand Prize,” Report from the Netflix Prize Winners, 2009. Available at http://www.netflixprize.com/assets/GrandPrize2009_BPC_BellKor.pdf.
- Goldberg, David, David Nichols, Brian M. Oki, and Douglas Terry. “Using Collaborative Filtering to Weave an Information Tapestry,” *Communications of the ACM* 35, no. 12 1992, December.
- Hinton, Geoffrey E. “Training Products of Experts by Minimizing Contrastive Divergence,” *Neural Computation* 14 (2002): 1771–1800.
- Konstan, Joseph A., and John Riedl. “Recommended for You,” *IEEE Spectrum*, 49, no. 10 (2012).
- Koren, Bell and Volinsky. “Matrix Factorization Techniques for Recommender Systems,” *IEEE Computer*, 2009.
- Salakhutdinov, Ruslan, and Andriy Mnih. “Bayesian Probabilistic Matrix Factorization Using Markov Chain Monte Carlo,” *Proceedings of the ICML* 25 (2008): 307.
- Salakhutdinov, Ruslan, and Andriy Mnih. “Probabilistic Matrix Factorization,” NIPS 2007. Available at <http://nips.cc/Conferences/2007/Program/event.php?ID=667>.
- Salakhutdinov, Ruslan, Andriy Mnih, and Geoffrey Hinton. “Restricted Boltzmann Machines for Collaborative Filtering,” *Proceedings of the ICML* (2007): 791–798. Available at <http://dl.acm.org/citation.cfm?id=1273596>.

CHAPTER 10

J!Archive. <http://www.J-Archive.com>.

Ken Jennings Detailed Statistics. <http://kenjenningsstatistics.blogspot.com>.

CHAPTER 17

¹ Thompson, Clive. “What is I.B.M.’s Watson?”. *The New York Times Magazine* (2010, June 16). Retrieved from www.nytimes.com/2010/06/20/magazine/20Computer-t.html?pagewanted=all

Index

A

AdaBoost, 106
adaptive regression, 82–83
additive methods, 106
agglomerative clustering, 138
Akaike information criterion (AIC), 69, 88
Akka, 46
algorithms, future development of, 238–241
aligned box criteria (ABC), 135
alternating least squares, 167
Amazon, 9
analytical tools
 about, 43
 Java and Java Virtual Machine (JVM) languages, 44–46
 Python, 49–50
 R language, 47–49
 SAS system, 50–52
 Weka (Waikato Environment for Knowledge Analysis), 43–44
analytics, 17–18
Apache Hadoop project, 7, 37–39, 38*f*, 40*f*, 45
Apple iPhone, 7
application program interface (API), 155, 227
Ashton, Kevin, 6, 236
assessing

predictive models, 67–70, 68*t*
quality of recommendation systems, 170–171
Auto-ID Center, 6, 236
average linkage, 134
averaged squared error (ASE), 201
averaging model process, 15
averaging process, 14–15

B

BackRub, 6
bagging, 124–125
banking case study, 197–204
Barksdale, Jim, 163
baseline model, 165–166
Bayes, Thomas, 111–112
Bayesian information criterion (BIC), 70, 88
Bayesian methods network
 classification
 about, 113–115
 inference in Bayesian networks, 122–123
 learning Bayesian networks, 120–122
 Naive Bayes Network, 116–117
 parameter learning, 117–120
 scoring for supervised learning, 123–124

- Bayesian network-augmented naive Bayes (BAN), 120
- Bayesian networks
- inference in, 122–123
 - learning, 120–122
- Bayesians, 114n14
- “Beowulf” cluster, 35, 35n1
- Bernoulli distribution, 85n7
- big data
- See also specific topics*
 - benefits of using, 12–17
 - complexities with, 19–21
 - defined, 3
 - fad nature of, 9–12
 - future of, 233–241
 - generalized linear model (GLM) applications and, 89–90
 - popularity of term, 10–11, 11f
 - regression analysis applications and, 83–84
 - timeline for, 5–8
 - value of using, 11–12
- Big Data Appliance, 40f
- “Big Data Dynamic Factor Models for Macroeconomic Measurement and Forecasting” (Diebolt), 3
- Big Data Research and Development Initiative, 8
- binary classification, 64–65, 113
- binary state, 225
- binning, 199
- Bluetooth, 6
- boosting, 124–125
- Boston Marathon, 160n2
- Box, George
- Empirical Model Building and Response Surfaces*, 57
- Box-Cox transformation
- method, 78
- breast cancer, 1–2
- Breiman, Leo
- Classification and Regression Trees*, 101
- Brin, Sergey, 6
- Broyden-Fletcher-Goldfarb-Shanno (BFGS) method, 96–97
- building models
- about, 223
 - methodology for, 58–61
 - response model, 142–143
- business rules, applying, 223
- Butte, Atul, 1
- C**
- C++, 49
- C language, 44–45, 49
- C4.5 splitting method, 101, 103
- Cafarella, Mike, 7, 37
- CAHPS, 207
- campaigns, operationalizing, 201–202
- cancer, 1–2
- capacity, of disk drives, 28
- CART, 101
- case studies
- financial services company, 197–204
 - health care provider, 205–214
 - high-tech product manufacturer, 229–232

- mobile application
 - recommendations, 225–228
- online brand management, 221–224
- technology manufacturer, 215–219
- Centers for Medicare and Medicaid Services (CMS), 206–207
- central processing unit (CPU), 29–30
- CHAID, 101, 105–106
- Challenger Space Shuttle, 20–21
- chips, 30
- Chi-square, 88, 105–106
- Churchill, Winston, 194
- city block (Manhattan) distance, 135
- classical supercomputers, 36
- Classification and Regression Trees* (Breiman), 101
- classification measures, 68*t*
- classification problems, 94–95
- Clojure, 45, 46
- cloud computing, 39
- cluster analysis, 132–133
- cluster computing, 35–36
- clustering techniques, 153, 231
- clusters/clustering
 - agglomerative cluster, 138
 - “Beowulf” cluster, 35, 35*n*1
 - divisive clustering, 138
 - hierarchical clustering, 132, 138
 - K*-means clustering, 132, 137–138
- number of clusters, 135–137
- profiling clusters, 138
- semisupervised clustering, 133
- supervised clustering, 133
- text clustering, 177
- unsupervised clustering, 132
- collaborative filtering, 164
- complete linkage, 134
- computing environment
 - about, 24–25
 - analytical tools, 43–52
 - distributed systems, 35–41
 - hardware, 27–34
- Conan Doyle, Sir Arthur, 55
- considerations, for platforms, 39–41
- content categorization, 177–178, 222
- content-based filtering, 163–164
- contrastive divergence, 169–170
- Cook’s D plot, 81, 88
- Corrected AIC (AICC), 88
- Cortes, Corinna
 - Support-Vector Networks*, 109
- cosine similarity measure, 134
- cost, of hardware, 36
- Cox models, 83
- Credit Card Accountability Responsibility and Disclosure Act (2009), 199–200
- cubic clustering criteria (CCC), 135
- customers, ranking potential, 201
- Cutting, Doug, 7, 37

D

DAG network structure, 120
 data
See also big data
 about, 54
 amount created, 6, 8
 common predictive modeling techniques, 71–126
 democratization of, 8
 incremental response modeling, 141–148
 personal, 4
 predictive modeling, 55–70
 preparing for building models, 58–59
 preprocessing, 230
 recommendation systems, 163–174
 segmentation, 127–139
 text analytics, 175–191
 time series data mining, 149–161
 validation, 97–98
 data mining
See also time series data mining
See also specific topics
 future of, 233–241
 multidisciplinary nature of, 55–56, 56f
 shifts in, 24–25
Data Mining with Decision Trees
 (Rokach and Maimon), 101
 Data Partition node, 62
 data scientist, 8
 data sets, privacy with, 234–236
 DATA Step, 50

database computing, 36–37
 decision matrix, 68t
 decision trees, 101–107, 102f, 103f, 231
 Deep Learning, 100
 Defense Advanced Research Projects Agency (DARPA), 5–6
 democratization of data, 8
 detecting patterns, 151–153
 deviance, 88, 88n9
 DFBETA statistic, 89, 89f
 Diebolt, Francis X.
 “Big Data Dynamic Factor Models for Macroeconomic Measurement and Forecasting” (Diebolt), 3
 differentiable, 85n6
 dimensionality, reducing, 150–151
 discrete Fourier transformation (DFT), 151
 discrete wavelet transformation (DWT), 151
 disk array, 28n1
 distance measures (metrics), 133–134
 distributed systems
 about, 35–36
 considerations, 39–41
 database computing, 36–37
 file system computing, 37–39
 distributed.net, 35
 divisive clustering, 138
 ductal carcinoma, 1–2
 dynamic time warping methods, 158

E

- economy of scale, 199
 Edison, Thomas, 60
Empirical Model Building and Response Surfaces (Box), 57
 ensemble methods, 124–126
 enterprise data warehouses (EDWs), 37, 58, 198–199, 217
 entropy, 105
Estimation of Dependencies Based on Empirical Data (Vapnik), 107
 Euclidean distance, 133
 exploratory data analysis, performing, 59–60
 external evaluation criterion, 134–135
 extremity-based model, 15

F

- Facebook, 7
 “failing fast” paradigm, 18
 feature films, as example of segmentation, 132
 feedback loop, 223
 file crawling, 176
 file system computing, 37–39
 financial services company case study, 197–204
 Fisher, Sir Ronald, 98
 forecasting, new product, 152–153
 Forrester Wave and Gartner Magic Quadrant, 50
 FORTRAN language, 44–45, 46, 49
 fraud, 16, 152

frequentists, 114n14

functional dimension analysis (FDA), 219

future

of big data, data mining and machine learning, 233–241
 development of algorithms, 238–241
 of software development, 237–238

G

- gain, 69
 Galton, Francis, 75
 gap, 135
 Gauss, Carl, 75
 Gaussian distribution, 119
 Gaussian kernel, 112
 generalized additive models (GAM), 83
 generalized linear models (GLMs)
 about, 84–86
 applications for big data, 89–90
 probit, 86–89
 Gentleman-Givens computational method, 83
 global minimum, assurance of, 98
 Global Positioning System (GPS), 5–6
 Goodnight, Jim, 50
 Google, 6, 10
 Gosset, William, 80n1
 gradient boosting, 106
 gradient descent method, 96
 graphical processing unit (GPU), 30–31

H

Hadoop, 7, 37–39, 38*f*, 40*f*, 45
 Hadoop Streaming Utility, 49
 hardware
 central processing unit (CPU), 29–30
 cost of, 36
 graphical processing unit (GPU), 30–31
 memory, 31–33
 network, 33–34
 storage (disk), 27–29
 health care provider case study, 205–214
 HEDIS, 207–208
 Hertzano, Ephraim, 129–130
 Hessian, 96–97, 96n11
 Hession Free Learning, 101
 Hickey, Rick (programmer), 46
 hierarchical clustering, 132, 138
 high-performance marketing solution, 202–203
 high-tech product manufacturer case study, 229–232
 Hinton, Geoffrey E., 100
 “Learning Representation by Back-Propagating Errors”, 92
 Hornik, Kurt
 “Multilayer Feedforward networks are Universal Approximators”, 92
 HOS, 208
 Huber M estimation, 83
 hyperplane, 107n13
 Hypertext Transfer Protocol (HTTP), 5
 hypothesis, 234

I

IBM, 7, 239
 identity output activation function, 94
 incremental response modeling
 about, 141–142
 building model, 142–143
 measuring, 143–148
 index, 177
 inference, in Bayesian networks, 122–123
 information retrieval, 176–177, 183–184, 211, 245–246
 in-memory databases (IMDBs), 37, 40*f*
 input scaling, 98
 Institute of Electrical and Electronic Engineers (IEEE), 6
 internal evaluation criterion, 134
 Internet, birth of, 5
 Internet of Things, 6, 236–237
 interpretability, as drawback of SVM, 113
 interval prediction, 66–67
 iPhone (Apple), 7
 IPv4 protocol, 7
 IRE, 208
 iterative model building, 60–61

J

Java and Java Virtual Machine (JVM) languages, 5, 44–46
 Jennings, Ken, 182
Jeopardy (TV show), 7, 180–191, 239–241

- K**
- Kaggle, 161n3
 - kernels, 112
 - K-means clustering, 132, 137–138
 - Kolmogorov-Smirnov (KS) statistic, 70
- L**
- Lagrange multipliers, 111
 - language detection, 222
 - latent factor models, 164
 - “Learning Representation by Back-Propagating Errors” (Rumelhart, Hinton and Williams), 92
 - Legendre, Adrien-Marie, 75
 - lift, 69
 - likelihood, 114
 - Limited Memory BFGS (LBFGS) method, 97–98
 - linear regression, 15
 - LinkedIn, 6
 - Linux, 237, 238
 - Lisp programming language, 46
 - lobular carcinoma, 2
 - locally estimated scatterplot smoothing (LOESS), 83
 - logistic regression model, 85
 - low-rank matrix factorization, 164, 166
- M**
- machine learning, future of, 233–241
 - See also specific topics*
 - Maimon, Oded
 - Data Mining with Decision Trees*, 103
- MapReduce paradigm, 38
 - marketing campaign process, traditional, 198–202
 - Markov blanket bayesian network (MB), 120
 - Martens, James, 101
 - massively parallel processing (MPP) databases, 37, 40f, 202–203, 211–213, 217
 - Matplotlib library, 49
 - matrix, rank of, 164n1
 - maximum likelihood estimation, 117
 - Mayer, Marissa, 24
 - McAuliffe, Sharon, 20
 - McCulloch, Warren, 90
 - mean, 51
 - Medicare Advantage, 206
 - memory, 31–33
 - methodology, for building models, 58–61
 - Michael J. Fox Foundation, 161
 - microdata, 235n1
 - Microsoft, 237
 - Minsky, Marvin
 - Perceptrons*, 91–92
 - misclassification rate, 105
 - MIT, 236
 - mobile application recommendations case study, 225–228
 - model lift, 204
 - models
 - creating, 223
 - methodology for building, 58–61
 - scoring, 201, 213

- monitoring process, 223–224
 monotonic, 85n6
 Moore, Gordon, 29–30
 Moore’s law, 29–30
 Multicore, 48
 multicore chips, 30
 “Multilayer Feedforward networks are Universal Approximators” (Hornik, Stinchcombe and White), 92
 multilayer perceptrons (MLP), 93–94
 multilevel classification, 66
- N**
 Naive Bayes Network, 116–117
 Narayanan, Arvind, 235
 National Science Foundation (NSF), 7
 National Security Agency, 10
 neighborhood-based methods, 164
 Nelder, John, 84, 87
 Nest thermostat, 237
 Netflix, 235
 network, 33–34
 NetWorkSpaces (NWS), 48, 48n1
 neural networks
 about, 15, 90–98
 basic example of, 98–101
 diagrammed, 90f, 93f
 key considerations, 97–98
 Newton’s method, 96
 next-best offer, 164
 Nike+ FuelBand, 154–161, 245–246
 Nike+iPod sensor, 154
 “No Harm Intended” (Pollack), 92
 Nolvadex, 2–3
 nominal classification, 66
 nonlinear relationships, 19–21
 nonparametric techniques, 15
 NoSQL database, 6, 40f
 Numpy library, 49
 Nutch, 37, 38n2
- O**
 object-oriented (OO)
 programming paradigms, 45–46
 one-class support vector machines (SVMs), 144–146
 online brand management case study, 221–224
 open source projects, 238
 operationalizing campaigns, 201–202
 optimization, 16–17
 Orange library, 49
 ordinary least squares, 76–82
 orthogonal regression, 83
 overfitting, 97–98
- P**
 Page, Larry, 6
 Papert, Seymour
 Perceptrons, 91–92
 Parallel, 48–49
 Parallel Virtual Machine (PVM), 48, 48n1
 parameter learning, 117–120

- parent child Bayesian network (PC), 120
- parsimony, 218
- partial least squares regression, 83
- path analysis, 218
- Patient Protection and Affordable Care Act (2010), 206
- Pattern library, 49
- patterns, detecting, 152–153
- Pearl, Judea
Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, 121n16
- Pearson’s chi-square statistic, 88
- Perceptron, 90–94, 91f
Perceptrons (Minsky and Papert), 91–92
- performing exploratory data analysis, 59–60
- personal data, 4
- piecewise aggregate approximation (PAA) method, 151
- Pitts, Walter, 90
- platforms, for file system computing, 37–39
- Pollack, Jordan B.
“*No Harm Intended*”, 92
- Polynomial kernel, 112
- POSIX-compliant, 48–49
- posterior, 114
- predictive modeling
about, 55–58
assessment of predictive models, 67–70
binary classification, 64–65
creating models, 200–201
interval prediction, 66–67
methodology for building models, 58–61
multilevel classification, 66
sEMMA, 61–63
- predictive modeling techniques
about, 71–72
Bayesian methods network classification, 113–124
decision and regression trees, 101–107
ensemble methods, 124–126
generalized linear models (GLMs), 84–90
neural networks, 90–101
regression analysis, 75–84
RFM (recency, frequency, and monetary modeling), 72–75, 73t
support vector machines (SVMs), 107–113
- prior, 114
- privacy, 234–236
- PRIZM, 132–133
- Probabilistic Programming for Advanced Machine Learning (PPAML), 239
Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference (Pearl), 121n16
- probit, 84n4, 86–89
- processes
averaging, 14–15
monitoring, 223–224
tail-based modeling, 15
- Proctor & Gamble, 236

- profiling clusters, 140
- Progressive Insurance's Snapshot™ device, 162
- proportional hazards regression, 83
- public radio, as example to demonstrate incremental response, 142–143
- p*-values, 88, 88n10
- Python, 49–50
- Q**
- quantile regression, 83
- Quasi-Newton methods, 96, 100–101
- R**
- R language, 47–49
- radial basis function networks (RBF), 93
- radio-frequency identification (RFID) tags, 234
- random access memory (RAM), 31–33, 44
- Random Forest, 106, 125–126
- rank sign test, 15
- ranking potential customers, 201
- receiver operating characteristic (ROC), 68
- recency, frequency, and monetary modeling (RFM), 72–75, 73*t*
- recommendation systems about, 163–164 assessing quality of, 170–171 how they work, 165–170
- SAS Library, 171–173 where they are used, 164–165
- Red Hat, 238
- reduced-dimension models, 164
- REG procedure, 51
- regression adaptive, 82–83 linear, 15 orthogonal, 83 problems with, 94 quantile, 83 robust, 83
- regression analysis about, 75–76 applications for big data, 83–84 assumptions of regression models, 82 ordinary least squares, 76–82 power of, 81–82 techniques, 82–83
- regression models, assumptions of, 82
- regression trees, 101–107
- relational database management system (RDBMS), 36
- relationship management, 17
- reproducible research, 234
- response model, building, 142–143
- Restricted Boltzmann Machines, 100–101, 164, 167–169
- Rhadoop, 48, 49
- RHIPE, 48, 49
- robust regression, 83

- Rokach, Lior
Data Mining with Decision Trees, 103
- root cause analysis, 231
- root mean square error (RMSE), 170–171
- Rosenblatt, Frank, 90
- R-Square (R^2), 81
- Rumelhart, David E.
“Learning Representation by Back-Propagating Errors”, 92
- Rummikub, 127–132, 127n1, 128f
- S**
- S language, 47
- Salakhutdinov, R.R., 100
- Sall, John, 50
- sampling, 13–14
- SAS Enterprise Miner software, 71, 127–128
- SAS Global Forum, 158
- SAS Library, 171–173
- SAS System, as analytical tool, 50–52
- SAS Text Miner software, 175
- SAS/ETS®, 158
- SAS/STAT®, 50
- Scala, 45–46
- scaled deviance, 88, 88n9
- scikit-learn toolkit, 49–50
- Scipy library, 49
- scoring
models, 201, 213
for supervised learning, 123–124
- search, 177
- seasonal analysis, 155–157
- segmentation
about, 127–132
cluster analysis, 132–133
distance measures (metrics), 133–134
evaluating clustering, 134–135
hierarchical clustering, 138
K-means algorithm, 137–138
number of clusters, 135–137
profiling clusters, 138–139
RFM, 72–73, 73t
- self-organizing maps (SOM), 93
- semisupervised clustering, 133
- sEMMA, 61–63
- sentiment analysis, 222
- SETI@Home, 35, 36
- Sigmoid kernel, 112
- similarity analysis, 152, 153, 157–161
- SIMILARITY procedure, 158
- single linkage, 134
- singular value decomposition (SVD), 151, 211–212
- size, of disk drives, 28–29
- slack variables, 111
- Snow, 48
- software
future developments in, 237–238
shifts in, 24–25
three-dimensional (3D)
computer-aided design (CAD), 30–31
- solid state devices (SSDs), 28
- spanning tree, 121n15
- Spark, 46

- speed
of CPU, 29–30
of memory, 33
network, 33–34
- star ratings, 206–207
- starting values, 97
- Stinchcombe, Maxwell
“Multilayer Feedforward networks are Universal Approximators”, 92
- stochastic gradient descent, 166–167
- storage (disk), 27–29
- storage area network (SAN), 33n3
- Strozzi, Carlo, 6
- Structured Query Language (SQL), 199
- success stories (case studies)
financial services company, 197–204
- health care provider, 205–214
- high-tech product
manufacturer, 229–232
- mobile application
recommendations, 225–228
- online brand management, 221–224
- qualities of, 194–195
- technology manufacturer, 215–219
- sum of squares error (SSE), 134
- supercomputers, 36
- supervised clustering, 133
- supervised learning, scoring for, 123–124
- support vector machines (SVMs), 107–113, 144–146
- Support-Vector Networks* (Cortes and Vapnik), 111
- symbolic aggregate approximation (SAX), 219
- T**
- tail-based modeling process, 15
- tamoxifen citrate, 2–3
- technology manufacturer case study, 215–219
- teraflop, 239n2
- text analytics
about, 175–176
content categorization, 177–178
example of, 180–191
information retrieval, 176–177, 183–184
text mining, 178–180
- text clustering, 180
- text extraction, 176–177
- text mining, 178–180
- text topic identification, 177
- 32-bit operating systems, 32n2
- three-dimensional (3D)
computer-aided design (CAD) software, 30–31
- TIBCO, 47
- time series data mining
about, 149–150
detecting patterns, 151–153
Nike+ FuelBand, 154–161
reducing dimensionality, 150–151

- TIOBE, 47, 47n1
 tools, analytical
 about, 43
 Java and Java Virtual Machine (JVM) languages, 44–46
 Python, 49–50
 R language, 47–49
 SAS system, 50–52
 Weka (Waikato Environment for Knowledge Analysis), 43–44
 traditional marketing campaign process, 198–202
 tree-augmented naive Bayes (TAN), 120–121
 tree-based methods, 101–107
 trend analysis, 157
 Truncated Newton Hessian Free Learning, 101
 Twitter, 221–224
- U**
 UNIVARIATE procedure, 51
 UnQL, 7
 unsupervised clustering, 132
 unsupervised segmentation, 135
- V**
 validation data, 97–98
 Vapnik, Vladimir
- Estimation of Dependencies Based on Empirical Data*, 107
Support-Vector Networks, 111
 variance, 105
 volatility, of memory, 32
- W**
 Ward's method, 138
 Watson, Thomas J., 239
 Watson computer, 7, 239–240
 web crawling, 176
 Wedderburn, Robert, 84, 87
 Weka (Waikato Environment for Knowledge Analysis), 43–44
 White, Halbert
 “Multilayer Feedforward networks are Universal Approximators”, 92
 Wikipedia, 6, 7
 Williams, Ronald J.
 “Learning Representation by Back-Propagating Errors”, 92
 Winsor, Charles, 80n2
 winsorize, 80, 80n2
 work flow productivity, 17–19
 World Wide Web, birth of, 5
- Z**
 Zuckerberg, Mark, 7