

Phuong Vothihong, Martin Czygan,
Ivan Idris, Magnus Vilhelm Persson,
Luiz Felipe Martins

Python: End-to-end Data Analysis

Learning Path

Leveraging the power of Python to clean, scrape, analyze, and visualize your data



Packt

Python: End-to-end Data Analysis

Leverage the power of Python to clean, scrape,
analyze, and visualize your data

A course in three modules



BIRMINGHAM - MUMBAI

Python: End-to-end Data Analysis

Copyright © 2016 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: May 2017

Production reference: 1050517

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78839-469-7

www.packtpub.com

Credits

Authors

Phuong Vo.T.H
Martin Czygan
Ivan Idris
Magnus VilhelmPersson
Luiz Felipe Martins

Reviewers

Dong Chao
Hai Minh Nguyen
Kenneth Emeka Odoh
Bill Chambers
Alexey Grigorev
Dr. VahidMirjalili
Michele Usuelli
Hang (Harvey) Yu
Laurie Lugrin
Chris Morgan
Michele Pratusevich

Content Development Editor

Aishwarya Pandere

Graphics

Jason Monteiro

Production Coordinator

Deepika Naik

Preface

The use of Python for data analysis and visualization has only increased in popularity in the last few years.

The aim of this book is to develop skills to effectively approach almost any data analysis problem, and extract all of the available information. This is done by introducing a range of varying techniques and methods such as uni- and multi-variate linear regression, cluster finding, Bayesian analysis, machine learning, and time series analysis. Exploratory data analysis is a key aspect to get a sense of what can be done and to maximize the insights that are gained from the data. Additionally, emphasis is put on presentation-ready figures that are clear and easy to interpret.

What this learning path covers

Module 1, Getting Started with Python Data Analysis, shows how to work with time-oriented data in Pandas. How do you clean, inspect, reshape, merge, or group data – these are the concerns in this chapter. The library of choice in the course will be Pandas again.

Module 2, Python Data Analysis Cookbook, demonstrates how to visualize data and mentions frequently encountered pitfalls. Also, discusses statistical probability distributions and correlation between two variables.

Module 3, Mastering Python Data Analysis, introduces linear, multiple, and logistic regression with in-depth examples of using SciPy and stats models packages to test various hypotheses of relationships between variables.

What you need for this learning path

Module 1:

There are not too many requirements to get started. You will need a Python programming environment installed on your system. Under Linux and Mac OS X, Python is usually installed by default. Installation on Windows is supported by an excellent installer provided and maintained by the community. This book uses a recent Python 2, but many examples will work with Python 3 as well.

The versions of the libraries used in this book are the following: NumPy 1.9.2, Pandas 0.16.2, matplotlib 1.4.3, tables 3.2.2, pymongo 3.0.3, redis 2.10.3, and scikit-learn 0.16.1. As these packages are all hosted on PyPI, the Python package index, they can be easily installed with pip. To install NumPy, you would write:

```
$ pip install numpy
```

If you are not using them already, we suggest you take a look at virtual environments for managing isolating Python environment on your computer. For Python 2, there are two packages of interest there: `virtualenv` and `virtualenvwrapper`. Since Python 3.3, there is a tool in the standard library called `pyvenv` (<https://docs.python.org/3/library/venv.html>), which serves the same purpose.

Most libraries will have an attribute for the version, so if you already have a library installed, you can quickly check its version:

```
>>>importredis  
>>>redis.__version__'2.10.3'
```

This works well for most libraries. A few, such as pymongo, use a different attribute (pymongo uses just `version`, without the underscores). While all the examples can be run interactively in a Python shell, we recommend using IPython. IPython started as a more versatile Python shell, but has since evolved into a powerful tool for exploration and sharing. We used IPython 4.0.0 with Python 2.7.10. IPython is a great way to work interactively with Python, be it in the terminal or in the browser.

Module 2:

First, you need a Python 3 distribution. I recommend the full Anaconda distribution as it comes with the majority of the software we need. I tested the code with Python 3.4 and the following packages:

- joblib 0.8.4
- IPython 3.2.1

- NetworkX 1.9.1
- NLTK 3.0.2
- Numexpr 2.3.1
- pandas 0.16.2
- SciPy 0.16.0
- seaborn 0.6.0
- sqlalchemy 0.9.9
- statsmodels 0.6.1
- matplotlib 1.5.0
- NumPy 1.10.1
- scikit-learn 0.17
- dautil0.0.1a29

For some recipes, you need to install extra software, but this is explained whenever the software is required.

Module 3:

All you need to follow through the examples in this book is a computer running any recent version of Python. While the examples use Python 3, they can easily be adapted to work with Python 2, with only minor changes. The packages used in the examples are NumPy, SciPy, matplotlib, Pandas, stats models, PyMC, Scikit-learn. Optionally, the packages basemap and cartopy are used to plot coordinate points on maps. The easiest way to obtain and maintain a Python environment that meets all the requirements of this book is to download a prepackaged Python distribution. In this book, we have checked all the code against Continuum Analytics' Anaconda Python distribution and Ubuntu Xenial Xerus (16.04) running Python 3.

To download the example data and code, an Internet connection is needed.

Who this learning path is for

This learning path is for developers, analysts, and data scientists who want to learn data analysis from scratch. This course will provide you with a solid foundation from which to analyze data with varying complexity. A working knowledge of Python (and a strong interest in playing with your data) is recommended.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at <http://www.packtpub.com>. If you purchased this course elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the course in the Search box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on Code Download.

You can also download the code files by clicking on the Code Files button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the Search box. Please note that you need to be logged into your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac

7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at <https://github.com/PacktPublishing/Python-End-to-end-Data-Analysis>. We also have other code bundles from our rich catalog of books, videos, and courses available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the course in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Module 1: Getting Started with Python Data Analysis

Chapters 1: Introducing Data Analysis and Libraries	3
Data analysis and processing	4
An overview of the libraries in data analysis	7
Python libraries in data analysis	9
NumPy	10
Pandas	10
Matplotlib	11
PyMongo	11
The scikit-learn library	11
Summary	11
Chapters 2: NumPy Arrays and Vectorized Computation	13
NumPy arrays	14
Data types	14
Array creation	16
Indexing and slicing	18
Fancy indexing	19
Numerical operations on arrays	20
Array functions	21
Data processing using arrays	23
Loading and saving data	24
Saving an array	24
Loading an array	25
Linear algebra with NumPy	26
NumPy random numbers	27
Summary	30

Chapters 3: Data Analysis with Pandas	33
An overview of the Pandas package	33
The Pandas data structure	34
Series	34
The DataFrame	36
The essential basic functionality	40
Reindexing and altering labels	40
Head and tail	41
Binary operations	42
Functional statistics	43
Function application	45
Sorting	46
Indexing and selecting data	48
Computational tools	49
Working with missing data	51
Advanced uses of Pandas for data analysis	54
Hierarchical indexing	54
The Panel data	56
Summary	58
Chapters 4: Data Visualization	61
The matplotlib API primer	62
Line properties	65
Figures and subplots	67
Exploring plot types	70
Scatter plots	70
Bar plots	71
Contour plots	72
Histogram plots	74
Legends and annotations	75
Plotting functions with Pandas	78
Additional Python data visualization tools	80
Bokeh	81
MayaVi	81
Summary	83
Chapters 5: Time Series	85
Time series primer	85
Working with date and time objects	86
Resampling time series	94

Downsampling time series data	94
Upsampling time series data	97
Time zone handling	99
Timedeltas	100
Time series plotting	101
Summary	105
Chapters 6: Interacting with Databases	107
Interacting with data in text format	107
Reading data from text format	107
Writing data to text format	112
Interacting with data in binary format	113
HDF5	114
Interacting with data in MongoDB	115
Interacting with data in Redis	120
The simple value	120
List	121
Set	122
Ordered set	123
Summary	124
Chapters 7: Data Analysis Application Examples	127
Data munging	128
Cleaning data	130
Filtering	133
Merging data	136
Reshaping data	139
Data aggregation	141
Grouping data	144
Summary	146
Chapters 8: Machine Learning Models with scikit-learn	147
An overview of machine learning models	147
The scikit-learn modules for different models	148
Data representation in scikit-learn	150
Supervised learning – classification and regression	152
Unsupervised learning – clustering and dimensionality reduction	158
Measuring prediction performance	162
Summary	164

Module 2: Python Data Analysis Cookbook

Chapter 1: Laying the Foundation for Reproducible Data Analysis	167
Introduction	168
Setting up Anaconda	168
Getting ready	169
How to do it...	169
There's more...	170
See also	170
Installing the Data Science Toolbox	170
Getting ready	171
How to do it...	171
How it works...	172
See also	172
Creating a virtual environment with virtualenv and virtualenvwrapper	172
Getting ready	173
How to do it...	173
See also	174
Sandboxing Python applications with Docker images	174
Getting ready	174
How to do it...	174
How it works...	176
See also	176
Keeping track of package versions and history in IPython Notebook	176
Getting ready	177
How to do it...	177
How it works...	179
See also	179
Configuring IPython	179
Getting ready	180
How to do it...	180
See also	181
Learning to log for robust error checking	182
Getting ready	182
How to do it...	182
How it works...	185
See also	185

Unit testing your code	185
Getting ready	185
How to do it...	186
How it works...	187
See also	187
Configuring pandas	188
Getting ready	188
How to do it...	188
Configuring matplotlib	190
Getting ready	191
How to do it...	191
How it works...	194
See also	194
Seeding random number generators and NumPy print options	194
Getting ready	194
How to do it...	194
See also	196
Standardizing reports, code style, and data access	196
Getting ready	197
How to do it...	197
See also	199
Chapter 2: Creating Attractive Data Visualizations	201
Introduction	202
Graphing Anscombe's quartet	202
How to do it...	202
See also	205
Choosing seaborn color palettes	205
How to do it...	205
See also	208
Choosing matplotlib color maps	208
How to do it...	208
See also	209
Interacting with IPython Notebook widgets	209
How to do it...	209
See also	213
Viewing a matrix of scatterplots	213
How to do it...	213
Visualizing with d3.js via mpld3	215
Getting ready	215
How to do it...	216

Creating heatmaps	217
Getting ready	217
How to do it...	217
See also	219
Combining box plots and kernel density plots with violin plots	220
How to do it...	220
See also	221
Visualizing network graphs with hive plots	221
Getting ready	222
How to do it...	222
Displaying geographical maps	224
Getting ready	224
How to do it...	224
Using ggplot2-like plots	226
Getting ready	227
How to do it...	227
Highlighting data points with influence plots	228
How to do it...	229
See also	231
Chapter 3: Statistical Data Analysis and Probability	233
Introduction	234
Fitting data to the exponential distribution	234
How to do it...	234
How it works...	236
See also	236
Fitting aggregated data to the gamma distribution	237
How to do it...	237
See also	238
Fitting aggregated counts to the Poisson distribution	238
How to do it...	239
See also	241
Determining bias	241
How to do it...	242
See also	244
Estimating kernel density	244
How to do it...	244
See also	246
Determining confidence intervals for mean, variance, and standard deviation	247
How to do it...	247

See also	249
Sampling with probability weights	249
How to do it...	250
See also	252
Exploring extreme values	253
How to do it...	253
See also	256
Correlating variables with Pearson's correlation	257
How to do it...	257
See also	260
Correlating variables with the Spearman rank correlation	260
How to do it...	260
See also	263
Correlating a binary and a continuous variable with the point biserial correlation	263
How to do it...	263
See also	265
Evaluating relations between variables with ANOVA	265
How to do it...	266
See also	267
Chapter 4: Dealing with Data and Numerical Issues	269
Introduction	269
Clipping and filtering outliers	270
How to do it...	270
See also	272
Winsorizing data	273
How to do it...	273
See also	274
Measuring central tendency of noisy data	275
How to do it...	275
See also	277
Normalizing with the Box-Cox transformation	278
How to do it...	278
How it works	280
See also	280
Transforming data with the power ladder	280
How to do it...	281
Transforming data with logarithms	282
How to do it...	283

Rebinning data	284
How to do it...	285
Applying logit() to transform proportions	286
How to do it...	287
Fitting a robust linear model	288
How to do it...	289
See also	291
Taking variance into account with weighted least squares	291
How to do it...	291
See also	294
Using arbitrary precision for optimization	294
Getting ready	294
How to do it...	294
See also	296
Using arbitrary precision for linear algebra	297
Getting ready	297
How to do it...	297
See also	299
Chapter 5: Web Mining, Databases, and Big Data	301
Introduction	302
Simulating web browsing	302
Getting ready	303
How to do it...	303
See also	305
Scraping the Web	305
Getting ready	306
How to do it...	306
Dealing with non-ASCII text and HTML entities	308
Getting ready	308
How to do it...	308
See also	310
Implementing association tables	310
Getting ready	310
How to do it...	310
Setting up database migration scripts	313
Getting ready	314
How to do it...	314
See also	314

Adding a table column to an existing table	314
Getting ready	314
How to do it...	315
Adding indices after table creation	316
Getting ready	316
How to do it...	316
How it works...	317
See also	317
Setting up a test web server	317
Getting ready	318
How to do it...	318
Implementing a star schema with fact and dimension tables	319
How to do it...	320
See also	324
Using HDFS	325
Getting ready	325
How to do it...	325
See also	326
Setting up Spark	326
Getting ready	327
How to do it...	327
See also	327
Clustering data with Spark	327
Getting ready	328
How to do it...	328
How it works...	331
There's more...	331
See also	331
Chapter 6: Signal Processing and Timeseries	333
Introduction	333
Spectral analysis with periodograms	334
How to do it...	334
See also	336
Estimating power spectral density with the Welch method	336
How to do it...	336
See also	338
Analyzing peaks	338
How to do it...	338
See also	340

Measuring phase synchronization	340
How to do it...	341
See also	342
Exponential smoothing	343
How to do it...	343
See also	345
Evaluating smoothing	346
How to do it...	346
See also	348
Using the Lomb-Scargle periodogram	349
How to do it...	349
See also	351
Analyzing the frequency spectrum of audio	351
How to do it...	352
See also	354
Analyzing signals with the discrete cosine transform	354
How to do it...	355
See also	356
Block bootstrapping time series data	357
How to do it...	357
See also	359
Moving block bootstrapping time series data	359
How to do it...	360
See also	362
Applying the discrete wavelet transform	363
Getting started	364
How to do it...	364
See also	366
Chapter 7: Selecting Stocks with Financial Data Analysis	367
Introduction	368
Computing simple and log returns	368
How to do it...	369
See also	369
Ranking stocks with the Sharpe ratio and liquidity	370
How to do it...	370
See also	372
Ranking stocks with the Calmar and Sortino ratios	372
How to do it...	372
See also	374

Analyzing returns statistics	374
How to do it...	375
Correlating individual stocks with the broader market	377
How to do it...	377
Exploring risk and return	380
How to do it...	380
See also	381
Examining the market with the non-parametric runs test	382
How to do it...	382
See also	384
Testing for random walks	385
How to do it...	385
See also	386
Determining market efficiency with autoregressive models	387
How to do it...	387
See also	389
Creating tables for a stock prices database	389
How to do it...	390
Populating the stock prices database	391
How to do it...	391
Optimizing an equal weights two-asset portfolio	396
How to do it...	397
See also	399
Chapter 8: Text Mining and Social Network Analysis	401
Introduction	401
Creating a categorized corpus	402
Getting ready	402
How to do it...	403
See also	405
Tokenizing news articles in sentences and words	405
Getting ready	405
How to do it...	405
See also	406
Stemming, lemmatizing, filtering, and TF-IDF scores	406
Getting ready	408
How to do it...	408
How it works	409

See also	410
Recognizing named entities	410
Getting ready	410
How to do it...	411
How it works	412
See also	412
Extracting topics with non-negative matrix factorization	412
How to do it...	413
How it works	414
See also	414
Implementing a basic terms database	414
How to do it...	415
How it works	418
See also	418
Computing social network density	418
Getting ready	419
How to do it...	419
See also	420
Calculating social network closeness centrality	420
Getting ready	420
How to do it...	420
See also	421
Determining the betweenness centrality	421
Getting ready	421
How to do it...	422
See also	422
Estimating the average clustering coefficient	423
Getting ready	423
How to do it...	423
See also	424
Calculating the assortativity coefficient of a graph	424
Getting ready	424
How to do it...	425
See also	425
Getting the clique number of a graph	425
Getting ready	426
How to do it...	426
See also	426

Creating a document graph with cosine similarity	427
How to do it...	428
See also	430
Chapter 9: Ensemble Learning and Dimensionality Reduction	431
Introduction	432
Recursively eliminating features	432
How to do it...	433
How it works	434
See also	434
Applying principal component analysis for dimension reduction	435
How to do it...	435
See also	436
Applying linear discriminant analysis for dimension reduction	437
How to do it...	437
See also	438
Stacking and majority voting for multiple models	438
How to do it...	439
See also	441
Learning with random forests	442
How to do it...	442
There's more...	444
See also	445
Fitting noisy data with the RANSAC algorithm	445
How to do it...	446
See also	448
Bagging to improve results	449
How to do it...	449
See also	451
Boosting for better learning	452
How to do it...	452
See also	454
Nesting cross-validation	455
How to do it...	455
See also	458
Reusing models with joblib	458
How to do it...	458
See also	459
Hierarchically clustering data	460
How to do it...	460
See also	461

Taking a Theano tour	462
Getting ready	462
How to do it...	462
See also	464
Chapter 10: Evaluating Classifiers, Regressors, and Clusters	465
Introduction	466
Getting classification straight with the confusion matrix	466
How to do it...	467
How it works	468
See also	469
Computing precision, recall, and F1-score	469
How to do it...	470
See also	472
Examining a receiver operating characteristic and the area under a curve	472
How to do it...	473
See also	474
Visualizing the goodness of fit	475
How to do it...	475
See also	476
Computing MSE and median absolute error	476
How to do it...	477
See also	479
Evaluating clusters with the mean silhouette coefficient	479
How to do it...	479
See also	481
Comparing results with a dummy classifier	482
How to do it...	482
See also	484
Determining MAPE and MPE	485
How to do it...	485
See also	487
Comparing with a dummy regressor	487
How to do it...	487
See also	489
Calculating the mean absolute error and the residual sum of squares	490
How to do it...	490
See also	492

Examining the kappa of classification	492
How to do it...	493
How it works	495
See also	495
Taking a look at the Matthews correlation coefficient	495
How to do it...	495
See also	497
Chapter 11: Analyzing Images	499
Introduction	499
Setting up OpenCV	500
Getting ready	500
How to do it...	501
How it works	502
There's more	503
Applying Scale-Invariant Feature Transform (SIFT)	503
Getting ready	503
How to do it...	503
See also	505
Detecting features with SURF	505
Getting ready	506
How to do it...	506
See also	507
Quantizing colors	507
Getting ready	508
How to do it...	508
See also	509
Denoising images	509
Getting ready	510
How to do it...	510
See also	511
Extracting patches from an image	511
Getting ready	512
How to do it...	512
See also	514
Detecting faces with Haar cascades	514
Getting ready	515
How to do it...	515
See also	517
Searching for bright stars	517
Getting ready	518

How to do it...	518
See also	520
Extracting metadata from images	521
Getting ready	521
How to do it...	521
See also	523
Extracting texture features from images	523
Getting ready	524
How to do it...	524
See also	526
Applying hierarchical clustering on images	526
How to do it...	526
See also	527
Segmenting images with spectral clustering	527
How to do it...	528
See also	529
Chapter 12: Parallelism and Performance	531
Introduction	531
Just-in-time compiling with Numba	533
Getting ready	533
How to do it...	533
How it works	534
See also	535
Speeding up numerical expressions with Numexpr	535
How to do it...	535
How it works	536
See also	536
Running multiple threads with the threading module	536
How to do it...	536
See also	539
Launching multiple tasks with the concurrent.futures module	540
How to do it...	540
See also	542
Accessing resources asynchronously with the asyncio module	543
How to do it...	543
See also	546
Distributed processing with execnet	546
Getting ready	547
How to do it...	547
See also	549

Profiling memory usage	550
Getting ready	550
How to do it...	550
See also	551
Calculating the mean, variance, skewness, and kurtosis on the fly	551
Getting ready	552
How to do it...	552
See also	556
Caching with a least recently used cache	556
Getting ready	556
How to do it...	556
See also	559
Caching HTTP requests	559
Getting ready	559
How to do it...	560
See also	560
Streaming counting with the Count-min sketch	561
How to do it...	562
See also	563
Harnessing the power of the GPU with OpenCL	564
Getting ready	564
How to do it...	564
See also	566
Appendix A: Glossary	567
Appendix B: Function Reference	573
IPython	573
Matplotlib	574
NumPy	575
pandas	576
Scikit-learn	577
SciPy	578
Seaborn	578
Statsmodels	579
Appendix C: Online Resources	581
IPython notebooks and open data	581
Mathematics and statistics	582
Presentations	582

Appendix D: Tips and Tricks for Command-line and Miscellaneous Tools	585
IPython notebooks	585
Command-line tools	586
The alias command	586
Command-line history	587
Reproducible sessions	587
Docker tips	588

Module 3: Mastering Python Data Analysis

Preface	1
Chapter 1: Tools of the Trade	7
Before you start	7
Using the notebook interface	9
Imports	10
An example using the Pandas library	10
Summary	18
Chapter 2: Exploring Data	19
The General Social Survey	20
Obtaining the data	20
Reading the data	21
Univariate data	23
Histograms	23
Making things pretty	28
Characterization	29
Concept of statistical inference	32
Numeric summaries and boxplots	33
Relationships between variables – scatterplots	37
Summary	40
Chapter 3: Learning About Models	41
Models and experiments	41
The cumulative distribution function	42
Working with distributions	51
The probability density function	61
Where do models come from?	63
Multivariate distributions	68
Summary	70
Chapter 4: Regression	71
Introducing linear regression	72
Getting the dataset	73
Testing with linear regression	81
Multivariate regression	91
Adding economic indicators	91

Taking a step back	98
Logistic regression	100
Some notes	107
Summary	107
Chapter 5: Clustering	108
Introduction to cluster finding	109
Starting out simple – John Snow on cholera	110
K-means clustering	116
Suicide rate versus GDP versus absolute latitude	116
Hierarchical clustering analysis	122
Reading in and reducing the data	122
Hierarchical cluster algorithm	132
Summary	137
Chapter 6: Bayesian Methods	138
The Bayesian method	138
Credible versus confidence intervals	139
Bayes formula	139
Python packages	140
U.S. air travel safety record	141
Getting the NTSB database	141
Binning the data	147
Bayesian analysis of the data	150
Binning by month	158
Plotting coordinates	160
Cartopy	160
Mpl toolkits – basemap	162
Climate change – CO₂ in the atmosphere	163
Getting the data	164
Creating and sampling the model	166
Summary	173
Chapter 7: Supervised and Unsupervised Learning	174
Introduction to machine learning	174
Scikit-learn	175
Linear regression	176
Climate data	176
Checking with Bayesian analysis and OLS	181
Clustering	183
Seeds classification	188

Visualizing the data	189
Feature selection	194
Classifying the data	196
The SVC linear kernel	198
The SVC Radial Basis Function	199
The SVC polynomial	200
K-Nearest Neighbour	200
Random Forest	201
Choosing your classifier	202
Summary	203
Chapter 8: Time Series Analysis	204
Introduction	204
Pandas and time series data	206
Indexing and slicing	209
Resampling, smoothing, and other estimates	212
Stationarity	218
Patterns and components	220
Decomposing components	221
Differencing	227
Time series models	229
Autoregressive – AR	230
Moving average – MA	232
Selecting p and q	233
Automatic function	234
The (Partial) AutoCorrelation Function	234
Autoregressive Integrated Moving Average – ARIMA	235
Summary	236
Appendix: More on Jupyter Notebook and matplotlib Styles	238
Jupyter Notebook	238
Useful keyboard shortcuts	239
Command mode shortcuts	239
Edit mode shortcuts	239
Markdown cells	240
Notebook Python extensions	241
Installing the extensions	241
Codefolding	243
Collapsible headings	245
Help panel	247
Initialization cells	247
NbExtensions menu item	249

Ruler	249
Skip-traceback	250
Table of contents	252
Other Jupyter Notebook tips	254
External connections	255
Export	255
Additional file types	255
Matplotlib styles	256
Useful resources	261
General resources	261
Packages	262
Data repositories	264
Visualization of data	265
Summary	266
Index	267

Module 1

Getting Started with Python Data Analysis

Learn to use powerful Python libraries for effective data processing and analysis

1

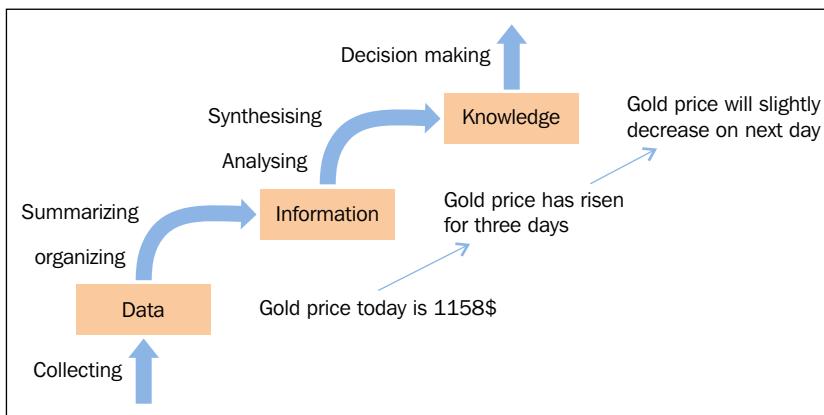
Introducing Data Analysis and Libraries

Data is raw information that can exist in any form, usable or not. We can easily get data everywhere in our lives; for example, the price of gold on the day of writing was \$ 1.158 per ounce. This does not have any meaning, except describing the price of gold. This also shows that data is useful based on context.

With the relational data connection, information appears and allows us to expand our knowledge beyond the range of our senses. When we possess gold price data gathered over time, one piece of information we might have is that the price has continuously risen from \$1.152 to \$1.158 over three days. This could be used by someone who tracks gold prices.

Knowledge helps people to create value in their lives and work. This value is based on information that is organized, synthesized, or summarized to enhance comprehension, awareness, or understanding. It represents a state or potential for action and decisions. When the price of gold continuously increases for three days, it will likely decrease on the next day; this is useful knowledge.

The following figure illustrates the steps from data to knowledge; we call this process, the data analysis process and we will introduce it in the next section:

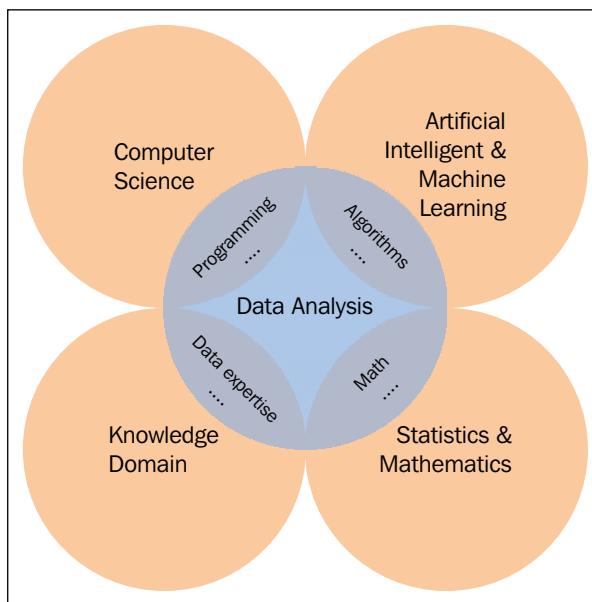


In this chapter, we will cover the following topics:

- Data analysis and process
- An overview of libraries in data analysis using different programming languages
- Common Python data analysis libraries

Data analysis and processing

Data is getting bigger and more diverse every day. Therefore, analyzing and processing data to advance human knowledge or to create value is a big challenge. To tackle these challenges, you will need domain knowledge and a variety of skills, drawing from areas such as computer science, **artificial intelligence (AI)** and **machine learning (ML)**, statistics and mathematics, and knowledge domain, as shown in the following figure:



Let's go through data analysis and its domain knowledge:

- **Computer science:** We need this knowledge to provide abstractions for efficient data processing. Basic Python programming experience is required to follow the next chapters. We will introduce Python libraries used in data analysis.
- **Artificial intelligence and machine learning:** If computer science knowledge helps us to program data analysis tools, artificial intelligence and machine learning help us to model the data and learn from it in order to build smart products.
- **Statistics and mathematics:** We cannot extract useful information from raw data if we do not use statistical techniques or mathematical functions.
- **Knowledge domain:** Besides technology and general techniques, it is important to have an insight into the specific domain. What do the data fields mean? What data do we need to collect? Based on the expertise, we explore and analyze raw data by applying the above techniques, step by step.

Data analysis is a process composed of the following steps:

- **Data requirements:** We have to define what kind of data will be collected based on the requirements or problem analysis. For example, if we want to detect a user's behavior while reading news on the internet, we should be aware of visited article links, dates and times, article categories, and the time the user spends on different pages.
- **Data collection:** Data can be collected from a variety of sources: mobile, personal computer, camera, or recording devices. It may also be obtained in different ways: communication, events, and interactions between person and person, person and device, or device and device. Data appears whenever and wherever in the world. The problem is how we can find and gather it to solve our problem? This is the mission of this step.
- **Data processing:** Data that is initially obtained must be processed or organized for analysis. This process is performance-sensitive. How fast can we create, insert, update, or query data? When building a real product that has to process big data, we should consider this step carefully. What kind of database should we use to store data? What kind of data structure, such as analysis, statistics, or visualization, is suitable for our purposes?
- **Data cleaning:** After being processed and organized, the data may still contain duplicates or errors. Therefore, we need a cleaning step to reduce those situations and increase the quality of the results in the following steps. Common tasks include record matching, deduplication, and column segmentation. Depending on the type of data, we can apply several types of data cleaning. For example, a user's history of visits to a news website might contain a lot of duplicate rows, because the user might have refreshed certain pages many times. For our specific issue, these rows might not carry any meaning when we explore the user's behavior so we should remove them before saving it to our database. Another situation we may encounter is click fraud on news—someone just wants to improve their website ranking or sabotage a website. In this case, the data will not help us to explore a user's behavior. We can use thresholds to check whether a visit page event comes from a real person or from malicious software.
- **Exploratory data analysis:** Now, we can start to analyze data through a variety of techniques referred to as exploratory data analysis. We may detect additional problems in data cleaning or discover requests for further data. Therefore, these steps may be iterative and repeated throughout the whole data analysis process. Data visualization techniques are also used to examine the data in graphs or charts. Visualization often facilitates understanding of data sets, especially if they are large or high-dimensional.

- **Modelling and algorithms:** A lot of mathematical formulas and algorithms may be applied to detect or predict useful knowledge from the raw data. For example, we can use similarity measures to cluster users who have exhibited similar news-reading behavior and recommend articles of interest to them next time. Alternatively, we can detect users' genders based on their news reading behavior by applying classification models such as the **Support Vector Machine (SVM)** or linear regression. Depending on the problem, we may use different algorithms to get an acceptable result. It can take a lot of time to evaluate the accuracy of the algorithms and choose the best one to implement for a certain product.
- **Data product:** The goal of this step is to build data products that receive data input and generate output according to the problem requirements. We will apply computer science knowledge to implement our selected algorithms as well as manage the data storage.

An overview of the libraries in data analysis

There are numerous data analysis libraries that help us to process and analyze data. They use different programming languages, and have different advantages and disadvantages of solving various data analysis problems. Now, we will introduce some common libraries that may be useful for you. They should give you an overview of the libraries in the field. However, the rest of this book focuses on Python-based libraries.

Some of the libraries that use the Java language for data analysis are as follows:

- **Weka:** This is the library that I became familiar with the first time I learned about data analysis. It has a graphical user interface that allows you to run experiments on a small dataset. This is great if you want to get a feel for what is possible in the data processing space. However, if you build a complex product, I think it is not the best choice, because of its performance, sketchy API design, non-optimal algorithms, and little documentation (<http://www.cs.waikato.ac.nz/ml/weka/>).

- **Mallet:** This is another Java library that is used for statistical natural language processing, document classification, clustering, topic modeling, information extraction, and other machine-learning applications on text. There is an add-on package for Mallet, called GRMM, that contains support for inference in general, graphical models, and training of **Conditional random fields (CRF)** with arbitrary graphical structures. In my experience, the library performance and the algorithms are better than Weka. However, its only focus is on text-processing problems. The reference page is at <http://mallet.cs.umass.edu/>.
- **Mahout:** This is Apache's machine-learning framework built on top of Hadoop; its goal is to build a scalable machine-learning library. It looks promising, but comes with all the baggage and overheads of Hadoop. The homepage is at <http://mahout.apache.org/>.
- **Spark:** This is a relatively new Apache project, supposedly up to a hundred times faster than Hadoop. It is also a scalable library that consists of common machine-learning algorithms and utilities. Development can be done in Python as well as in any JVM language. The reference page is at <https://spark.apache.org/docs/1.5.0/mllib-guide.html>.

Here are a few libraries that are implemented in C++:

- **Vowpal Wabbit:** This library is a fast, out-of-core learning system sponsored by Microsoft Research and, previously, Yahoo! Research. It has been used to learn a tera-feature (10¹²) dataset on 1,000 nodes in one hour. More information can be found in the publication at <http://arxiv.org/abs/1110.4198>.
- **MultiBoost:** This package is a multiclass, multi label, and multitask classification boosting software implemented in C++. If you use this software, you should refer to the paper published in 2012 in the *Journal Machine Learning Research, MultiBoost: A Multi-purpose Boosting Package, D.Benbouzid, R. Busa-Fekete, N. Casagrande, F.-D. Collin, and B. Kégl*.
- **MLpack:** This is also a C++ machine-learning library, developed by the **Fundamental Algorithmic and Statistical Tools Laboratory (FASTLab)** at Georgia Tech. It focusses on scalability, speed, and ease-of-use, and was presented at the BigLearning workshop of NIPS 2011. Its homepage is at <http://www.mlpack.org/about.html>.
- **Caffe:** The last C++ library we want to mention is Caffe. This is a deep learning framework made with expression, speed, and modularity in mind. It is developed by the **Berkeley Vision and Learning Center (BVLC)** and community contributors. You can find more information about it at <http://caffe.berkeleyvision.org/>.

Other libraries for data processing and analysis are as follows:

- **Statsmodels:** This is a great Python library for statistical modeling and is mainly used for predictive and exploratory analysis.
- **Modular toolkit for data processing (MDP):** This is a collection of supervised and unsupervised learning algorithms and other data processing units that can be combined into data processing sequences and more complex feed-forward network architectures (<http://mdp-toolkit.sourceforge.net/index.html>).
- **Orange:** This is an open source data visualization and analysis for novices and experts. It is packed with features for data analysis and has add-ons for bioinformatics and text mining. It contains an implementation of self-organizing maps, which sets it apart from the other projects as well (<http://orange.biolab.si/>).
- **Mirador:** This is a tool for the visual exploration of complex datasets, supporting Mac and Windows. It enables users to discover correlation patterns and derive new hypotheses from data (<http://orange.biolab.si/>).
- **RapidMiner:** This is another GUI-based tool for data mining, machine learning, and predictive analysis (<https://rapidminer.com/>).
- **Theano:** This bridges the gap between Python and lower-level languages. Theano gives very significant performance gains, particularly for large matrix operations, and is, therefore, a good choice for deep learning models. However, it is not easy to debug because of the additional compilation layer.
- **Natural language processing toolkit (NLTK):** This is written in Python with very unique and salient features.

Here, I could not list all libraries for data analysis. However, I think the above libraries are enough to take a lot of your time to learn and build data analysis applications. I hope you will enjoy them after reading this book.

Python libraries in data analysis

Python is a multi-platform, general-purpose programming language that can run on Windows, Linux/Unix, and Mac OS X, and has been ported to Java and .NET virtual machines as well. It has a powerful standard library. In addition, it has many libraries for data analysis: Pylearn2, Hebel, Pybrain, Pattern, MontePython, and MILK. In this book, we will cover some common Python data analysis libraries such as Numpy, Pandas, Matplotlib, PyMongo, and scikit-learn. Now, to help you get started, I will briefly present an overview of each library for those who are less familiar with the scientific Python stack.

NumPy

One of the fundamental packages used for scientific computing in Python is Numpy. Among other things, it contains the following:

- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions for performing array computations
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra operations, Fourier transformations, and random number capabilities

Besides this, it can also be used as an efficient multidimensional container of generic data. Arbitrary data types can be defined and integrated with a wide variety of databases.

Pandas

Pandas is a Python package that supports rich data structures and functions for analyzing data, and is developed by the PyData Development Team. It is focused on the improvement of Python's data libraries. Pandas consists of the following things:

- A set of labeled array data structures; the primary of which are Series, DataFrame, and Panel
- Index objects enabling both simple axis indexing and multilevel/hierarchical axis indexing
- An intergraded group by engine for aggregating and transforming datasets
- Date range generation and custom date offsets
- Input/output tools that load and save data from flat files or PyTables/HDF5 format
- Optimal memory versions of the standard data structures
- Moving window statistics and static and moving window linear/panel regression

Due to these features, Pandas is an ideal tool for systems that need complex data structures or high-performance time series functions such as financial data analysis applications.

Matplotlib

Matplotlib is the single most used Python package for 2D-graphics. It provides both a very quick way to visualize data from Python and publication-quality figures in many formats: line plots, contour plots, scatter plots, and Basemap plots. It comes with a set of default settings, but allows customization of all kinds of properties. However, we can easily create our chart with the defaults of almost every property in Matplotlib.

PyMongo

MongoDB is a type of NoSQL database. It is highly scalable, robust, and perfect to work with JavaScript-based web applications, because we can store data as JSON documents and use flexible schemas.

PyMongo is a Python distribution containing tools for working with MongoDB. Many tools have also been written for working with PyMongo to add more features such as MongoKit, Humongolus, MongoAlchemy, and Ming.

The scikit-learn library

The scikit-learn is an open source machine-learning library using the Python programming language. It supports various machine learning models, such as classification, regression, and clustering algorithms, interoperated with the Python numerical and scientific libraries NumPy and SciPy. The latest scikit-learn version is 0.16.1, published in April 2015.

Summary

In this chapter, we presented three main points. Firstly, we figured out the relationship between raw data, information and knowledge. Due to its contribution to our lives, we continued to discuss an overview of data analysis and processing steps in the second section. Finally, we introduced a few common supported libraries that are useful for practical data analysis applications. Among those, in the next chapters, we will focus on Python libraries in data analysis.

Practice exercise

The following table describes users' rankings on Snow White movies:

UserID	Sex	Location	Ranking
A	Male	Philips	4
B	Male	VN	2
C	Male	Canada	1
D	Male	Canada	2
E	Female	VN	5
F	Female	NY	4

Exercise 1: What information can we find in this table? What kind of knowledge can we derive from it?

Exercise 2: Based on the data analysis process in this chapter, try to define the data requirements and analysis steps needed to predict whether user B likes Maleficent movies or not.

2

NumPy Arrays and Vectorized Computation

NumPy is the fundamental package supported for presenting and computing data with high performance in Python. It provides some interesting features as follows:

- Extension package to Python for multidimensional arrays (`ndarrays`), various derived objects (such as masked arrays), matrices providing vectorization operations, and broadcasting capabilities. Vectorization can significantly increase the performance of array computations by taking advantage of **Single Instruction Multiple Data (SIMD)** instruction sets in modern CPUs.
- Fast and convenient operations on arrays of data, including mathematical manipulation, basic statistical operations, sorting, selecting, linear algebra, random number generation, discrete Fourier transforms, and so on.
- Efficiency tools that are closer to hardware because of integrating C/C++/Fortran code.

NumPy is a good starting package for you to get familiar with arrays and array-oriented computing in data analysis. Also, it is the basic step to learn other, more effective tools such as Pandas, which we will see in the next chapter. We will be using NumPy version 1.9.1.

NumPy arrays

An array can be used to contain values of a data object in an experiment or simulation step, pixels of an image, or a signal recorded by a measurement device. For example, the latitude of the Eiffel Tower, Paris is 48.858598 and the longitude is 2.294495. It can be presented in a NumPy array object as p:

```
>>> import numpy as np  
>>> p = np.array([48.858598, 2.294495])  
>>> p  
Output: array([48.858598, 2.294495])
```

This is a manual construction of an array using the np.array function. The standard convention to import NumPy is as follows:

```
>>> import numpy as np
```

You can, of course, put from numpy import * in your code to avoid having to write np. However, you should be careful with this habit because of the potential code conflicts (further information on code conventions can be found in the *Python Style Guide*, also known as **PEP8**, at <https://www.python.org/dev/peps/pep-0008/>).

There are two requirements of a NumPy array: a fixed size at creation and a uniform, fixed data type, with a fixed size in memory. The following functions help you to get information on the p matrix:

```
>>> p.ndim      # getting dimension of array p  
1  
>>> p.shape    # getting size of each array dimension  
(2,)  
>>> len(p)     # getting dimension length of array p  
2  
>>> p.dtype     # getting data type of array p  
dtype('float64')
```

Data types

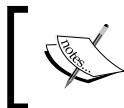
There are five basic numerical types including Booleans (bool), integers (int), unsigned integers (uint), floating point (float), and complex. They indicate how many bits are needed to represent elements of an array in memory. Besides that, NumPy also has some types, such as intc and intp, that have different bit sizes depending on the platform.

See the following table for a listing of NumPy's supported data types:

Type	Type code	Description	Range of value
bool		Boolean stored as a byte	True/False
intc		Similar to C int (int32 or int 64)	
intp		Integer used for indexing (same as C size_t)	
int8, uint8	i1, u1	Signed and unsigned 8-bit integer types	int8: (-128 to 127) uint8: (0 to 255)
int16, uint16	i2, u2	Signed and unsigned 16-bit integer types	int16: (-32768 to 32767) uint16: (0 to 65535)
int32, uint32	i4, u4	Signed and unsigned 32-bit integer types	int32: (-2147483648 to 2147483647) uint32: (0 to 4294967295)
int64, uint64	i8, u8	Signed and unsigned 64-bit integer types	Int64: (-9223372036854775808 to 9223372036854775807) uint64: (0 to 18446744073709551615)
float16	f2	Half precision float: sign bit, 5 bits exponent, and 10b bits mantissa	
float32	f4 / f	Single precision float: sign bit, 8 bits exponent, and 23 bits mantissa	
float64	f8 / d	Double precision float: sign bit, 11 bits exponent, and 52 bits mantissa	
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32-bit, 64-bit, and 128-bit floats	
object	0	Python object type	
string_	S	Fixed-length string type	Declare a string dtype with length 10, using S10
unicode_	U	Fixed-length Unicode type	Similar to string_ example, we have 'U10'

We can easily convert or cast an array from one `dtype` to another using the `astype` method:

```
>>> a = np.array([1, 2, 3, 4])
>>> a.dtype
dtype('int64')
>>> float_b = a.astype(np.float64)
>>> float_b.dtype
dtype('float64')
```



The `astype` function will create a new array with a copy of data from an old array, even though the new `dtype` is similar to the old one.



Array creation

There are various functions provided to create an array object. They are very useful for us to create and store data in a multidimensional array in different situations.

Now, in the following table we will summarize some of NumPy's common functions and their use by examples for array creation:

Function	Description	Example
<code>empty</code> , <code>empty_like</code>	Create a new array of the given shape and type, without initializing elements	<pre>>>> np.empty([3,2], dtype=np.float64) array([[0., 0.], [0., 0.], [0., 0.]]) >>> a = np.array([[1, 2], [4, 3]]) >>> np.empty_like(a) array([[0, 0], [0, 0]])</pre>
<code>eye</code> , <code>identity</code>	Create a NxN identity matrix with ones on the diagonal and zero elsewhere	<pre>>>> np.eye(2, dtype=np.int) array([[1, 0], [0, 1]])</pre>
<code>ones</code> , <code>ones_like</code>	Create a new array with the given shape and type, filled with 1s for all elements	<pre>>>> np.ones(5) array([1., 1., 1., 1., 1.]) >>> np.ones(4, dtype=np.int) array([1, 1, 1, 1]) >>> x = np.array([[0,1,2], [3,4,5]]) >>> np.ones_like(x) array([[1, 1, 1], [1, 1, 1]])</pre>

Function	Description	Example
zeros, zeros_like	This is similar to ones, ones_like, but initializing elements with 0s instead	<pre>>>> np.zeros(5) array([0., 0., 0., 0., 0.]) >>> np.zeros(4, dtype=np.int) array([0, 0, 0, 0]) >>> x = np.array([[0, 1, 2], [3, 4, 5]]) >>> np.zeros_like(x) array([[0, 0, 0], [0, 0, 0]])</pre>
arange	Create an array with even spaced values in a given interval	<pre>>>> np.arange(2, 5) array([2, 3, 4]) >>> np.arange(4, 12, 5) array([4, 9])</pre>
full, full_like	Create a new array with the given shape and type, filled with a selected value	<pre>>>> np.full((2,2), 3, dtype=np.int) array([[3, 3], [3, 3]]) >>> x = np.ones(3) >>> np.full_like(x, 2) array([2., 2., 2.])</pre>
array	Create an array from the existing data	<pre>>>> np.array([[1.1, 2.2, 3.3], [4.4, 5.5, 6.6]]) array([1.1, 2.2, 3.3], [4.4, 5.5, 6.6])</pre>
asarray	Convert the input to an array	<pre>>>> a = [3.14, 2.46] >>> np.asarray(a) array([3.14, 2.46])</pre>
copy	Return an array copy of the given object	<pre>>>> a = np.array([[1, 2], [3, 4]]) >>> np.copy(a) array([[1, 2], [3, 4]])</pre>
fromstring	Create 1-D array from a string or text	<pre>>>> np.fromstring('3.14 2.17', dtype=np.float, sep=' ') array([3.14, 2.17])</pre>

Indexing and slicing

As with other Python sequence types, such as lists, it is very easy to access and assign a value of each array's element:

```
>>> a = np.arange(7)
>>> a
array([0, 1, 2, 3, 4, 5, 6])
>>> a[1], a[4], a[-1]
(1, 4, 6)
```



In Python, array indices start at 0. This is in contrast to Fortran or Matlab, where indices begin at 1.



As another example, if our array is multidimensional, we need tuples of integers to index an item:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> a[0, 2]      # first row, third column
3
>>> a[0, 2] = 10
>>> a
array([[1, 2, 10], [4, 5, 6], [7, 8, 9]])
>>> b = a[2]
>>> b
array([7, 8, 9])
>>> c = a[:2]
>>> c
array([[1, 2, 10], [4, 5, 6]])
```

We call `b` and `c` as array slices, which are views on the original one. It means that the data is not copied to `b` or `c`, and whenever we modify their values, it will be reflected in the array `a` as well:

```
>>> b[-1] = 11
>>> a
array([[1, 2, 10], [4, 5, 6], [7, 8, 11]])
```



We use a colon (:) character to take the entire axis when we omit the index number.



Fancy indexing

Besides indexing with slices, NumPy also supports indexing with Boolean or integer arrays (masks). This method is called **fancy indexing**. It creates copies, not views.

First, we take a look at an example of indexing with a Boolean mask array:

```
>>> a = np.array([3, 5, 1, 10])
>>> b = (a % 5 == 0)
>>> b
array([False, True, False, True], dtype=bool)
>>> c = np.array([[0, 1], [2, 3], [4, 5], [6, 7]])
>>> c[b]
array([[2, 3], [6, 7]])
```

The second example is an illustration of using integer masks on arrays:

```
>>> a = np.array([[1, 2, 3, 4],
   [5, 6, 7, 8],
   [9, 10, 11, 12],
   [13, 14, 15, 16]])
>>> a[[2, 1]]
array([[9, 10, 11, 12], [5, 6, 7, 8]])
>>> a[[-2, -1]]          # select rows from the end
array([[9, 10, 11, 12], [13, 14, 15, 16]])
>>> a[[2, 3], [0, 1]]    # take elements at (2, 0) and (3, 1)
array([9, 14])
```



The mask array must have the same length as the axis that it's indexing.



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Numerical operations on arrays

We are getting familiar with creating and accessing ndarrays. Now, we continue to the next step, applying some mathematical operations to array data without writing any for loops, of course, with higher performance.

Scalar operations will propagate the value to each element of the array:

```
>>> a = np.ones(4)
>>> a * 2
array([2., 2., 2., 2.])
>>> a + 3
array([4., 4., 4., 4.])
```

All arithmetic operations between arrays apply the operation element wise:

```
>>> a = np.ones([2, 4])
>>> a * a
array([[1., 1., 1., 1.], [1., 1., 1., 1.]])
>>> a + a
array([[2., 2., 2., 2.], [2., 2., 2., 2.]])
```

Also, here are some examples of comparisons and logical operations:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([1, 1, 5, 3])
>>> a == b
array([True, False, False, False], dtype=bool)

>>> np.array_equal(a, b)      # array-wise comparison
False

>>> c = np.array([1, 0])
>>> d = np.array([1, 1])
>>> np.logical_and(c, d)      # logical operations
array([True, False])
```

Array functions

Many helpful array functions are supported in NumPy for analyzing data. We will list some part of them that are common in use. Firstly, the transposing function is another kind of reshaping form that returns a view on the original data array without copying anything:

```
>>> a = np.array([[0, 5, 10], [20, 25, 30]])  
>>> a.reshape(3, 2)  
array([[0, 5], [10, 20], [25, 30]])  
  
>>> a.T  
array([[0, 20], [5, 25], [10, 30]])
```

In general, we have the `swapaxes` method that takes a pair of axis numbers and returns a view on the data, without making a copy:

```
>>> a = np.array([[[0, 1, 2], [3, 4, 5]],  
[[6, 7, 8], [9, 10, 11]]])  
>>> a.swapaxes(1, 2)  
array([[[0, 3],  
[1, 4],  
[2, 5]],  
[[6, 9],  
[7, 10],  
[8, 11]])
```

The transposing function is used to do matrix computations; for example, computing the inner matrix product $\mathbf{X}^T \cdot \mathbf{X}$ using `np.dot`:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])  
>>> np.dot(a.T, a)  
array([[17, 22, 27],  
[22, 29, 36],  
[27, 36, 45]])
```

Sorting data in an array is also an important demand in processing data. Let's take a look at some sorting functions and their use:

```
>>> a = np.array ([[6, 34, 1, 6], [0, 5, 2, -1]])  
  
>>> np.sort(a)      # sort along the last axis  
array([[1, 6, 6, 34], [-1, 0, 2, 5]])  
  
>>> np.sort(a, axis=0)    # sort along the first axis  
array([[0, 5, 1, -1], [6, 34, 2, 6]])  
  
>>> b = np.argsort(a)    # fancy indexing of sorted array  
>>> b  
array([[2, 0, 3, 1], [3, 0, 2, 1]])  
>>> a[0][b[0]]  
array([1, 6, 6, 34])  
  
>>> np.argmax(a)      # get index of maximum element  
1
```

See the following table for a listing of array functions:

Function	Description	Example
sin, cos, tan, cosh, sinh, tanh, arcos, arctan, deg2rad	Trigonometric and hyperbolic functions	<pre>>>> a = np.array([0., 30., 45.]) >>> np.sin(a * np.pi / 180) array([0., 0.5, 0.7071678])</pre>
around, round, rint, fix, floor, ceil, trunc	Rounding elements of an array to the given or nearest number	<pre>>>> a = np.array([0.34, 1.65]) >>> np.round(a) array([0., 2.])</pre>
sqrt, square, exp, expm1, exp2, log, log10, log1p, logaddexp	Computing the exponents and logarithms of an array	<pre>>>> np.exp(np.array([2.25, 3.16])) array([9.4877, 23.5705])</pre>

Function	Description	Example
add, negative, multiply, devide, power, subtract, mod, modf, remainder	Set of arithmetic functions on arrays	<pre>>>> a = np.arange(6) >>> x1 = a.reshape(2,3) >>> x2 = np.arange(3) >>> np.multiply(x1, x2) array([[0,1,4], [0,4,10]])</pre>
greater, greater_equal, less, less_equal, equal, not_equal	Perform elementwise comparison: >, >=, <, <=, ==, !=	<pre>>>> np.greater(x1, x2) array([[False, False, False], [True, True, True]], dtype = bool)</pre>

Data processing using arrays

With the NumPy package, we can easily solve many kinds of data processing tasks without writing complex loops. It is very helpful for us to control our code as well as the performance of the program. In this part, we want to introduce some mathematical and statistical functions.

See the following table for a listing of mathematical and statistical functions:

Function	Description	Example
sum	Calculate the sum of all the elements in an array or along the axis	<pre>>>> a = np.array([[2,4], [3,5]]) >>> np.sum(a, axis=0) array([5, 9])</pre>
prod	Compute the product of array elements over the given axis	<pre>>>> np.prod(a, axis=1) array([8, 15])</pre>
diff	Calculate the discrete difference along the given axis	<pre>>>> np.diff(a, axis=0) array([[1,1]])</pre>
gradient	Return the gradient of an array	<pre>>>> np.gradient(a) [array([[1., 1.], [1., 1.]]), array([[2., 2.], [2., 2.]])]</pre>
cross	Return the cross product of two arrays	<pre>>>> b = np.array([[1,2], [3,4]]) >>> np.cross(a,b) array([0, -3])</pre>

Function	Description	Example
std, var	Return standard deviation and variance of arrays	<pre>>>> np.std(a) 1.1180339 >>> np.var(a) 1.25</pre>
mean	Calculate arithmetic mean of an array	<pre>>>> np.mean(a) 3.5</pre>
where	Return elements, either from x or y, that satisfy a condition	<pre>>>> np.where([[True, True], [False, True]], [[1,2],[3,4]], [[5,6],[7,8]]) array([[1,2], [7, 4]])</pre>
unique	Return the sorted unique values in an array	<pre>>>> id = np.array(['a', 'b', 'c', 'c', 'd']) >>> np.unique(id) array(['a', 'b', 'c', 'd'], dtype=' S1')</pre>
intersect1d	Compute the sorted and common elements in two arrays	<pre>>>> a = np.array(['a', 'b', 'a', 'c', 'd', 'c']) >>> b = np.array(['a', 'xyz', 'klm', 'd']) >>> np.intersect1d(a,b) array(['a', 'd'], dtype=' S3')</pre>

Loading and saving data

We can also save and load data to and from a disk, either in text or binary format, by using different supported functions in NumPy package.

Saving an array

Arrays are saved by default in an uncompressed raw binary format, with the file extension .npy by the np.save function:

```
>>> a = np.array([[0, 1, 2], [3, 4, 5]])
>>> np.save('test1.npy', a)
```



The library automatically assigns the .npy extension, if we omit it.



If we want to store several arrays into a single file in an uncompressed .npz format, we can use the np.savez function, as shown in the following example:

```
>>> a = np.arange(4)
>>> b = np.arange(7)
>>> np.savez('test2.npz', arr0=a, arr1=b)
```

The .npz file is a zipped archive of files named after the variables they contain. When we load an .npz file, we get back a dictionary-like object that can be queried for its lists of arrays:

```
>>> dic = np.load('test2.npz')
>>> dic['arr0']
array([0, 1, 2, 3])
```

Another way to save array data into a file is using the np.savetxt function that allows us to set format properties in the output file:

```
>>> x = np.arange(4)
>>> # e.g., set comma as separator between elements
>>> np.savetxt('test3.out', x, delimiter=',')
```

Loading an array

We have two common functions such as np.load and np.loadtxt, which correspond to the saving functions, for loading an array:

```
>>> np.load('test1.npy')
array([[0, 1, 2], [3, 4, 5]])
>>> np.loadtxt('test3.out', delimiter=',')
array([0., 1., 2., 3.])
```

Similar to the np.savetxt function, the np.loadtxt function also has a lot of options for loading an array from a text file.

Linear algebra with NumPy

Linear algebra is a branch of mathematics concerned with vector spaces and the mappings between those spaces. NumPy has a package called **linalg** that supports powerful linear algebra functions. We can use these functions to find eigenvalues and eigenvectors or to perform singular value decomposition:

```
>>> A = np.array([[1, 4, 6],  
... [5, 2, 2],  
... [-1, 6, 8]])  
  
>>> w, v = np.linalg.eig(A)  
          # eigenvalues  
array([-0.111 + 1.5756j, -0.111 - 1.5756j, 11.222+0.j])  
  
>>> v          # eigenvector  
array([[[-0.0981 + 0.2726j, -0.0981 - 0.2726j, 0.5764+0.j],  
... [0.7683+0.j, 0.7683-0.j, 0.4591+0.j],  
... [-0.5656 - 0.0762j, -0.5656 + 0.00763j, 0.6759+0.j]])
```

The function is implemented using the geev Lapack routines that compute the eigenvalues and eigenvectors of general square matrices.

Another common problem is solving linear systems such as $Ax = b$ with A as a matrix and x and b as vectors. The problem can be solved easily using the `numpy.linalg.solve` function:

```
>>> A = np.array([[1, 4, 6], [5, 2, 2], [-1, 6, 8]])  
  
>>> b = np.array([1, 2, 3])  
  
>>> x = np.linalg.solve(A, b)  
  
>>> x  
array([-1.77635e-16, 2.5, -1.5])
```

The following table will summarise some commonly used functions in the `numpy.linalg` package:

Function	Description	Example
dot	Calculate the dot product of two arrays	<pre>>>> a = np.array([[1, 0], [0, 1]]) >>> b = np.array([[4, 1], [2, 2]]) >>> np.dot(a,b) array([[4, 1], [2, 2]])</pre>

Function	Description	Example
inner, outer	Calculate the inner and outer product of two arrays	<pre>>>> a = np.array([1, 1, 1]) >>> b = np.array([3, 5, 1]) >>> np.inner(a,b) 9</pre>
linalg.norm	Find a matrix or vector norm	<pre>>>> a = np.arange(3) >>> np.linalg.norm(a) 2.23606</pre>
linalg.det	Compute the determinant of an array	<pre>>>> a = np.array([[1,2],[3,4]]) >>> np.linalg.det(a) -2.0</pre>
linalg.inv	Compute the inverse of a matrix	<pre>>>> a = np.array([[1,2],[3,4]]) >>> np.linalg.inv(a) array([[-2., 1.], [1.5, -0.5]])</pre>
linalg.qr	Calculate the QR decomposition	<pre>>>> a = np.array([[1,2],[3,4]]) >>> np.linalg.qr(a) (array([[0.316, 0.948], [0.948, 0.316]]), array([[3.162, 4.427], [0., 0.632]]))</pre>
linalg.cond	Compute the condition number of a matrix	<pre>>>> a = np.array([[1,3],[2,4]]) >>> np.linalg.cond(a) 14.933034</pre>
trace	Compute the sum of the diagonal element	<pre>>>> np.trace(np.arange(6)). reshape(2,3)) 4</pre>

NumPy random numbers

An important part of any simulation is the ability to generate random numbers. For this purpose, NumPy provides various routines in the submodule `random`. It uses a particular algorithm, called the Mersenne Twister, to generate pseudorandom numbers.

First, we need to define a seed that makes the random numbers predictable. When the value is reset, the same numbers will appear every time. If we do not assign the seed, NumPy automatically selects a random seed value based on the system's random number generator device or on the clock:

```
>>> np.random.seed(20)
```

An array of random numbers in the [0.0, 1.0] interval can be generated as follows:

```
>>> np.random.rand(5)
array([0.5881308, 0.89771373, 0.89153073, 0.81583748,
       0.03588959])
>>> np.random.rand(5)
array([0.69175758, 0.37868094, 0.51851095, 0.65795147,
       0.19385022])
>>> np.random.seed(20)      # reset seed number
>>> np.random.rand(5)
array([0.5881308, 0.89771373, 0.89153073, 0.81583748,
       0.03588959])
```

If we want to generate random integers in the half-open interval [min, max], we can user the randint(min, max, length) function:

```
>>> np.random.randint(10, 20, 5)
array([17, 12, 10, 16, 18])
```

NumPy also provides for many other distributions, including the Beta, binomial, chi-square, Dirichlet, exponential, F, Gamma, geometric, or Gumbel.

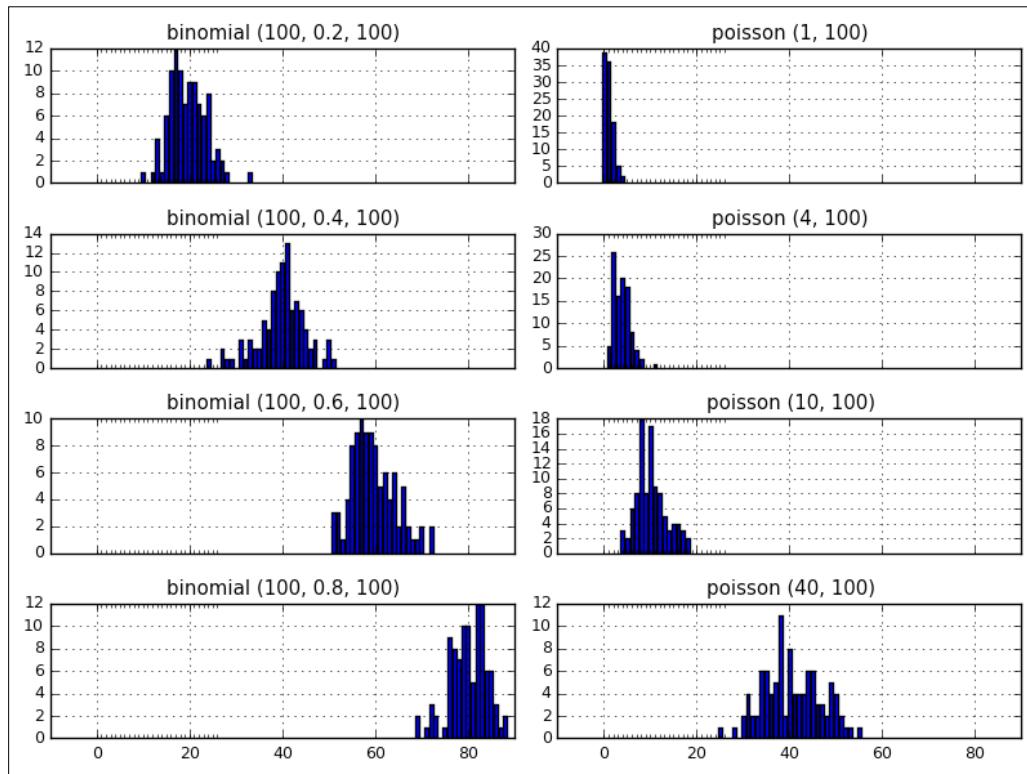
The following table will list some distribution functions and give examples for generating random numbers:

Function	Description	Example
binomial	Draw samples from a binomial distribution (n: number of trials, p: probability)	<pre>>>> n, p = 100, 0.2 >>> np.random.binomial(n, p, 3) array([17, 14, 23])</pre>
dirichlet	Draw samples using a Dirichlet distribution	<pre>>>> np.random. dirichlet(alpha=(2,3), size=3) array([[0.519, 0.480], [0.639, 0.36], [0.838, 0.161]])</pre>
poisson	Draw samples from a Poisson distribution	<pre>>>> np.random.poisson(lam=2, size= 2) array([4,1])</pre>
normal	Draw samples using a normal Gaussian distribution	<pre>>>> np.random.normal (loc=2.5, scale=0.3, size=3) array([2.4436, 2.849, 2.741])</pre>
uniform	Draw samples using a uniform distribution	<pre>>>> np.random.uniform(low=0.5, high=2.5, size=3) array([1.38, 1.04, 2.19])</pre>

We can also use the random number generation to shuffle items in a list. Sometimes this is useful when we want to sort a list in a random order:

```
>>> a = np.arange(10)
>>> np.random.shuffle(a)
>>> a
array([7, 6, 3, 1, 4, 2, 5, 0, 9, 8])
```

The following figure shows two distributions, `binomial` and `poisson`, side by side with various parameters (the visualization was created with `matplotlib`, which will be covered in *Chapter 4, Data Visualization*):



Summary

In this chapter, we covered a lot of information related to the NumPy package, especially commonly used functions that are very helpful to process and analyze data in `ndarray`. Firstly, we learned the properties and data type of `ndarray` in the NumPy package. Secondly, we focused on how to create and manipulate an `ndarray` in different ways, such as conversion from other structures, reading an array from disk, or just generating a new array with given values. Thirdly, we studied how to access and control the value of each element in `ndarray` by using indexing and slicing.

Then, we are getting familiar with some common functions and operations on ndarray.

And finally, we continue with some advance functions that are related to statistic, linear algebra and sampling data. Those functions play important role in data analysis.

However, while NumPy by itself does not provide very much high-level data analytical functionality, having an understanding of it will help you use tools such as Pandas much more effectively. This tool will be discussed in the next chapter.

Practice exercises

Exercise 1: Using an array creation function, let's try to create arrays variable in the following situations:

- Create ndarray from the existing data
- Initialize ndarray which elements are filled with ones, zeros, or a given interval
- Loading and saving data from a file to an ndarray

Exercise 2: What is the difference between `np.dot(a, b)` and `(a*b)`?

Exercise 3: Consider the vector [1, 2, 3, 4, 5] building a new vector with four consecutive zeros interleaved between each value.

Exercise 4: Taking the data example file `chapter2-data.txt`, which includes information on a system log, solves the following tasks:

- Try to build an ndarray from the data file
- Statistic frequency of each device type in the built matrix
- List unique OS that appears in the data log
- Sort user by provinceID and count the number of users in each province

3

Data Analysis with Pandas

In this chapter, we will explore another data analysis library called Pandas. The goal of this chapter is to give you some basic knowledge and concrete examples for getting started with Pandas.

An overview of the Pandas package

Pandas is a Python package that supports fast, flexible, and expressive data structures, as well as computing functions for data analysis. The following are some prominent features that Pandas supports:

- Data structure with labeled axes. This makes the program clean and clear and avoids common errors from misaligned data.
- Flexible handling of missing data.
- Intelligent label-based slicing, fancy indexing, and subset creation of large datasets.
- Powerful arithmetic operations and statistical computations on a custom axis via axis label.
- Robust input and output support for loading or saving data from and to files, databases, or HDF5 format.

Related to Pandas installation, we recommend an easy way, that is to install it as a part of Anaconda, a cross-platform distribution for data analysis and scientific computing. You can refer to the reference at <http://docs.continuum.io/anaconda/> to download and install the library.

After installation, we can use it like other Python packages. Firstly, we have to import the following packages at the beginning of the program:

```
>>> import pandas as pd  
>>> import numpy as np
```

The Pandas data structure

Let's first get acquainted with two of Pandas' primary data structures: the Series and the DataFrame. They can handle the majority of use cases in finance, statistic, social science, and many areas of engineering.

Series

A Series is a one-dimensional object similar to an array, list, or column in table. Each item in a Series is assigned to an entry in an index:

```
>>> s1 = pd.Series(np.random.rand(4),  
                 index=['a', 'b', 'c', 'd'])  
  
>>> s1  
  
a    0.6122  
b    0.98096  
c    0.3350  
d    0.7221  
  
dtype: float64
```

By default, if no index is passed, it will be created to have values ranging from 0 to N-1, where N is the length of the Series:

```
>>> s2 = pd.Series(np.random.rand(4))  
  
>>> s2  
  
0    0.6913  
1    0.8487  
2    0.8627  
3    0.7286  
  
dtype: float64
```

We can access the value of a Series by using the index:

```
>>> s1['c']
0.3350
>>>s1['c'] = 3.14
>>> s1['c', 'a', 'b']
c    3.14
a    0.6122
b    0.98096
```

This accessing method is similar to a Python dictionary. Therefore, Pandas also allows us to initialize a Series object directly from a Python dictionary:

```
>>> s3 = pd.Series({'001': 'Nam', '002': 'Mary',
                   '003': 'Peter'})
>>> s3
001    Nam
002    Mary
003    Peter
dtype: object
```

Sometimes, we want to filter or rename the index of a Series created from a Python dictionary. At such times, we can pass the selected index list directly to the initial function, similarly to the process in the above example. Only elements that exist in the index list will be in the Series object. Conversely, indexes that are missing in the dictionary are initialized to default NaN values by Pandas:

```
>>> s4 = pd.Series({'001': 'Nam', '002': 'Mary',
                   '003': 'Peter'}, index=[
                   '002', '001', '024', '065'])
>>> s4
002    Mary
001    Nam
024    NaN
065    NaN
dtype: object
ect
```

The library also supports functions that detect missing data:

```
>>> pd.isnull(s4)
002    False
001    False
024    True
065    True
dtype: bool
```

Similarly, we can also initialize a Series from a scalar value:

```
>>> s5 = pd.Series(2.71, index=['x', 'y'])
>>> s5
x    2.71
y    2.71
dtype: float64
```

A Series object can be initialized with NumPy objects as well, such as ndarray. Moreover, Pandas can automatically align data indexed in different ways in arithmetic operations:

```
>>> s6 = pd.Series(np.array([2.71, 3.14]), index=['z', 'y'])
>>> s6
z    2.71
y    3.14
dtype: float64
>>> s5 + s6
x    NaN
y    5.85
z    NaN
dtype: float64
```

The DataFrame

The DataFrame is a tabular data structure comprising a set of ordered columns and rows. It can be thought of as a group of Series objects that share an index (the column names). There are a number of ways to initialize a DataFrame object. Firstly, let's take a look at the common example of creating DataFrame from a dictionary of lists:

```
>>> data = {'Year': [2000, 2005, 2010, 2014],
           'Median_Age': [24.2, 26.4, 28.5, 30.3],
```

```
'Density': [244, 256, 268, 279]}

>>> df1 = pd.DataFrame(data)

>>> df1

   Density  Median_Age    Year
0     244        24.2  2000
1     256        26.4  2005
2     268        28.5  2010
3     279        30.3  2014
```

By default, the DataFrame constructor will order the column alphabetically. We can edit the default order by passing the column's attribute to the initializing function:

```
>>> df2 = pd.DataFrame(data, columns=['Year', 'Density',
                                         'Median_Age'])

>>> df2

   Year  Density  Median_Age
0   2000     244        24.2
1   2005     256        26.4
2   2010     268        28.5
3   2014     279        30.3

>>> df2.index

Int64Index([0, 1, 2, 3], dtype='int64')
```

We can provide the index labels of a DataFrame similar to a Series:

```
>>> df3 = pd.DataFrame(data, columns=['Year', 'Density',
                                         'Median_Age'], index=['a', 'b', 'c', 'd'])

>>> df3.index

Index(['a', 'b', 'c', 'd'], dtype='object')
```

We can construct a DataFrame out of nested lists as well:

```
>>> df4 = pd.DataFrame([
    ['Peter', 16, 'pupil', 'TN', 'M', None],
    ['Mary', 21, 'student', 'SG', 'F', None],
    ['Nam', 22, 'student', 'HN', 'M', None],
    ['Mai', 31, 'nurse', 'SG', 'F', None],
    ['John', 28, 'laywer', 'SG', 'M', None]],
columns=['name', 'age', 'career', 'province', 'sex', 'award'])
```

Columns can be accessed by column name as a Series can, either by dictionary-like notation or as an attribute, if the column name is a syntactically valid attribute name:

```
>>> df4.name    # or df4['name']

0    Peter
1    Mary
2    Nam
3    Mai
4    John

Name: name, dtype: object
```

To modify or append a new column to the created DataFrame, we specify the column name and the value we want to assign:

```
>>> df4['award'] = None

>>> df4

   name  age   career  province  sex  award
0  Peter   16    pupil        TN    M  None
1   Mary   21  student        SG    F  None
2    Nam   22  student        HN    M  None
3    Mai   31    nurse        SG    F  None
4   John   28    lawer        SG    M  None
```

Using a couple of methods, rows can be retrieved by position or name:

```
>>> df4.ix[1]

name          Mary
age           21
career       student
province      SG
sex            F
award         None

Name: 1, dtype: object
```

A DataFrame object can also be created from different data structures such as a list of dictionaries, a dictionary of Series, or a record array. The method to initialize a DataFrame object is similar to the examples above.

Another common case is to provide a DataFrame with data from a location such as a text file. In this situation, we use the `read_csv` function that expects the column separator to be a comma, by default. However, we can change that by using the `sep` parameter:

```
# person.csv file
name,age,career,province,sex
Peter,16,pupil,TN,M
Mary,21,student,SG,F
Nam,22,student,HN,M
Mai,31,nurse,SG,F
John,28,lawer,SG,M
# loading person.csv into a DataFrame
>>> df4 = pd.read_csv('person.csv')
>>> df4
   name  age  career  province  sex
0  Peter   16    pupil      TN     M
1   Mary   21  student      SG     F
2    Nam   22  student      HN     M
3    Mai   31    nurse      SG     F
4   John   28  laywer      SG     M
```

While reading a data file, we sometimes want to skip a line or an invalid value. As for Pandas 0.16.2, `read_csv` supports over 50 parameters for controlling the loading process. Some common useful parameters are as follows:

- `sep`: This is a delimiter between columns. The default is comma symbol.
- `dtype`: This is a data type for data or columns.
- `header`: This sets row numbers to use as the column names.
- `skiprows`: This skips line numbers to skip at the start of the file.
- `error_bad_lines`: This shows invalid lines (too many fields) that will, by default, cause an exception, such that no DataFrame will be returned. If we set the value of this parameter as `false`, the bad lines will be skipped.

Moreover, Pandas also has support for reading and writing a DataFrame directly from or to a database such as the `read_frame` or `write_frame` function within the Pandas module. We will come back to these methods later in this chapter.

The essential basic functionality

Pandas supports many essential functionalities that are useful to manipulate Pandas data structures. In this book, we will focus on the most important features regarding exploration and analysis.

Reindexing and altering labels

Reindex is a critical method in the Pandas data structures. It confirms whether the new or modified data satisfies a given set of labels along a particular axis of Pandas object.

First, let's view a reindex example on a Series object:

```
>>> s2.reindex([0, 2, 'b', 3])
0    0.6913
2    0.8627
b    NaN
3    0.7286
dtype: float64
```

When reindexed labels do not exist in the data object, a default value of `NaN` will be automatically assigned to the position; this holds true for the DataFrame case as well:

```
>>> df1.reindex(index=[0, 2, 'b', 3],
                 columns=['Density', 'Year', 'Median_Age','C'])
   Density  Year  Median_Age      C
0      244  2000       24.2    NaN
2      268  2010       28.5    NaN
b      NaN    NaN       NaN    NaN
3      279  2014       30.3    NaN
```

We can change the `NaN` value in the missing index case to a custom value by setting the `fill_value` parameter. Let us take a look at the arguments that the `reindex` function supports, as shown in the following table:

Argument	Description
<code>index</code>	This is the new labels/index to conform to.
<code>method</code>	This is the method to use for filling holes in a reindexed object. The default setting is <code>unfill gaps</code> . <code>pad/ffill</code> : fill values forward <code>backfill/bfill</code> : fill values backward <code>nearest</code> : use the nearest value to fill the gap
<code>copy</code>	This return a new object. The default setting is <code>true</code> .
<code>level</code>	The matches index values on the passed multiple index level.
<code>fill_value</code>	This is the value to use for missing values. The default setting is <code>NaN</code> .
<code>limit</code>	This is the maximum size gap to fill in forward or backward method.

Head and tail

In common data analysis situations, our data structure objects contain many columns and a large number of rows. Therefore, we cannot view or load all information of the objects. Pandas supports functions that allow us to inspect a small sample. By default, the functions return five elements, but we can set a custom number as well. The following example shows how to display the first five and the last three rows of a longer Series:

```
>>> s7 = pd.Series(np.random.rand(10000))
>>> s7.head()
0    0.631059
1    0.766085
2    0.066891
3    0.867591
4    0.339678
```

```
dtype: float64
>>> s7.tail(3)
9997    0.412178
9998    0.800711
9999    0.438344
dtype: float64
```

We can also use these functions for DataFrame objects in the same way.

Binary operations

Firstly, we will consider arithmetic operations between objects. In different indexes objects case, the expected result will be the union of the index pairs. We will not explain this again because we had an example about it in the above section (s5 + s6). This time, we will show another example with a DataFrame:

```
>>> df5 = pd.DataFrame(np.arange(9).reshape(3,3),
                      columns=['a','b','c'])

>>> df5
   a   b   c
0  0   1   2
1  3   4   5
2  6   7   8

>>> df6 = pd.DataFrame(np.arange(8).reshape(2,4),
                      columns=['a','b','c','d'])

>>> df6
   a   b   c   d
0  0   1   2   3
1  4   5   6   7

>>> df5 + df6
      a   b   c   d
0  0.0  2.0  4.0  NaN
1  7.0  9.0  11.0  NaN
2  NaN  NaN  NaN  NaN
```

The mechanisms for returning the result between two kinds of data structure are similar. A problem that we need to consider is the missing data between objects. In this case, if we want to fill with a fixed value, such as 0, we can use the arithmetic functions such as add, sub, div, and mul, and the function's supported parameters such as fill_value:

```
>>> df7 = df5.add(df6, fill_value=0)
>>> df7
   a   b   c   d
0  0   2   4   3
1  7   9  11   7
2  6   7   8   NaN
```

Next, we will discuss comparison operations between data objects. We have some supported functions such as equal (eq), not equal (ne), greater than (gt), less than (lt), less equal (le), and greater equal (ge). Here is an example:

```
>>> df5.eq(df6)
      a      b      c      d
0  True  True  True  False
1 False  False  False  False
2 False  False  False  False
```

Functional statistics

The supported statistics method of a library is really important in data analysis. To get inside a big data object, we need to know some summarized information such as mean, sum, or quantile. Pandas supports a large number of methods to compute them. Let's consider a simple example of calculating the sum information of df5, which is a DataFrame object:

```
>>> df5.sum()
a    9
b   12
c   15
dtype: int64
```

When we do not specify which axis we want to calculate `sum` information, by default, the function will calculate on index axis, which is axis 0:

- **Series:** We do not need to specify the axis.
- **DataFrame:** Columns (`axis = 1`) or index (`axis = 0`). The default setting is axis 0.

We also have the `skipna` parameter that allows us to decide whether to exclude missing data or not. By default, it is set as `true`:

```
>>> df7.sum(skipna=False)
a    13
b    18
c    23
d    NaN
dtype: float64
```

Another function that we want to consider is `describe()`. It is very convenient for us to summarize most of the statistical information of a data structure such as the Series and DataFrame, as well:

```
>>> df5.describe()
      a    b    c
count  3.0  3.0  3.0
mean   3.0  4.0  5.0
std    3.0  3.0  3.0
min    0.0  1.0  2.0
25%   1.5  2.5  3.5
50%   3.0  4.0  5.0
75%   4.5  5.5  6.5
max   6.0  7.0  8.0
```

We can specify percentiles to include or exclude in the output by using the `percentiles` parameter; for example, consider the following:

```
>>> df5.describe(percentiles=[0.5, 0.8])
      a    b    c
count  3.0  3.0  3.0
mean   3.0  4.0  5.0
std    3.0  3.0  3.0
```

```
min      0.0  1.0  2.0
50%     3.0  4.0  5.0
80%     4.8  5.8  6.8
max     6.0  7.0  8.0
```

Here, we have a summary table for common supported statistics functions in Pandas:

Function	Description
<code>idxmin(axis), idxmax(axis)</code>	This compute the index labels with the minimum or maximum corresponding values.
<code>value_counts()</code>	This compute the frequency of unique values.
<code>count()</code>	This return the number of non-null values in a data object.
<code>mean(), median(), min(), max()</code>	This return mean, median, minimum, and maximum values of an axis in a data object.
<code>std(), var(), sem()</code>	These return the standard deviation, variance, and standard error of mean.
<code>abs()</code>	This gets the absolute value of a data object.

Function application

Pandas supports function application that allows us to apply some functions supported in other packages such as NumPy or our own functions on data structure objects. Here, we illustrate two examples of these cases, firstly, using `apply` to execute the `std()` function, which is the standard deviation calculating function of the NumPy package:

```
>>> df5.apply(np.std, axis=1)      # default: axis=0
0    0.816497
1    0.816497
2    0.816497
dtype: float64
```

Secondly, if we want to apply a formula to a data object, we can also use `apply` function by following these steps:

1. Define the function or formula that you want to apply on a data object.
2. Call the defined function or formula via `apply`. In this step, we also need to figure out the axis that we want to apply the calculation to:

```
>>> f = lambda x: x.max() - x.min()      # step 1
>>> df5.apply(f, axis=1)                  # step 2
0    2
1    2
2    2
dtype: int64
>>> def sigmoid(x):
    return 1/(1 + np.exp(x))
>>> df5.apply(sigmoid)
   a          b          c
0 0.500000  0.268941  0.119203
1 0.047426  0.017986  0.006693
2 0.002473  0.000911  0.000335
```

Sorting

There are two kinds of sorting method that we are interested in: sorting by row or column index and sorting by data value.

Firstly, we will consider methods for sorting by row and column index. In this case, we have the `sort_index()` function. We also have `axis` parameter to set whether the function should sort by row or column. The `ascending` option with the `true` or `false` value will allow us to sort data in ascending or descending order. The default setting for this option is `true`:

```
>>> df7 = pd.DataFrame(np.arange(12).reshape(3,4),
                      columns=['b', 'd', 'a', 'c'],
                      index=['x', 'y', 'z'])

>>> df7
   b  d  a  c
x  0  1  2  3
y  4  5  6  7
z  8  9 10 11
```

```
>>> df7.sort_index(axis=1)
      a   b   c   d
x    2   0   3   1
y    6   4   7   5
z   10   8  11   9
```

Series has a method `order` that sorts by value. For `NaN` values in the object, we can also have a special treatment via the `na_position` option:

```
>>> s4.order(na_position='first')
024      NaN
065      NaN
002      Mary
001      Nam
dtype: object
>>> s4
002      Mary
001      Nam
024      NaN
065      NaN
dtype: object
```

Besides that, Series also has the `sort()` function that sorts data by value. However, the function will not return a copy of the sorted data:

```
>>> s4.sort(na_position='first')
>>> s4
024      NaN
065      NaN
002      Mary
001      Nam
dtype: object
```

If we want to apply sort function to a DataFrame object, we need to figure out which columns or rows will be sorted:

```
>>> df7.sort(['b', 'd'], ascending=False)
      b   d   a   c
z  8   9  10  11
y  4   5   6   7
x  0   1   2   3
```

If we do not want to automatically save the sorting result to the current data object, we can change the setting of the `inplace` parameter to `False`.

Indexing and selecting data

In this section, we will focus on how to get, set, or slice subsets of Pandas data structure objects. As we learned in previous sections, Series or DataFrame objects have axis labeling information. This information can be used to identify items that we want to select or assign a new value to in the object:

```
>>> s4[['024', '002']]    # selecting data of Series object
024      NaN
002      Mary
dtype: object
>>> s4[['024', '002']] = 'unknown' # assigning data
>>> s4
024      unknown
065      NaN
002      unknown
001      Nam
dtype: object
```

If the data object is a DataFrame structure, we can also proceed in a similar way:

```
>>> df5[['b', 'c']]
      b   c
0   1   2
1   4   5
2   7   8
```

For label indexing on the rows of DataFrame, we use the `ix` function that enables us to select a set of rows and columns in the object. There are two parameters that we need to specify: the `row` and `column` labels that we want to get. By default, if we do not specify the selected column names, the function will return selected rows with all columns in the object:

```
>>> df5.ix[0]
a    0
b    1
c    2
Name: 0, dtype: int64
>>> df5.ix[0, 1:3]
b    1
c    2
Name: 0, dtype: int64
```

Moreover, we have many ways to select and edit data contained in a Pandas object. We summarize these functions in the following table:

Method	Description
<code>icol, irow</code>	This selects a single row or column by integer location.
<code>get_value, set_value</code>	This selects or sets a single value of a data object by row or column label.
<code>xs</code>	This selects a single column or row as a Series by label.

 Pandas data objects may contain duplicate indices. In this case, when we get or set a data value via index label, it will affect all rows or columns that have the same selected index name.

Computational tools

Let's start with correlation and covariance computation between two data objects. Both the Series and DataFrame have a `cov` method. On a DataFrame object, this method will compute the covariance between the Series inside the object:

```
>>> s1 = pd.Series(np.random.rand(3))
>>> s1
```

```
0    0.460324
1    0.993279
2    0.032957
dtype: float64
>>> s2 = pd.Series(np.random.rand(3))
>>> s2
0    0.777509
1    0.573716
2    0.664212
dtype: float64
>>> s1.cov(s2)
-0.024516360159045424

>>> df8 = pd.DataFrame(np.random.rand(12).reshape(4,3),
                      columns=['a','b','c'])
>>> df8
      a          b          c
0  0.200049  0.070034  0.978615
1  0.293063  0.609812  0.788773
2  0.853431  0.243656  0.978057
3  0.985584  0.500765  0.481180
>>> df8.cov()
      a          b          c
a  0.155307  0.021273 -0.048449
b  0.021273  0.059925 -0.040029
c -0.048449 -0.040029  0.055067
```

Usage of the correlation method is similar to the covariance method. It computes the correlation between Series inside a data object in case the data object is a DataFrame. However, we need to specify which method will be used to compute the correlations. The available methods are `pearson`, `kendall`, and `spearman`. By default, the function applies the `spearman` method:

```
>>> df8.corr(method = 'spearman')
      a          b          c
a  1.0  0.4 -0.8
b  0.4  1.0 -0.8
c -0.8 -0.8  1.0
```

We also have the `corrwith` function that supports calculating correlations between Series that have the same label contained in different DataFrame objects:

```
>>> df9 = pd.DataFrame(np.arange(8).reshape(4,2),
                      columns=['a', 'b'])

>>> df9
   a   b
0  0   1
1  2   3
2  4   5
3  6   7

>>> df8.corrwith(df9)
a    0.955567
b    0.488370
c      NaN
dtype: float64
```

Working with missing data

In this section, we will discuss missing, `NaN`, or `null` values, in Pandas data structures. It is a very common situation to arrive with missing data in an object. One such case that creates missing data is reindexing:

```
>>> df8 = pd.DataFrame(np.arange(12).reshape(4,3),
                      columns=['a', 'b', 'c'])

   a   b   c
0  0   1   2
1  3   4   5
2  6   7   8
3  9  10  11

>>> df9 = df8.reindex(columns = ['a', 'b', 'c', 'd'])
      a   b   c   d
0  0   1   2  NaN
1  3   4   5  NaN
2  6   7   8  NaN
3  9  10  11  NaN
```

```
>>> df10 = df8.reindex([3, 2, 'a', 0])
      a    b    c
3    9   10   11
2    6    7    8
a  NaN  NaN  NaN
0    0    1    2
```

To manipulate missing values, we can use the `isnull()` or `notnull()` functions to detect the missing values in a Series object, as well as in a DataFrame object:

```
>>> df10.isnull()
      a        b        c
3  False  False  False
2  False  False  False
a  True   True   True
0  False  False  False
```

On a Series, we can drop all null data and index values by using the `dropna` function:

```
>>> s4 = pd.Series({'001': 'Nam', '002': 'Mary',
                   '003': 'Peter'},
                   index=['002', '001', '024', '065'])

>>> s4
002    Mary
001    Nam
024    NaN
065    NaN
dtype: object

>>> s4.dropna()      # dropping all null value of Series object
002    Mary
001    Nam
dtype: object
```

With a DataFrame object, it is a little bit more complex than with Series. We can tell which rows or columns we want to drop and also if all entries must be null or a single null value is enough. By default, the function will drop any row containing a missing value:

```
>>> df9.dropna()      # all rows will be dropped
Empty DataFrame
Columns: [a, b, c, d]
Index: []
>>> df9.dropna(axis=1)
   a   b   c
0  0   1   2
1  3   4   5
2  6   7   8
3  9  10  11
```

Another way to control missing values is to use the supported parameters of functions that we introduced in the previous section. They are also very useful to solve this problem. In our experience, we should assign a fixed value in missing cases when we create data objects. This will make our objects cleaner in later processing steps. For example, consider the following:

```
>>> df11 = df8.reindex([3, 2, 'a', 0], fill_value = 0)
>>> df11
   a   b   c
3  9  10  11
2  6   7   8
a  0   0   0
0  0   1   2
```

We can also use the `fillna` function to fill a custom value in missing values:

```
>>> df9.fillna(-1)
   a   b   c   d
0  0   1   2  -1
1  3   4   5  -1
2  6   7   8  -1
3  9  10  11  -1
```

Advanced uses of Pandas for data analysis

In this section we will consider some advanced Pandas use cases.

Hierarchical indexing

Hierarchical indexing provides us with a way to work with higher dimensional data in a lower dimension by structuring the data object into multiple index levels on an axis:

```
>>> s8 = pd.Series(np.random.rand(8), index=[['a','a','b','b','c','c',
'd','d'], [0, 1, 0, 1, 0, 1, 0, 1]])
>>> s8
a    0    0.721652
     1    0.297784
b    0    0.271995
     1    0.125342
c    0    0.444074
     1    0.948363
d    0    0.197565
     1    0.883776
dtype: float64
```

In the preceding example, we have a Series object that has two index levels. The object can be rearranged into a DataFrame using the `unstack` function. In an inverse situation, the `stack` function can be used:

```
>>> s8.unstack()
          0            1
a  0.549211  0.420874
b  0.051516  0.715021
c  0.503072  0.720772
d  0.373037  0.207026
```

We can also create a DataFrame to have a hierarchical index in both axes:

```
>>> df = pd.DataFrame(np.random.rand(12).reshape(4,3),
                     index=[['a', 'a', 'b', 'b'],
                            [0, 1, 0, 1]],
                     columns[['x', 'x', 'y'], [0, 1, 0]])

>>> df
          x           y
      0       1       0
a 0  0.636893  0.729521  0.747230
  1  0.749002  0.323388  0.259496
b 0  0.214046  0.926961  0.679686
  0.013258  0.416101  0.626927

>>> df.index
MultiIndex(levels=[['a', 'b'], [0, 1]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

>>> df.columns
MultiIndex(levels=[['x', 'y'], [0, 1]],
           labels=[[0, 0, 1], [0, 1, 0]])
```

The methods for getting or setting values or subsets of the data objects with multiple index levels are similar to those of the nonhierarchical case:

```
>>> df['x']
          0       1
a 0  0.636893  0.729521
  1  0.749002  0.323388
b 0  0.214046  0.926961
  0.013258  0.416101

>>> df[[0]]
          x
      0
a 0  0.636893
  1  0.749002
b 0  0.214046
  0.013258
```

```
>>> df.ix['a', 'x']
      0      1
0  0.636893  0.729521
0.749002  0.323388
>>> df.ix['a','x'].ix[1]
0    0.749002
1    0.323388
Name: 1, dtype: float64
```

After grouping data into multiple index levels, we can also use most of the descriptive and statistics functions that have a level option, which can be used to specify the level we want to process:

```
>>> df.std(level=1)
      x          y
      0      1      0
0  0.298998  0.139611  0.047761
0.520250  0.065558  0.259813
>>> df.std(level=0)
      x          y
      0      1      0
a  0.079273  0.287180  0.344880
b  0.141979  0.361232  0.037306
```

The Panel data

The Panel is another data structure for three-dimensional data in Pandas. However, it is less frequently used than the Series or the DataFrame. You can think of a Panel as a table of DataFrame objects. We can create a Panel object from a 3D ndarray or a dictionary of DataFrame objects:

```
# create a Panel from 3D ndarray
>>> panel = pd.Panel(np.random.rand(2, 4, 5),
                     items = ['item1', 'item2'])

>>> panel
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 5 (minor_axis)
Items axis: item1 to item2
Major_axis axis: 0 to 3
```

```
Minor_axis axis: 0 to 4

>>> df1 = pd.DataFrame(np.arange(12).reshape(4, 3),
                      columns=['a','b','c'])

>>> df1
   a   b   c
0  0   1   2
1  3   4   5
2  6   7   8
3  9  10  11

>>> df2 = pd.DataFrame(np.arange(9).reshape(3, 3),
                      columns=['a','b','c'])

>>> df2
   a   b   c
0  0   1   2
1  3   4   5
2  6   7   8

# create another Panel from a dict of DataFrame objects
>>> panel2 = pd.Panel({'item1': df1, 'item2': df2})

>>> panel2
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 3 (minor_axis)
Items axis: item1 to item2
Major_axis axis: 0 to 3
Minor_axis axis: a to c
```

Each item in a Panel is a DataFrame. We can select an item, by item name:

```
>>> panel2['item1']
   a   b   c
0  0   1   2
1  3   4   5
2  6   7   8
3  9  10  11
```

Alternatively, if we want to select data via an axis or data position, we can use the `ix` method, like on Series or DataFrame:

```
>>> panel2.ix[:, 1:3, ['b', 'c']]
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: item1 to item2
Major_axis axis: 1 to 3
Minor_axis axis: b to c
>>> panel2.ix[:, 2, :]
   item1  item2
a      6      6
b      7      7
c      8      8
```

Summary

We have finished covering the basics of the Pandas data analysis library. Whenever you learn about a library for data analysis, you need to consider the three parts that we explained in this chapter. Data structures: we have two common data object types in the Pandas library; Series and DataFrames. Method to access and manipulate data objects: Pandas supports many way to select, set or slice subsets of data object. However, the general mechanism is using index labels or the positions of items to identify values. Functions and utilities: They are the most important part of a powerful library. In this chapter, we covered all common supported functions of Pandas which allow us compute statistics on data easily. The library also has a lot of other useful functions and utilities that we could not explain in this chapter. We encourage you to start your own research, if you want to expand your experience with Pandas. It helps us to process large data in an optimized way. You will see more of Pandas in action later in this book.

Until now, we learned about two popular Python libraries: NumPy and Pandas. Pandas is built on NumPy, and as a result it allows for a bit more convenient interaction with data. However, in some situations, we can flexibly combine both of them to accomplish our goals.

Practice exercises

The link https://www.census.gov/2010census/csv/pop_change.csv contains an US census dataset. It has 23 columns and one row for each US state, as well as a few rows for macro regions such as North, South, and West.

- Get this dataset into a Pandas DataFrame. Hint: just skip those rows that do not seem helpful, such as comments or description.
- While the dataset contains change metrics for each decade, we are interested in the population change during the second half of the twentieth century, that is between, 1950 and 2000. Which region has seen the biggest and the smallest population growth in this time span? Also, which US state?

Advanced open-ended exercise:

- Find more census data on the internet; not just on the US but on the world's countries. Try to find GDP data for the same time as well. Try to align this data to explore patterns. How are GDP and population growth related? Are there any special cases, such as countries with high GDP but low population growth or countries with the opposite history?

4

Data Visualization

Data visualization is concerned with the presentation of data in a pictorial or graphical form. It is one of the most important tasks in data analysis, since it enables us to see analytical results, detect outliers, and make decisions for model building. There are many Python libraries for visualization, of which matplotlib, seaborn, bokeh, and ggplot are among the most popular. However, in this chapter, we mainly focus on the matplotlib library that is used by many people in many different contexts.

Matplotlib produces publication-quality figures in a variety of formats, and interactive environments across Python platforms. Another advantage is that Pandas comes equipped with useful wrappers around several matplotlib plotting routines, allowing for quick and handy plotting of Series and DataFrame objects.

The IPython package started as an alternative to the standard interactive Python shell, but has since evolved into an indispensable tool for data exploration, visualization, and rapid prototyping. It is possible to use the graphical capabilities offered by matplotlib from IPython through various options, of which the simplest to get started with is the `pylab` flag:

```
$ ipython --pylab
```

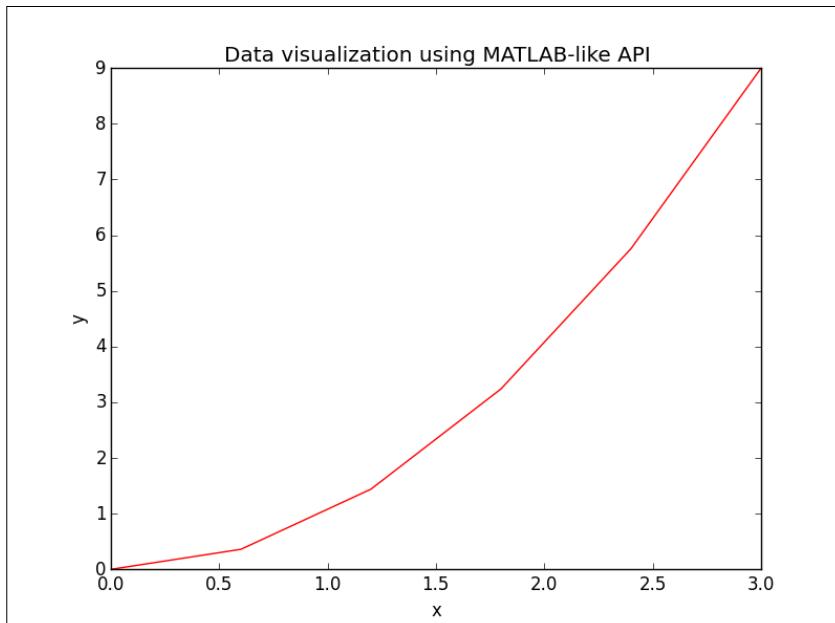
This flag will preload `matplotlib` and `numpy` for interactive use with the default `matplotlib` backend. IPython can run in various environments: in a terminal, as a `Qt` application, or inside a browser. These options are worth exploring, since IPython has enjoyed adoption for many use cases, such as prototyping, interactive slides for more engaging conference talks or lectures, and as a tool for sharing research.

The matplotlib API primer

The easiest way to get started with plotting using matplotlib is often by using the MATLAB API that is supported by the package:

```
>>> import matplotlib.pyplot as plt
>>> from numpy import *
>>> x = linspace(0, 3, 6)
>>> x
array([0., 0.6, 1.2, 1.8, 2.4, 3.])
>>> y = power(x, 2)
>>> y
array([0., 0.36, 1.44, 3.24, 5.76, 9.])
>>> figure()
>>> plot(x, y, 'r')
>>> xlabel('x')
>>> ylabel('y')
>>> title('Data visualization in MATLAB-like API')
>>> plt.show()
```

The output for the preceding command is as follows:



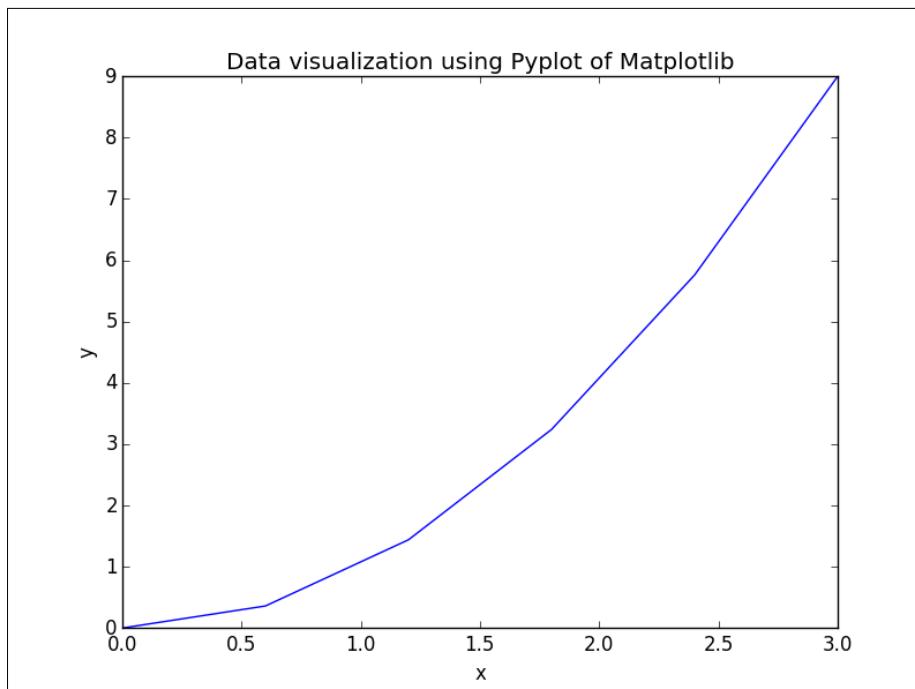
However, star imports should not be used unless there is a good reason for doing so. In the case of matplotlib, we can use the canonical import:

```
>>> import matplotlib.pyplot as plt
```

The preceding example could then be written as follows:

```
>>> plt.plot(x, y)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.title('Data visualization using Pyplot of Matplotlib')
>>> plt.show()
```

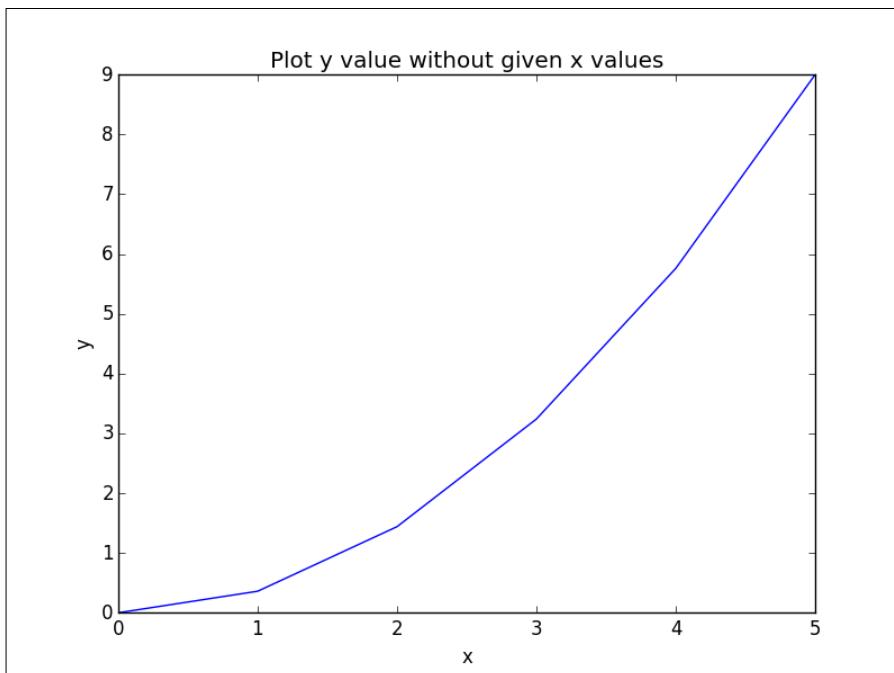
The output for the preceding command is as follows:



If we only provide a single argument to the `plot` function, it will automatically use it as the `y` values and generate the `x` values from 0 to `N-1`, where `N` is equal to the number of values:

```
>>> plt.plot(y)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.title('Plot y value without given x values')
>>> plt.show()
```

The output for the preceding command is as follows:



By default, the range of the axes is constrained by the range of the input `x` and `y` data. If we want to specify the `viewport` of the axes, we can use the `axis()` method to set custom ranges. For example, in the previous visualization, we could increase the range of the `x` axis from [0, 5] to [0, 6], and that of the `y` axis from [0, 9] to [0, 10], by writing the following command:

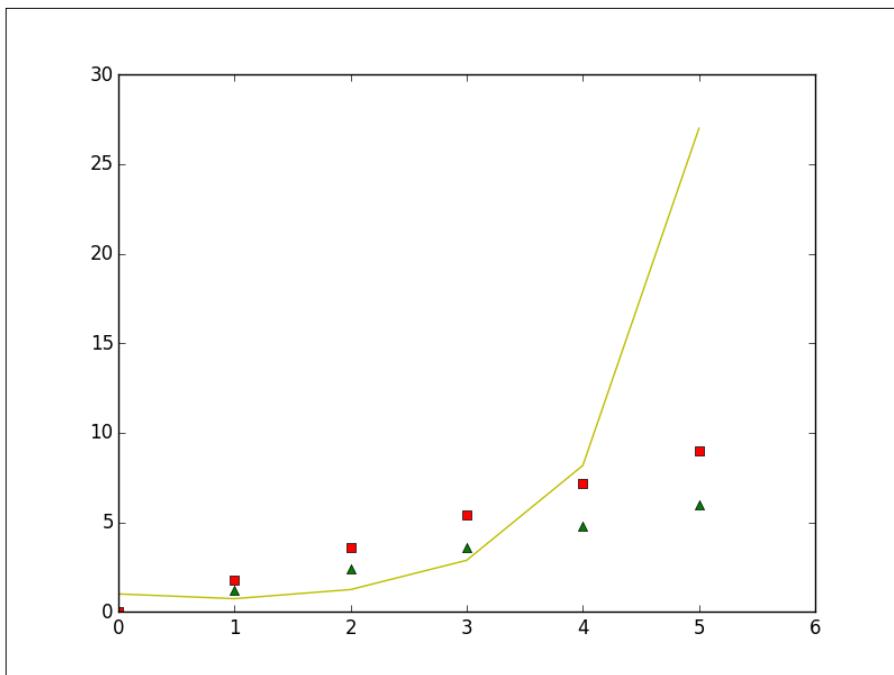
```
>>> plt.axis([0, 6, 0, 12])
```

Line properties

The default line format when we plot data in matplotlib is a solid blue line, which is abbreviated as `b-`. To change this setting, we only need to add the symbol code, which includes letters as color string and symbols as line style string, to the `plot` function. Let us consider a plot of several lines with different format styles:

```
>>> plt.plot(x*2, 'g^', x*3, 'rs', x**x, 'y-')
>>> plt.axis([0, 6, 0, 30])
>>> plt.show()
```

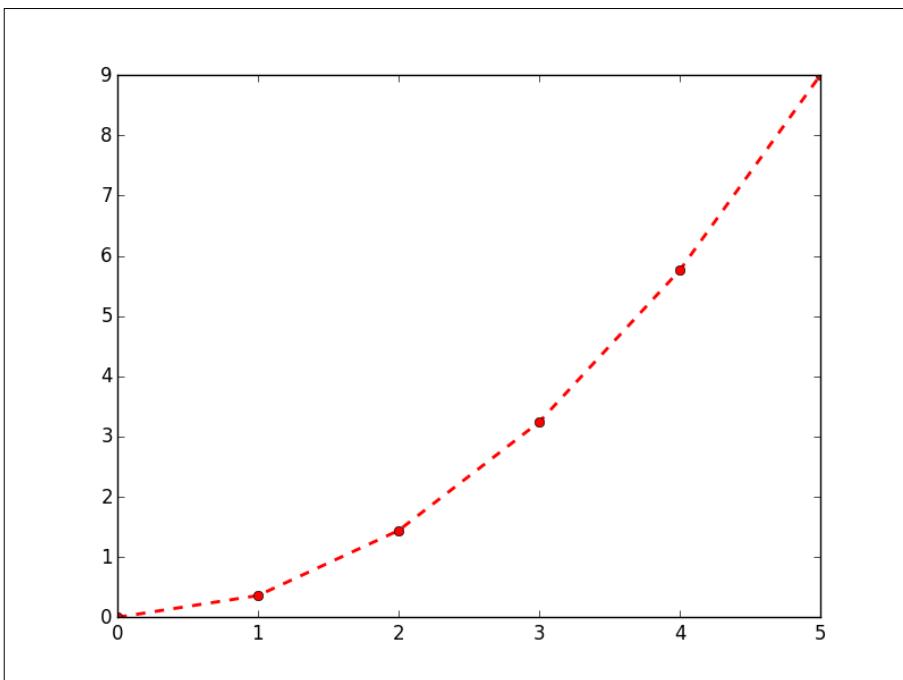
The output for the preceding command is as follows:



There are many line styles and attributes, such as color, line width, and dash style, that we can choose from to control the appearance of our plots. The following example illustrates several ways to set line properties:

```
>>> line = plt.plot(y, color='red', linewidth=2.0)
>>> line.set_linestyle('---')
>>> plt.setp(line, marker='o')
>>> plt.show()
```

The output for the preceding command is as follows:



The following table lists some common properties of the `line2d` plotting:

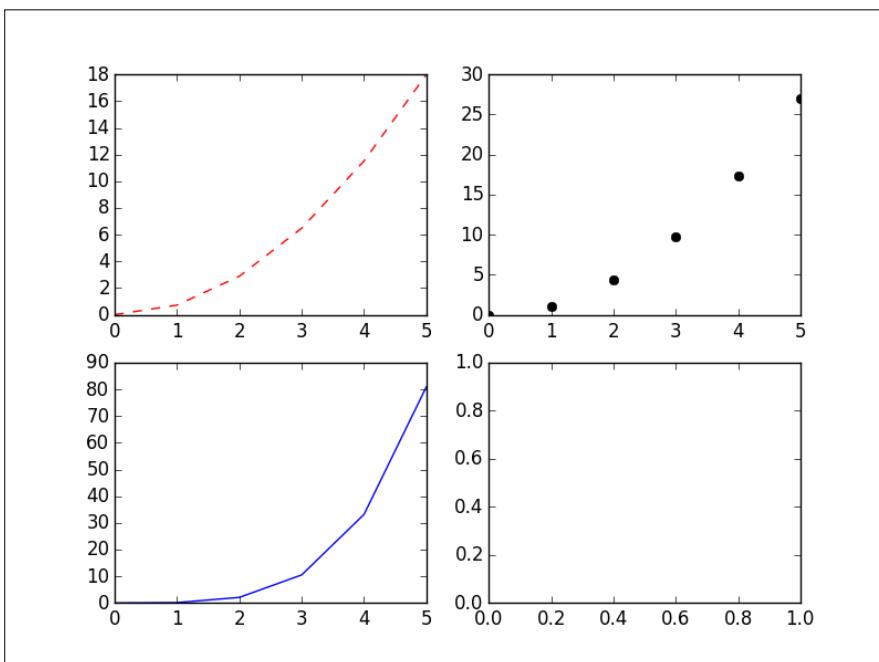
Property	Value type	Description
<code>color or c</code>	Any matplotlib color	This sets the color of the line in the figure
<code>dashes</code>	On/off	This sets the sequence of ink in the points
<code>data</code>	<code>np.array xdata,</code> <code>np.array ydata</code>	This sets the data used for visualization
<code>linestyle or ls</code>	<code>['-' '--' '-.' ':' ...]</code>	This sets the line style in the figure
<code>linewidth or lw</code>	Float value in points	This sets the width of line in the figure
<code>marker</code>	Any symbol	This sets the style at data points in the figure

Figures and subplots

By default, all plotting commands apply to the current figure and axes. In some situations, we want to visualize data in multiple figures and axes to compare different plots or to use the space on a page more efficiently. There are two steps required before we can plot the data. Firstly, we have to define which figure we want to plot. Secondly, we need to figure out the position of our subplot in the figure:

```
>>> plt.figure('a')      # define a figure, named 'a'  
>>> plt.subplot(221)      # the first position of 4 subplots in 2x2 figure  
>>> plt.plot(y+y, 'r--')  
>>> plt.subplot(222)      # the second position of 4 subplots  
>>> plt.plot(y*y, 'ko')  
>>> plt.subplot(223)      # the third position of 4 subplots  
>>> plt.plot(y*y, 'b^')  
>>> plt.subplot(224)  
>>> plt.show()
```

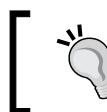
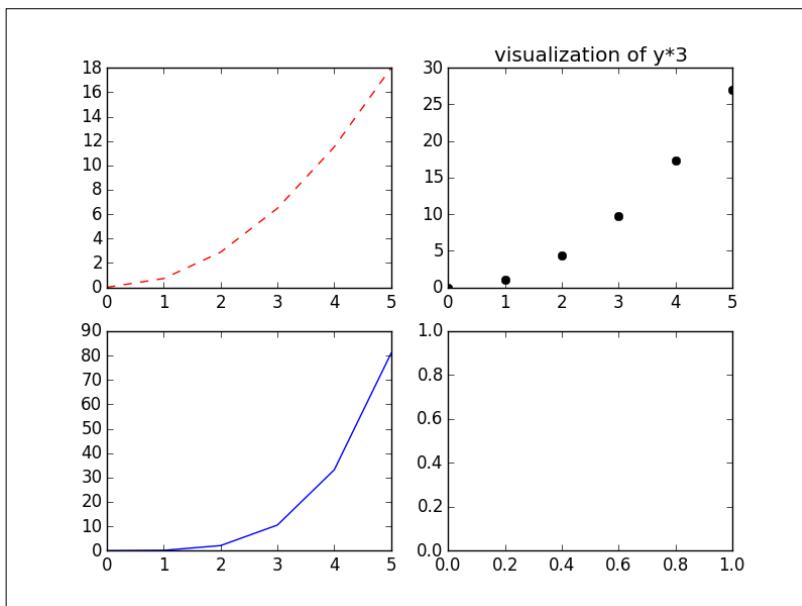
The output for the preceding command is as follows:



In this case, we currently have the figure a. If we want to modify any subplot in figure a, we first call the command to select the figure and subplot, and then execute the function to modify the subplot. Here, for example, we change the title of the second plot of our four-plot figure:

```
>>> plt.figure('a')
>>> plt.subplot(222)
>>> plt.title('visualization of y*3')
>>> plt.show()
```

The output for the preceding command is as follows:



Integer subplot specification must be a three-digit number if we are not using commas to separate indices. So, `plt.subplot(221)` is equal to the `plt.subplot(2, 2, 1)` command.

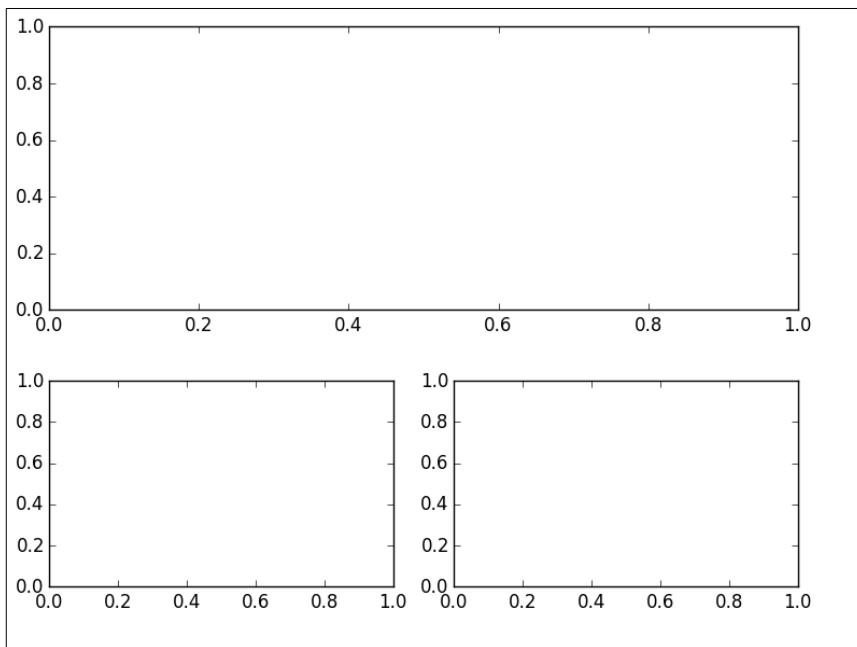


There is a convenience method, `plt.subplots()`, to creating a figure that contains a given number of subplots. As in the previous example, we can use the `plt.subplots(2, 2)` command to create a 2×2 figure that consists of four subplots.

We can also create the axes manually, instead of rectangular grid, by using the `plt.axes([left, bottom, width, height])` command, where all input parameters are in the fractional [0, 1] coordinates:

```
>>> plt.figure('b')      # create another figure, named 'b'  
>>> ax1 = plt.axes([0.05, 0.1, 0.4, 0.32])  
>>> ax2 = plt.axes([0.52, 0.1, 0.4, 0.32])  
>>> ax3 = plt.axes([0.05, 0.53, 0.87, 0.44])  
>>> plt.show()
```

The output for the preceding command is as follows:



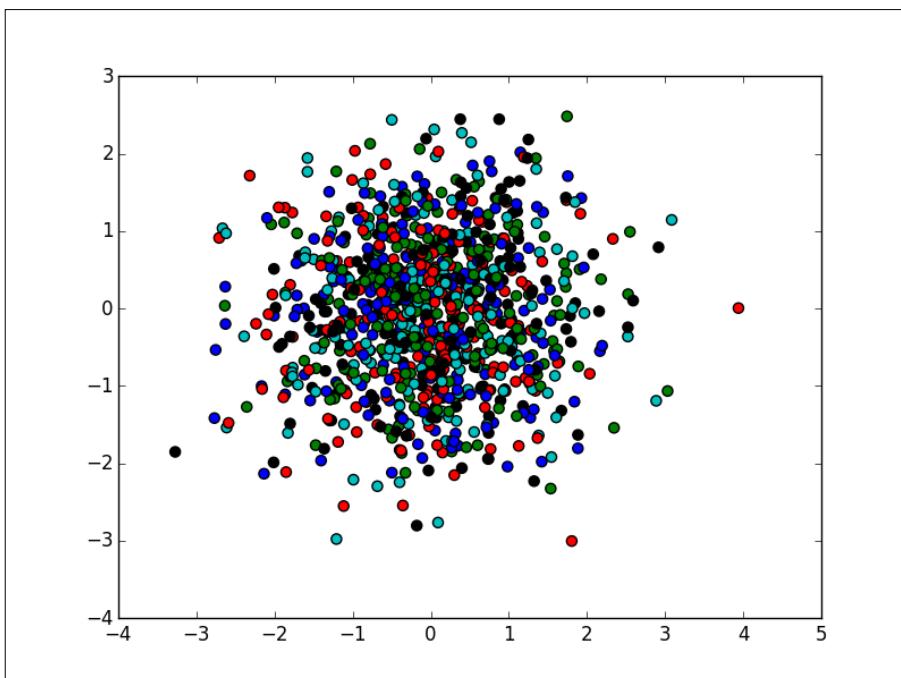
However, when you manually create axes, it takes more time to balance coordinates and sizes between subplots to arrive at a well-proportioned figure.

Exploring plot types

We have looked at how to create simple line plots so far. The matplotlib library supports many more plot types that are useful for data visualization. However, our goal is to provide the basic knowledge that will help you to understand and use the library for visualizing data in the most common situations. Therefore, we will only focus on four kinds of plot types: **scatter plots**, **bar plots**, **contour plots**, and **histograms**.

Scatter plots

A scatter plot is used to visualize the relationship between variables measured in the same dataset. It is easy to plot a simple scatter plot, using the `plt.scatter()` function, that requires numeric columns for both the x and y axis:



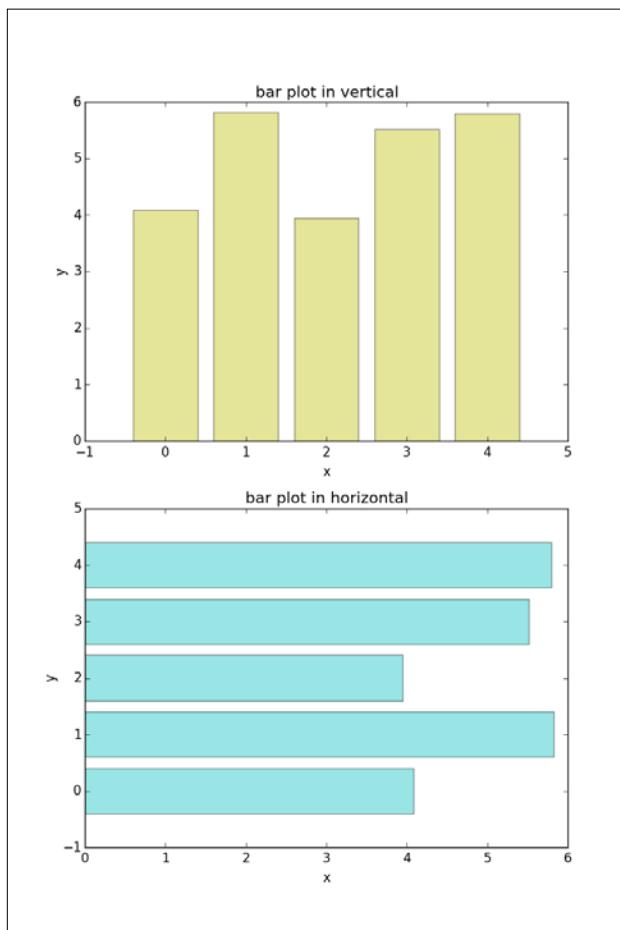
Let's take a look at the command for the preceding output:

```
>>> X = np.random.normal(0, 1, 1000)
>>> Y = np.random.normal(0, 1, 1000)
>>> plt.scatter(X, Y, c = ['b', 'g', 'k', 'r', 'c'])
>>> plt.show()
```

Bar plots

A bar plot is used to present grouped data with rectangular bars, which can be either vertical or horizontal, with the lengths of the bars corresponding to their values.

We use the `plt.bar()` command to visualize a vertical bar, and the `plt.bardh()` command for the other:



The command for the preceding output is as follows:

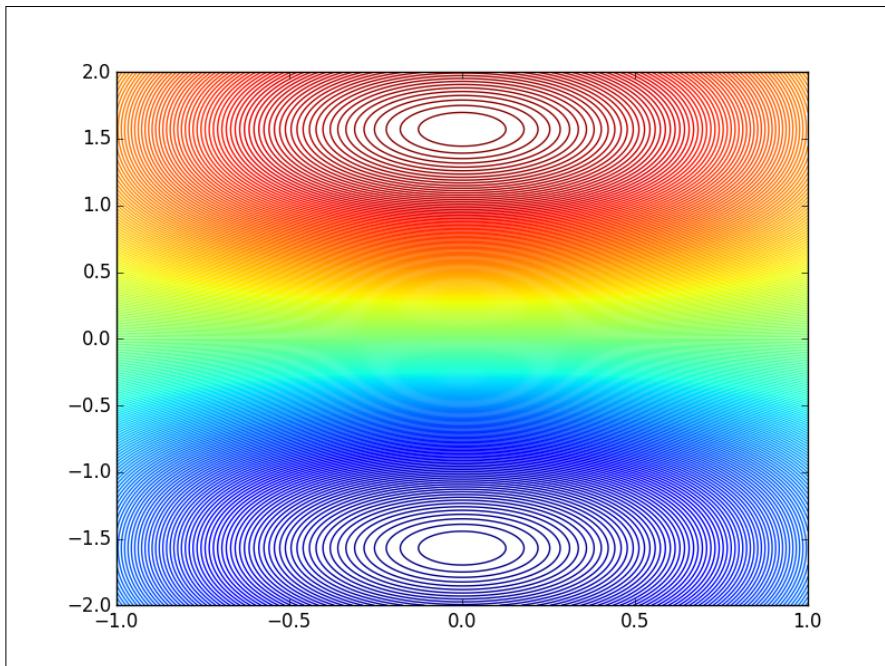
```
>>> X = np.arange(5)
>>> Y = 3.14 + 2.71 * np.random.rand(5)
>>> plt.subplots(2)
>>> # the first subplot
>>> plt.subplot(211)
>>> plt.bar(X, Y, align='center', alpha=0.4, color='y')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.title('bar plot in vertical')
>>> # the second subplot
>>> plt.subplot(212)
>>> plt.bart(X, Y, align='center', alpha=0.4, color='c')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.title('bar plot in horizontal')
>>> plt.show()
```

Contour plots

We use contour plots to present the relationship between three numeric variables in two dimensions. Two variables are drawn along the x and y axes, and the third variable, z, is used for contour levels that are plotted as curves in different colors:

```
>>> x = np.linspace(-1, 1, 255)
>>> y = np.linspace(-2, 2, 300)
>>> z = np.sin(y[:, np.newaxis]) * np.cos(x)
>>> plt.contour(x, y, z, 255, linewidth=2)
>>> plt.show()
```

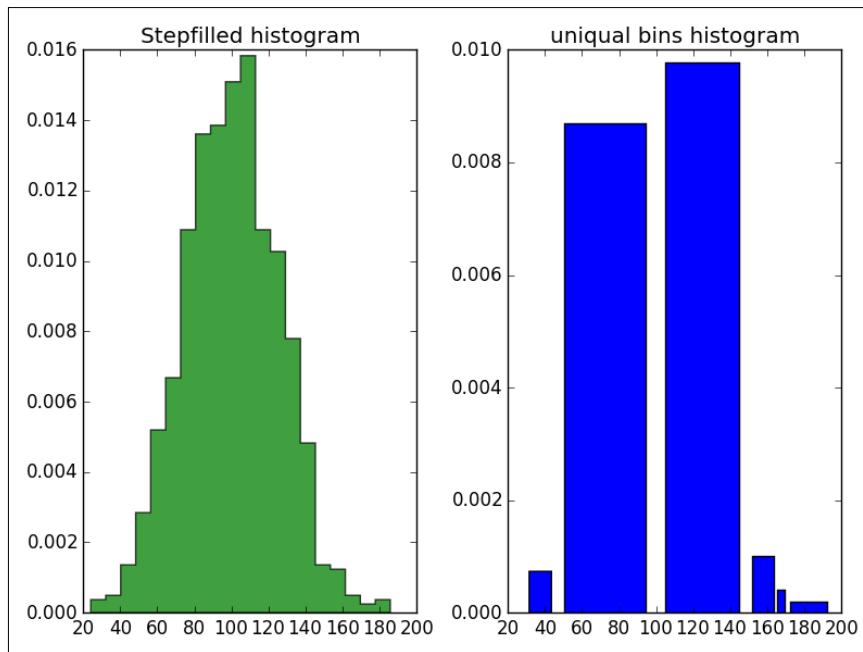
Let's take a look at the contour plot in the following image:



If we want to draw contour lines and filled contours, we can use the `plt.contourf()` method instead of `plt.contour()`. In contrast to MATLAB, matplotlib's `contourf()` will not draw the polygon edges.

Histogram plots

A histogram represents the distribution of numerical data graphically. Usually, the range of values is partitioned into bins of equal size, with the height of each bin corresponding to the frequency of values within that bin:



The command for the preceding output is as follows:

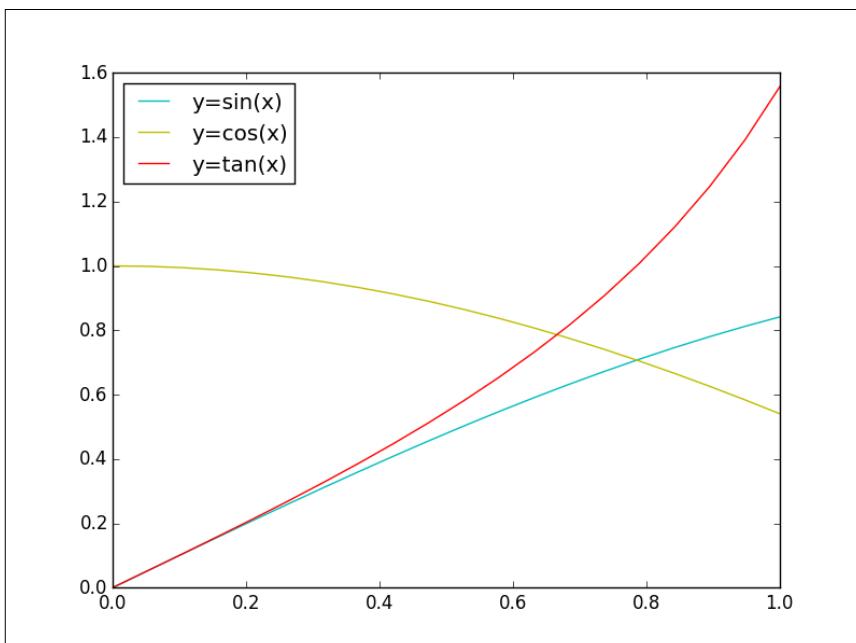
```
>>> mu, sigma = 100, 25
>>> fig, (ax0, ax1) = plt.subplots(ncols=2)
>>> x = mu + sigma * np.random.randn(1000)
>>> ax0.hist(x, 20, normed=1, histtype='stepfilled',
            facecolor='g', alpha=0.75)
>>> ax0.set_title('Stepfilled histogram')
>>> ax1.hist(x, bins=[100,150, 165, 170, 195] normed=1,
            histtype='bar', rwidth=0.8)
>>> ax1.set_title('unique bins histogram')
>>> # automatically adjust subplot parameters to give specified padding
>>> plt.tight_layout()
>>> plt.show()
```

Legends and annotations

Legends are an important element that is used to identify the plot elements in a figure. The easiest way to show a legend inside a figure is to use the `label` argument of the `plot` function, and show the labels by calling the `plt.legend()` method:

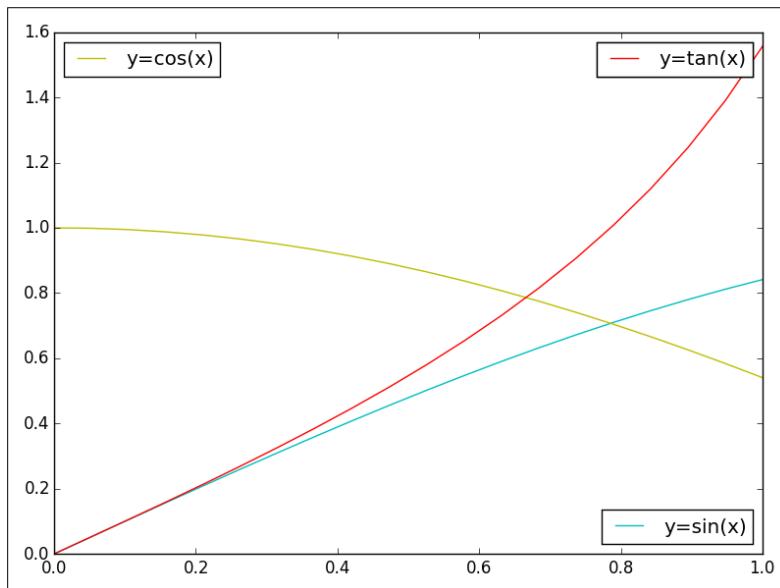
```
>>> x = np.linspace(0, 1, 20)
>>> y1 = np.sin(x)
>>> y2 = np.cos(x)
>>> y3 = np.tan(x)
>>> plt.plot(x, y1, 'c', label='y=sin(x)')
>>> plt.plot(x, y2, 'y', label='y=cos(x)')
>>> plt.plot(x, y3, 'r', label='y=tan(x)')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

The output for the preceding command as follows:



The `loc` argument in the legend command is used to figure out the position of the label box. There are several valid location options: `lower left`, `right`, `upper left`, `lower center`, `upper right`, `center`, `lower right`, `upper right`, `center right`, `best`, `upper center`, and `center left`. The default position setting is `upper right`. However, when we set an invalid location option that does not exist in the above list, the function automatically falls back to the best option.

If we want to split the legend into multiple boxes in a figure, we can manually set our expected labels for plot lines, as shown in the following image:



The output for the preceding command is as follows:

```
>>> p1 = plt.plot(x, y1, 'c', label='y=sin(x)')
>>> p2 = plt.plot(x, y2, 'y', label='y=cos(x)')
>>> p3 = plt.plot(x, y3, 'r', label='y=tan(x)')
>>> lsin = plt.legend(handles=p1, loc='lower right')
>>> lcos = plt.legend(handles=p2, loc='upper left')
>>> ltan = plt.legend(handles=p3, loc='upper right')
>>> # with above code, only 'y=tan(x)' legend appears in the figure
>>> # fix: add lsin, lcos as separate artists to the axes
```

```
>>> plt.gca().add_artist(lsin)
>>> plt.gca().add_artist(lcos)
>>> # automatically adjust subplot parameters to specified padding
>>> plt.tight_layout()
>>> plt.show()
```

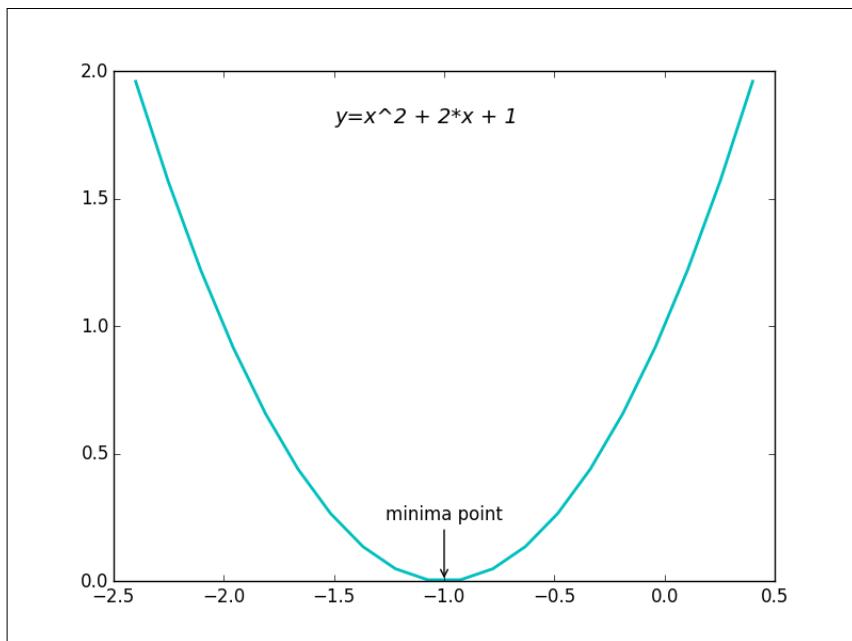
The other element in a figure that we want to introduce is the annotations which can consist of text, arrows, or other shapes to explain parts of the figure in detail, or to emphasize some special data points. There are different methods for showing annotations, such as `text`, `arrow`, and `annotation`.

- The `text` method draws text at the given coordinates (`x`, `y`) on the plot; optionally with custom properties. There are some common arguments in the function: `x`, `y`, label text, and font-related properties that can be passed in via `fontdict`, such as `family`, `fontsize`, and `style`.
- The `annotate` method can draw both text and arrows arranged appropriately. Arguments of this function are `s` (label text), `xy` (the position of element to annotation), `xytext` (the position of the label `s`), `xycoords` (the string that indicates what type of coordinate `xy` is), and `arrowprops` (the dictionary of line properties for the arrow that connects the annotation).

Here is a simple example to illustrate the `annotate` and `text` functions:

```
>>> x = np.linspace(-2.4, 0.4, 20)
>>> y = x*x + 2*x + 1
>>> plt.plot(x, y, 'c', linewidth=2.0)
>>> plt.text(-1.5, 1.8, 'y=x^2 + 2*x + 1',
            fontsize=14, style='italic')
>>> plt.annotate('minima point', xy=(-1, 0),
                xytext=(-1, 0.3),
                horizontalalignment='center',
                verticalalignment='top',
                arrowprops=dict(arrowstyle='->',
                               connectionstyle='arc3'))
>>> plt.show()
```

The output for the preceding command is as follows:



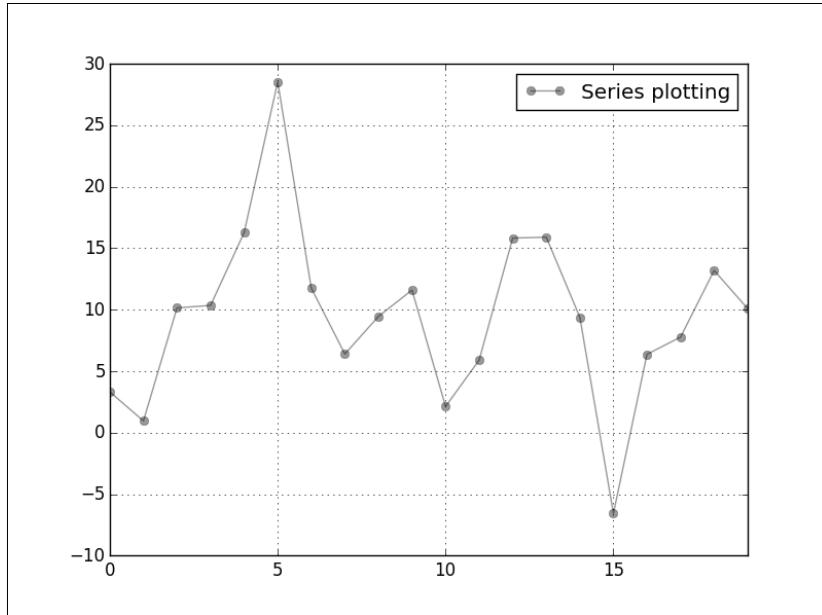
Plotting functions with Pandas

We have covered most of the important components in a plot figure using matplotlib. In this section, we will introduce another powerful plotting method for directly creating standard visualization from Pandas data objects that are often used to manipulate data.

For Series or DataFrame objects in Pandas, most plotting types are supported, such as line, bar, box, histogram, and scatter plots, and pie charts. To select a plot type, we use the `kind` argument of the `plot` function. With no kind of plot specified, the `plot` function will generate a line style visualization by default , as in the following example:

```
>>> s = pd.Series(np.random.normal(10, 8, 20))
>>> s.plot(style='ko-', alpha=0.4, label='Series plotting')
>>> plt.legend()
>>> plt.show()
```

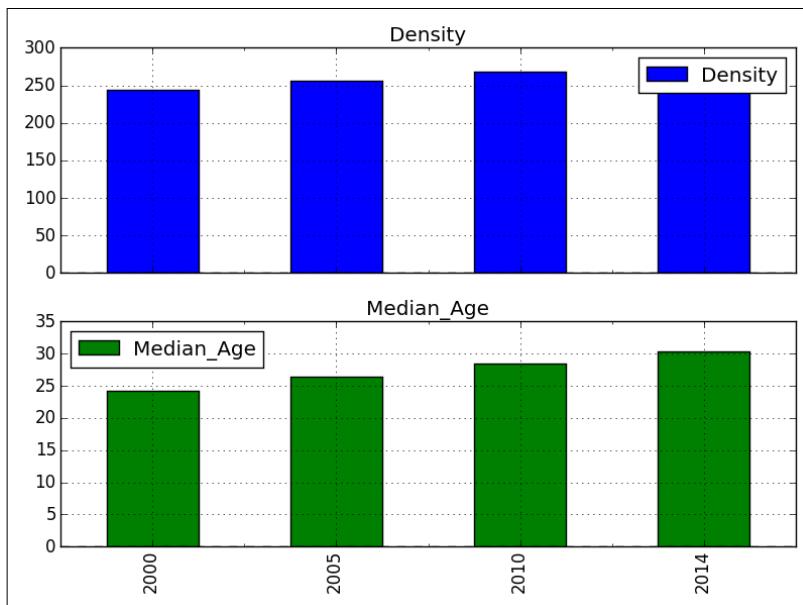
The output for the preceding command is as follows:



Another example will visualize the data of a DataFrame object consisting of multiple columns:

```
>>> data = {'Median_Age': [24.2, 26.4, 28.5, 30.3],  
           'Density': [244, 256, 268, 279]}  
>>> index_label = ['2000', '2005', '2010', '2014'];  
>>> df1 = pd.DataFrame(data, index=index_label)  
>>> df1.plot(kind='bar', subplots=True, sharex=True)  
>>> plt.tight_layout();  
>>> plt.show()
```

The output for the preceding command is as follows:



The plot method of the DataFrame has a number of options that allow us to handle the plotting of the columns. For example, in the above DataFrame visualization, we chose to plot the columns in separate subplots. The following table lists more options:

Argument	Value	Description
subplots	True/False	The plots each data column in a separate subplot
logy	True/False	The gets a log-scale y axis
secondary_y	True/False	The plots data on a secondary y axis
sharex, sharey	True/False	The shares the same x or y axis, linking sticks and limits

Additional Python data visualization tools

Besides matplotlib, there are other powerful data visualization toolkits based on Python. While we cannot dive deeper into these libraries, we would like to at least briefly introduce them in this session.

Bokeh

Bokeh is a project by Peter Wang, Hugo Shi, and others at Continuum Analytics. It aims to provide elegant and engaging visualizations in the style of `D3.js`. The library can quickly and easily create interactive plots, dashboards, and data applications. Here are a few differences between matplotlib and Bokeh:

- Bokeh achieves cross-platform ubiquity through IPython's new model of in-browser client-side rendering
- Bokeh uses a syntax familiar to R and ggplot users, while matplotlib is more familiar to Matlab users
- Bokeh has a coherent vision to build a ggplot-inspired in-browser interactive visualization tool, while Matplotlib has a coherent vision of focusing on 2D cross-platform graphics.

The basic steps for creating plots with Bokeh are as follows:

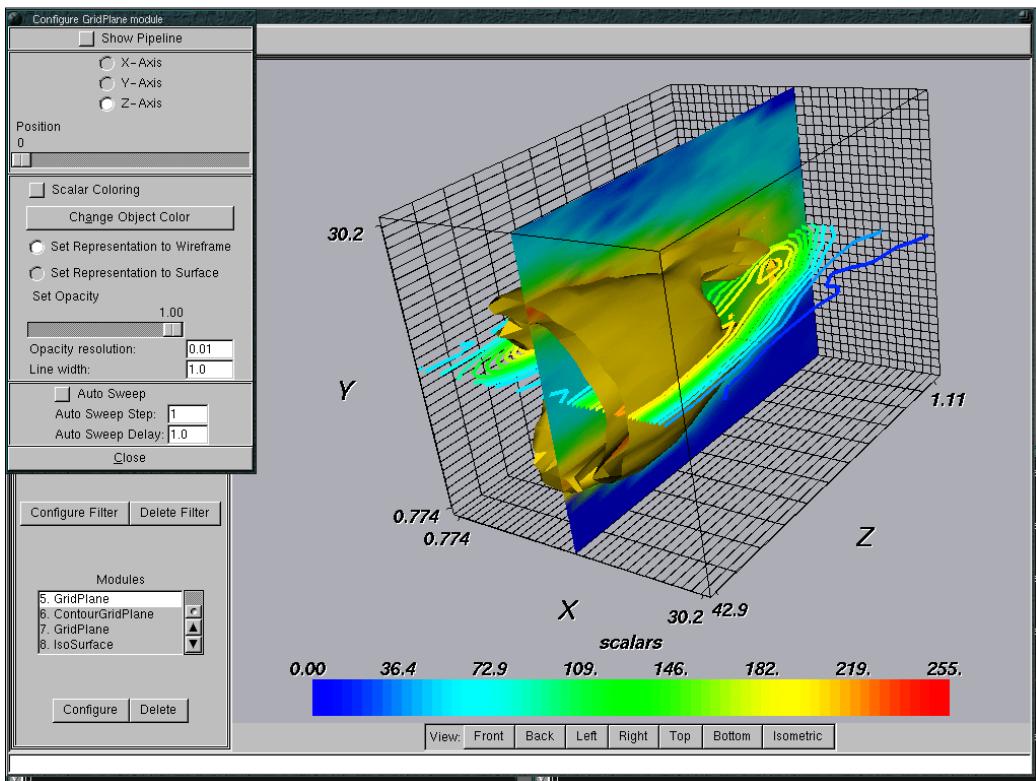
- Prepare some data in a list, series, and Dataframe
- Tell Bokeh where you want to generate the output
- Call `figure()` to create a plot with some overall options, similar to the matplotlib options discussed earlier
- Add renderers for your data, with visual customizations such as colors, legends, and width
- Ask Bokeh to `show()` or `save()` the results

MayaVi

MayaVi is a library for interactive scientific data visualization and 3D plotting, built on top of the award-winning **visualization toolkit (VTK)**, which is a traits-based wrapper for the open-source visualization library. It offers the following:

- The possibility to interact with the data and object in the visualization through dialogs.
- An interface in Python for scripting. MayaVi can work with Numpy and scipy for 3D plotting out of the box and can be used within IPython notebooks, which is similar to matplotlib.
- An abstraction over VTK that offers a simpler programming model.

Let's view an illustration made entirely using MayaVi based on VTK examples and their provided data:



Summary

We finished covering most of the basics, such as functions, arguments, and properties for data visualization, based on the matplotlib library. We hope that, through the examples, you will be able to understand and apply them to your own problems. In general, to visualize data, we need to consider five steps- that is, getting data into suitable Python or Pandas data structures, such as lists, dictionaries, Series, or DataFrames. We explained in the previous chapters, how to accomplish this step. The second step is defining plots and subplots for the data object in question. We discussed this in the figures and subplots session. The third step is selecting a plot style and its attributes to show in the subplots such as: `line`, `bar`, `histogram`, `scatter plot`, `line style`, and `color`. The fourth step is adding extra components to the subplots, like legends, annotations and text. The fifth step is displaying or saving the results.

By now, you can do quite a few things with a dataset; for example, manipulation, cleaning, exploration, and visualization based on Python libraries such as Numpy, Pandas, and matplotlib. You can now combine this knowledge and practice with these libraries to get more and more familiar with Python data analysis.

Practice exercises:

- Name two real or fictional datasets and explain which kind of plot would best fit the data: line plots, bar charts, scatter plots, contour plots, or histograms. Name one or two applications, where each of the plot type is common (for example, histograms are often used in image editing applications).
- We only focused on the most common plot types of matplotlib. After a bit of research, can you name a few more plot types that are available in matplotlib?
- Take one Pandas data structure from *Chapter 3, Data Analysis with Pandas* and plot the data in a suitable way. Then, save it as a PNG image to the disk.

5

Time Series

Time series typically consist of a sequence of data points coming from measurements taken over time. This kind of data is very common and occurs in a multitude of fields.

A business executive is interested in stock prices, prices of goods and services or monthly sales figures. A meteorologist takes temperature measurements several times a day and also keeps records of precipitation, humidity, wind direction and force. A neurologist can use electroencephalography to measure electrical activity of the brain along the scalp. A sociologist can use campaign contribution data to learn about political parties and their supporters and use these insights as an argumentation aid. More examples for time series data can be enumerated almost endlessly.

Time series primer

In general, time series serve two purposes. First, they help us to learn about the underlying process that generated the data. On the other hand, we would like to be able to forecast future values of the same or related series using existing data. When we measure temperature, precipitation or wind, we would like to learn more about more complex things, such as weather or the climate of a region and how various factors interact. At the same time, we might be interested in weather forecasting.

In this chapter we will explore the time series capabilities of Pandas. Apart from its powerful core data structures – the series and the DataFrame – Pandas comes with helper functions for dealing with time related data. With its extensive built-in optimizations, Pandas is capable of handling large time series with millions of data points with ease.

We will gradually approach time series, starting with the basic building blocks of date and time objects.

Working with date and time objects

Python supports date and time handling in the `date time` and `time` modules from the standard library:

```
>>> import datetime  
>>> datetime.datetime(2000, 1, 1)  
datetime.datetime(2000, 1, 1, 0, 0)
```

Sometimes, dates are given or expected as strings, so a conversion from or to strings is necessary, which is realized by two functions: `strptime` and `strftime`, respectively:

```
>>> datetime.datetime.strptime("2000/1/1", "%Y/%m/%d")  
datetime.datetime(2000, 1, 1, 0, 0)  
>>> datetime.datetime(2000, 1, 1, 0, 0).strftime("%Y%m%d")  
'20000101'
```

Real-world data usually comes in all kinds of shapes and it would be great if we did not need to remember the exact date format specifies for parsing. Thankfully, Pandas abstracts away a lot of the friction, when dealing with strings representing dates or time. One of these helper functions is `to_datetime`:

```
>>> import pandas as pd  
>>> import numpy as np  
>>> pd.to_datetime("4th of July")  
Timestamp('2015-07-04')  
>>> pd.to_datetime("13.01.2000")  
Timestamp('2000-01-13 00:00:00')  
>>> pd.to_datetime("7/8/2000")  
Timestamp('2000-07-08 00:00:00')
```

The last can refer to August 7th or July 8th, depending on the region. To disambiguate this case, `to_datetime` can be passed a keyword argument `dayfirst`:

```
>>> pd.to_datetime("7/8/2000", dayfirst=True)  
Timestamp('2000-08-07 00:00:00')
```

`Timestamp` objects can be seen as Pandas' version of `datetime` objects and indeed, the `Timestamp` class is a subclass of `datetime`:

```
>>> issubclass(pd.Timestamp, datetime.datetime)  
True
```

Which means they can be used interchangeably in many cases:

```
>>> ts = pd.to_datetime(9466848000000000000)
>>> ts.year, ts.month, ts.day, ts.weekday()
(2000, 1, 1, 5)
```

Timestamp objects are an important part of time series capabilities of Pandas, since timestamps are the building block of `DatetimeIndex` objects:

```
>>> index = [pd.Timestamp("2000-01-01"),
             pd.Timestamp("2000-01-02"),
             pd.Timestamp("2000-01-03")]
>>> ts = pd.Series(np.random.randn(len(index)), index=index)
>>> ts
2000-01-01    0.731897
2000-01-02    0.761540
2000-01-03   -1.316866
dtype: float64
>>> ts.indexDatetime
Index(['2000-01-01', '2000-01-02', '2000-01-03'],
      dtype='datetime64[ns]', freq=None, tz=None)
```

There are a few things to note here: We create a list of timestamp objects and pass it to the series constructor as `index`. This list of timestamps gets converted into a `DatetimeIndex` on the fly. If we had passed only the date strings, we would not get a `DatetimeIndex`, just an `index`:

```
>>> ts = pd.Series(np.random.randn(len(index)), index=[
                  "2000-01-01", "2000-01-02", "2000-01-03"])
>>> ts.index
Index([u'2000-01-01', u'2000-01-02', u'2000-01-03'], dtype='object')
```

However, the `to_datetime` function is flexible enough to be of help, if all we have is a list of date strings:

```
>>> index = pd.to_datetime(["2000-01-01", "2000-01-02", "2000-01-03"])
>>> ts = pd.Series(np.random.randn(len(index)), index=index)
>>> ts.index
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03'],
      dtype='datetime64[ns]', freq=None, tz=None)
```

Another thing to note is that while we have a `DatetimeIndex`, the `freq` and `tz` attributes are both `None`. We will learn about the utility of both attributes later in this chapter.

With `to_datetime` we are able to convert a variety of strings and even lists of strings into timestamp or `DatetimeIndex` objects. Sometimes we are not explicitly given all the information about a series and we have to generate sequences of time stamps of fixed intervals ourselves.

Pandas offer another great utility function for this task: `date_range`.

The `date_range` function helps to generate a fixed frequency `datetime` index between start and end dates. It is also possible to specify either the start or end date and the number of timestamps to generate.

The frequency can be specified by the `freq` parameter, which supports a number of offsets. You can use typical time intervals like hours, minutes, and seconds:

```
>>> pd.date_range(start="2000-01-01", periods=3, freq='H')
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 01:00:00',
               '2000-01-01 02:00:00'], dtype='datetime64[ns]', freq='H', tz=None)
>>> pd.date_range(start="2000-01-01", periods=3, freq='T')
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 00:01:00',
               '2000-01-01 00:02:00'], dtype='datetime64[ns]', freq='T', tz=None)

>>> pd.date_range(start="2000-01-01", periods=3, freq='S')
DatetimeIndex(['2000-01-01 00:00:00', '2000-01-01 00:00:01',
               '2000-01-01 00:00:02'], dtype='datetime64[ns]', freq='S', tz=None)
```

The `freq` attribute allows us to specify a multitude of options. Pandas has been used successfully in finance and economics, not least because it is really simple to work with business dates as well. As an example, to get an index with the first three business days of the millennium, the `B` offset alias can be used:

```
>>> pd.date_range(start="2000-01-01", periods=3, freq='B')
DatetimeIndex(['2000-01-03', '2000-01-04', '2000-01-05'],
              dtype='datetime64[ns]', freq='B', tz=None)
```

The following table shows the available offset aliases and can be also be looked up in the Pandas documentation on time series under <http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases>:

Alias	Description
B	Business day frequency
C	Custom business day frequency
D	Calendar day frequency
W	Weekly frequency

Alias	Description
M	Month end frequency
BM	Business month end frequency
CBM	Custom business month end frequency
MS	Month start frequency
BMS	Business month start frequency
CBMS	Custom business month start frequency
Q	Quarter end frequency
BQ	Business quarter frequency
QS	Quarter start frequency
BQS	Business quarter start frequency
A	Year end frequency
BA	Business year end frequency
AS	Year start frequency
BAS	Business year start frequency
BH	Business hour frequency
H	Hourly frequency
T	Minutely frequency
S	Secondly frequency
L	Milliseconds
U	Microseconds
N	Nanoseconds

Moreover, The offset aliases can be used in combination as well. Here, we are generating a `datetime` index with five elements, each one day, one hour, one minute and one second apart:

```
>>> pd.date_range(start="2000-01-01", periods=5, freq='1D1h1min10s')

DatetimeIndex(['2000-01-01 00:00:00', '2000-01-02 01:01:10',
               '2000-01-03 02:02:20', '2000-01-04 03:03:30',
               '2000-01-05 04:04:40'],
              dtype='datetime64[ns]', freq='90070S', tz=None)
```

If we want to index data every 12 hours of our business time, which by default starts at 9 AM and ends at 5 PM, we would simply prefix the `BH` alias:

```
>>> pd.date_range(start="2000-01-01", periods=5, freq='12BH')
DatetimeIndex(['2000-01-03 09:00:00', '2000-01-04 13:00:00',
               '2000-01-06 09:00:00', '2000-01-07 13:00:00',
               '2000-01-11 09:00:00'],  
              dtype='datetime64[ns]', freq='12BH', tz=None)
```

A custom definition of what a business hour means is also possible:

```
>>> ts.index
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03'],
              dtype='datetime64[ns]', freq=None, tz=None)
```

We can use this custom business hour to build indexes as well:

```
>>> pd.date_range(start="2000-01-01", periods=5, freq=12 * bh)
DatetimeIndex(['2000-01-03 07:00:00', '2000-01-03 19:00:00',
               '2000-01-04 07:00:00', '2000-01-04 19:00:00',
               '2000-01-05 07:00:00', '2000-01-05 19:00:00',
               '2000-01-06 07:00:00'],  
              dtype='datetime64[ns]', freq='12BH', tz=None)
```

Some frequencies allow us to specify an anchoring suffix, which allows us to express intervals, such as every Friday or every second Tuesday of the month:

```
>>> pd.date_range(start="2000-01-01", periods=5, freq='W-FRI')
DatetimeIndex(['2000-01-07', '2000-01-14', '2000-01-21', '2000-01-28',
               '2000-02-04'], dtype='datetime64[ns]', freq='W-FRI', tz=None)
>>> pd.date_range(start="2000-01-01", periods=5, freq='WOM-2TUE')
DatetimeIndex(['2000-01-11', '2000-02-08', '2000-03-14', '2000-04-11',
               '2000-05-09'], dtype='datetime64[ns]', freq='WOM-2TUE', tz=None)
```

Finally, we can merge various indexes of different frequencies. The possibilities are endless. We only show one example, where we combine two indexes – each over a decade – one pointing to every first business day of a year and one to the last day of February:

```
>>> s = pd.date_range(start="2000-01-01", periods=10, freq='BAS-JAN')
>>> t = pd.date_range(start="2000-01-01", periods=10, freq='A-FEB')
>>> s.union(t)
```

```
DatetimeIndex(['2000-01-03', '2000-02-29', '2001-01-01', '2001-02-28',
               '2002-01-01', '2002-02-28', '2003-01-01', '2003-02-28',
               '2004-01-01', '2004-02-29', '2005-01-03', '2005-02-28',
               '2006-01-02', '2006-02-28', '2007-01-01', '2007-02-28',
               '2008-01-01', '2008-02-29', '2009-01-01', '2009-02-28'],
              dtype='datetime64[ns]', freq=None, tz=None)
```

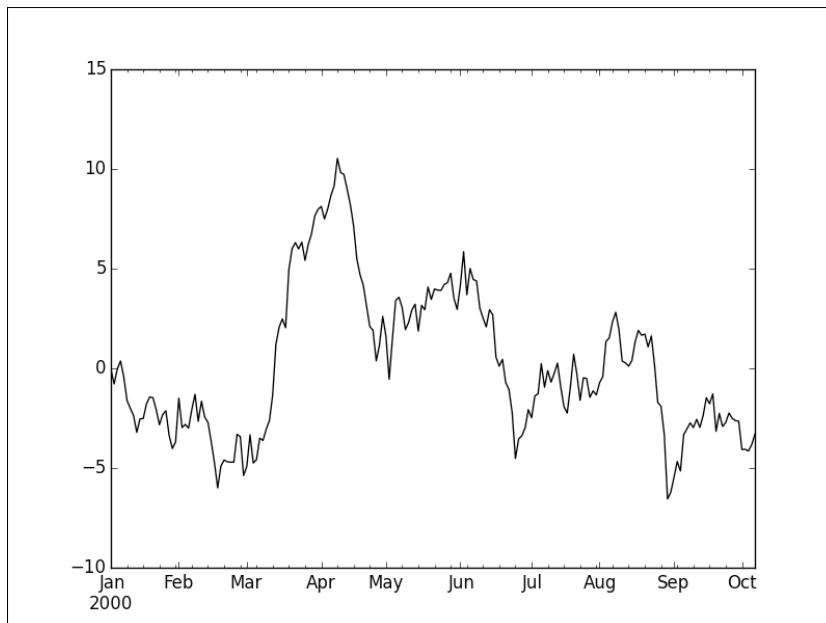
We see, that 2000 and 2005 did not start on a weekday and that 2000, 2004, and 2008 were the leap years.

We have seen two powerful functions so far, `to_datetime` and `date_range`. Now we want to dive into time series by first showing how you can create and plot time series data with only a few lines. In the rest of this section, we will show various ways to access and slice time series data.

It is easy to get started with time series data in Pandas. A random walk can be created and plotted in a few lines:

```
>>> index = pd.date_range(start='2000-01-01', periods=200, freq='B')
>>> ts = pd.Series(np.random.randn(len(index)), index=index)
>>> walk = ts.cumsum()
>>> walk.plot()
```

A possible output of this plot is show in the following figure:



Just as with usual series objects, you can select parts and slice the index:

```
>>> ts.head()
2000-01-03    1.464142
2000-01-04    0.103077
2000-01-05    0.762656
2000-01-06    1.157041
2000-01-07   -0.427284
Freq: B, dtype: float64
>>> ts[0]
1.4641415817112928
>>> ts[1:3]
2000-01-04    0.103077
2000-01-05    0.762656
```

We can use date strings as keys, even though our series has a DatetimeIndex:

```
>>> ts['2000-01-03']
1.4641415817112928
```

Even though the DatetimeIndex is made of timestamp objects, we can use datetime objects as keys as well:

```
>>> ts[datetime.datetime(2000, 1, 3)]
1.4641415817112928
```

Access is similar to lookup in dictionaries or lists, but more powerful. We can, for example, slice with strings or even mixed objects:

```
>>> ts['2000-01-03':'2000-01-05']
2000-01-03    1.464142
2000-01-04    0.103077
2000-01-05    0.762656
Freq: B, dtype: float64
>>> ts['2000-01-03':datetime.datetime(2000, 1, 5)]
2000-01-03    1.464142
2000-01-04    0.103077
2000-01-05    0.762656
Freq: B, dtype: float64
>>> ts['2000-01-03':datetime.date(2000, 1, 5)]
2000-01-03   -0.807669
2000-01-04    0.029802
2000-01-05   -0.434855
Freq: B, dtype: float64
```

It is even possible to use partial strings to select groups of entries. If we are only interested in February, we could simply write:

```
>>> ts['2000-02']
2000-02-01    0.277544
2000-02-02   -0.844352
2000-02-03   -1.900688
2000-02-04   -0.120010
2000-02-07   -0.465916
2000-02-08   -0.575722
2000-02-09    0.426153
2000-02-10    0.720124
2000-02-11    0.213050
2000-02-14   -0.604096
2000-02-15   -1.275345
2000-02-16   -0.708486
2000-02-17   -0.262574
2000-02-18    1.898234
2000-02-21    0.772746
2000-02-22    1.142317
2000-02-23   -1.461767
2000-02-24   -2.746059
2000-02-25   -0.608201
2000-02-28    0.513832
2000-02-29   -0.132000
```

To see all entries from March until May, including:

```
>>> ts['2000-03':'2000-05']
2000-03-01    0.528070
2000-03-02    0.200661
...
2000-05-30    1.206963
2000-05-31    0.230351
Freq: B, dtype: float64
```

Time series can be shifted forward or backward in time. The index stays in place, the values move:

```
>>> small_ts = ts['2000-02-01':'2000-02-05']
>>> small_ts
2000-02-01    0.277544
```

```
2000-02-02    -0.844352
2000-02-03    -1.900688
2000-02-04    -0.120010
Freq: B, dtype: float64
>>> small_ts.shift(2)
2000-02-01      NaN
2000-02-02      NaN
2000-02-03    0.277544
2000-02-04   -0.844352
Freq: B, dtype: float64
```

To shift backwards in time, we simply use negative values:

```
>>> small_ts.shift(-2)
2000-02-01   -1.900688
2000-02-02   -0.120010
2000-02-03      NaN
2000-02-04      NaN
Freq: B, dtype: float64
```

Resampling time series

Resampling describes the process of frequency conversion over time series data. It is a helpful technique in various circumstances as it fosters understanding by grouping together and aggregating data. It is possible to create a new time series from daily temperature data that shows the average temperature per week or month. On the other hand, real-world data may not be taken in uniform intervals and it is required to map observations into uniform intervals or to fill in missing values for certain points in time. These are two of the main use directions of resampling: binning and aggregation, and filling in missing data. Downsampling and upsampling occur in other fields as well, such as digital signal processing. There, the process of downsampling is often called decimation and performs a reduction of the sample rate. The inverse process is called **interpolation**, where the sample rate is increased. We will look at both directions from a data analysis angle.

Downsampling time series data

Downsampling reduces the number of samples in the data. During this reduction, we are able to apply aggregations over data points. Let's imagine a busy airport with thousands of people passing through every hour. The airport administration has installed a visitor counter in the main area, to get an impression of exactly how busy their airport is.

They are receiving data from the counter device every minute. Here are the hypothetical measurements for a day, beginning at 08:00, ending 600 minutes later at 18:00:

```
>>> rng = pd.date_range('4/29/2015 8:00', periods=600, freq='T')
>>> ts = pd.Series(np.random.randint(0, 100, len(rng)), index=rng)
>>> ts.head()
2015-04-29 08:00:00    9
2015-04-29 08:01:00   60
2015-04-29 08:02:00   65
2015-04-29 08:03:00   25
2015-04-29 08:04:00   19
```

To get a better picture of the day, we can downsample this time series to larger intervals, for example, 10 minutes. We can choose an aggregation function as well. The default aggregation is to take all the values and calculate the mean:

```
>>> ts.resample('10min').head()
2015-04-29 08:00:00    49.1
2015-04-29 08:10:00    56.0
2015-04-29 08:20:00    42.0
2015-04-29 08:30:00    51.9
2015-04-29 08:40:00    59.0
Freq: 10T, dtype: float64
```

In our airport example, we are also interested in the sum of the values, that is, the combined number of visitors for a given time frame. We can choose the aggregation function by passing a function or a function name to the how parameter works:

```
>>> ts.resample('10min', how='sum').head()
2015-04-29 08:00:00    442
2015-04-29 08:10:00    409
2015-04-29 08:20:00    532
2015-04-29 08:30:00    433
2015-04-29 08:40:00    470
Freq: 10T, dtype: int64
```

Or we can reduce the sampling interval even more by resampling to an hourly interval:

```
>>> ts.resample('1h', how='sum').head()
2015-04-29 08:00:00    2745
2015-04-29 09:00:00    2897
2015-04-29 10:00:00    3088
```

```
2015-04-29 11:00:00    2616
2015-04-29 12:00:00    2691
Freq: H, dtype: int64
```

We can ask for other things as well. For example, what was the maximum number of people that passed through our airport within one hour:

```
>>> ts.resample('1h', how='max').head()
2015-04-29 08:00:00    97
2015-04-29 09:00:00    98
2015-04-29 10:00:00    99
2015-04-29 11:00:00    98
2015-04-29 12:00:00    99
Freq: H, dtype: int64
```

Or we can define a custom function if we are interested in more unusual metrics. For example, we could be interested in selecting a random sample for each hour:

```
>>> import random
>>> ts.resample('1h', how=lambda m: random.choice(m)).head()
2015-04-29 08:00:00    28
2015-04-29 09:00:00    14
2015-04-29 10:00:00    68
2015-04-29 11:00:00    31
2015-04-29 12:00:00     5
```

If you specify a function by string, Pandas uses highly optimized versions.

The built-in functions that can be used as argument to how are: sum, mean, std, sem, max, min, median, first, last, ohlc. The ohlc metric is popular in finance. It stands for open-high-low-close. An OHLC chart is a typical way to illustrate movements in the price of a financial instrument over time.

While in our airport this metric might not be that valuable, we can compute it nonetheless:

```
>>> ts.resample('1h', how='ohlc').head()
          open  high  low  close
2015-04-29 08:00:00    9   97   0   14
2015-04-29 09:00:00   68   98   3   12
2015-04-29 10:00:00   71   99   1     1
2015-04-29 11:00:00   59   98   0     4
2015-04-29 12:00:00   56   99   3    55
```

Upsampling time series data

In upsampling, the frequency of the time series is increased. As a result, we have more sample points than data points. One of the main questions is how to account for the entries in the series where we have no measurement.

Let's start with hourly data for a single day:

```
>>> rng = pd.date_range('4/29/2015 8:00', periods=10, freq='H')
>>> ts = pd.Series(np.random.randint(0, 100, len(rng)), index=rng)
>>> ts.head()
2015-04-29 08:00:00    30
2015-04-29 09:00:00    27
2015-04-29 10:00:00    54
2015-04-29 11:00:00     9
2015-04-29 12:00:00    48
Freq: H, dtype: int64
```

If we upsample to data points taken every 15 minutes, our time series will be extended with NaN values:

```
>>> ts.resample('15min')
>>> ts.head()
2015-04-29 08:00:00    30
2015-04-29 08:15:00    NaN
2015-04-29 08:30:00    NaN
2015-04-29 08:45:00    NaN
2015-04-29 09:00:00    27
```

There are various ways to deal with missing values, which can be controlled by the `fill_method` keyword argument to `resample`. Values can be filled either forward or backward:

```
>>> ts.resample('15min', fill_method='ffill').head()
2015-04-29 08:00:00    30
2015-04-29 08:15:00    30
2015-04-29 08:30:00    30
2015-04-29 08:45:00    30
2015-04-29 09:00:00    27
Freq: 15T, dtype: int64
>>> ts.resample('15min', fill_method='bfill').head()
2015-04-29 08:00:00    30
2015-04-29 08:15:00    27
2015-04-29 08:30:00    27
```

```
2015-04-29 08:45:00    27
2015-04-29 09:00:00    27
```

With the `limit` parameter, it is possible to control the number of missing values to be filled:

```
>>> ts.resample('15min', fill_method='ffill', limit=2).head()
2015-04-29 08:00:00    30
2015-04-29 08:15:00    30
2015-04-29 08:30:00    30
2015-04-29 08:45:00    NaN
2015-04-29 09:00:00    27
Freq: 15T, dtype: float64
```

If you want to adjust the labels during resampling, you can use the `loffset` keyword argument:

```
>>> ts.resample('15min', fill_method='ffill', limit=2, loffset='5min').head()
2015-04-29 08:05:00    30
2015-04-29 08:20:00    30
2015-04-29 08:35:00    30
2015-04-29 08:50:00    NaN
2015-04-29 09:05:00    27
Freq: 15T, dtype: float64
```

There is another way to fill in missing values. We could employ an algorithm to construct new data points that would somehow fit the existing points, for some definition of somehow. This process is called interpolation.

We can ask Pandas to interpolate a time series for us:

```
>>> tsx = ts.resample('15min')
>>> tsx.interpolate().head()
2015-04-29 08:00:00    30.00
2015-04-29 08:15:00    29.25
2015-04-29 08:30:00    28.50
2015-04-29 08:45:00    27.75
2015-04-29 09:00:00    27.00
Freq: 15T, dtype: float64
```

We saw the default `interpolate` method – a linear interpolation – in action. Pandas assumes a linear relationship between two existing points.

Pandas supports over a dozen interpolation functions, some of which require the `scipy` library to be installed. We will not cover interpolation methods in this chapter, but we encourage you to explore the various methods yourself. The right interpolation method will depend on the requirements of your application.

Time zone handling

While, by default, Pandas objects are time zone unaware, many real-world applications will make use of time zones. As with working with time in general, time zones are no trivial matter: do you know which countries have daylight saving time and do you know when the time zone is switched in those countries? Thankfully, Pandas builds on the time zone capabilities of two popular and proven utility libraries for time and date handling: `pytz` and `dateutil`:

```
>>> t = pd.Timestamp('2000-01-01')
>>> t.tz is None
True
```

To supply time zone information, you can use the `tz` keyword argument:

```
>>> t = pd.Timestamp('2000-01-01', tz='Europe/Berlin')
>>> t.tz
<DstTzInfo 'Europe/Berlin' CET+1:00:00 STD>
```

This works for ranges as well:

```
>>> rng = pd.date_range('1/1/2000 00:00', periods=10, freq='D',
tz='Europe/London')
>>> rng
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04',
               '2000-01-05', '2000-01-06', '2000-01-07', '2000-01-08',
               '2000-01-09', '2000-01-10'],
              dtype='datetime64[ns]', freq='D', tz='Europe/London')
```

Time zone objects can also be constructed beforehand:

```
>>> import pytz
>>> tz = pytz.timezone('Europe/London')
>>> rng = pd.date_range('1/1/2000 00:00', periods=10, freq='D', tz=tz)
>>> rng
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04',
               '2000-01-05', '2000-01-06', '2000-01-07', '2000-01-08',
               '2000-01-09', '2000-01-10'],
              dtype='datetime64[ns]', freq='D', tz='Europe/London')
```

Sometimes, you will already have a time zone unaware time series object that you would like to make time zone aware. The `tz_localize` function helps to switch between time zone aware and time zone unaware objects:

```
>>> rng = pd.date_range('1/1/2000 00:00', periods=10, freq='D')
>>> ts = pd.Series(np.random.randn(len(rng)), rng)
>>> ts.index.tz is None
True
>>> ts_utc = ts.tz_localize('UTC')
>>> ts_utc.index.tz
<UTC>
```

To move a time zone aware object to other time zones, you can use the `tz_convert` method:

```
>>> ts_utc.tz_convert('Europe/Berlin').index.tz
<DstTzInfo 'Europe/Berlin' LMT+0:53:00 STD>
```

Finally, to detach any time zone information from an object, it is possible to pass `None` to either `tz_convert` or `tz_localize`:

```
>>> ts_utc.tz_convert(None).index.tz is None
True
>>> ts_utc.tz_localize(None).index.tz is None
True
```

Timedeltas

Along with the powerful timestamp object, which acts as a building block for the `DatetimeIndex`, there is another useful data structure, which has been introduced in Pandas 0.15 – the Timedelta. The Timedelta can serve as a basis for indices as well, in this case a `TimedeltaIndex`.

Timedeltas are differences in times, expressed in difference units. The `Timedelta` class in Pandas is a subclass of `datetime.timedelta` from the Python standard library. As with other Pandas data structures, the Timedelta can be constructed from a variety of inputs:

```
>>> pd.Timedelta('1 days')
Timedelta('1 days 00:00:00')
>>> pd.Timedelta('-1 days 2 min 10s 3us')
```

```
Timedelta('-2 days +23:57:49.999997')
>>> pd.Timedelta(days=1,seconds=1)
Timedelta('1 days 00:00:01')
```

As you would expect, Timedeltas allow basic arithmetic:

```
>>> pd.Timedelta(days=1) + pd.Timedelta(seconds=1)
Timedelta('1 days 00:00:01')
```

Similar to `to_datetime`, there is a `to_timedelta` function that can parse strings or lists of strings into Timedelta structures or TimedeltaIndices:

```
>>> pd.to_timedelta('20.1s')
Timedelta('0 days 00:00:20.100000')
```

Instead of absolute dates, we could create an index of timedeltas. Imagine measurements from a volcano, for example. We might want to take measurements but index it from a given date, for example the date of the last eruption. We could create a `timedelta` index that has the last seven days as entries:

```
>>> pd.to_timedelta(np.arange(7), unit='D')
TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days', '5
days', '6 days'], dtype='timedelta64[ns]', freq=None)
```

We could then work with time series data, indexed from the last eruption. If we had measurements for many eruptions (from possibly multiple volcanos), we would have an index that would make comparisons and analysis of this data easier. For example, we could ask whether there is a typical pattern that occurs between the third day and the fifth day after an eruption. This question would not be impossible to answer with a `DatetimeIndex`, but a `TimedeltaIndex` makes this kind of exploration much more convenient.

Time series plotting

Pandas comes with great support for plotting, and this holds true for time series data as well.

As a first example, let's take some monthly data and plot it:

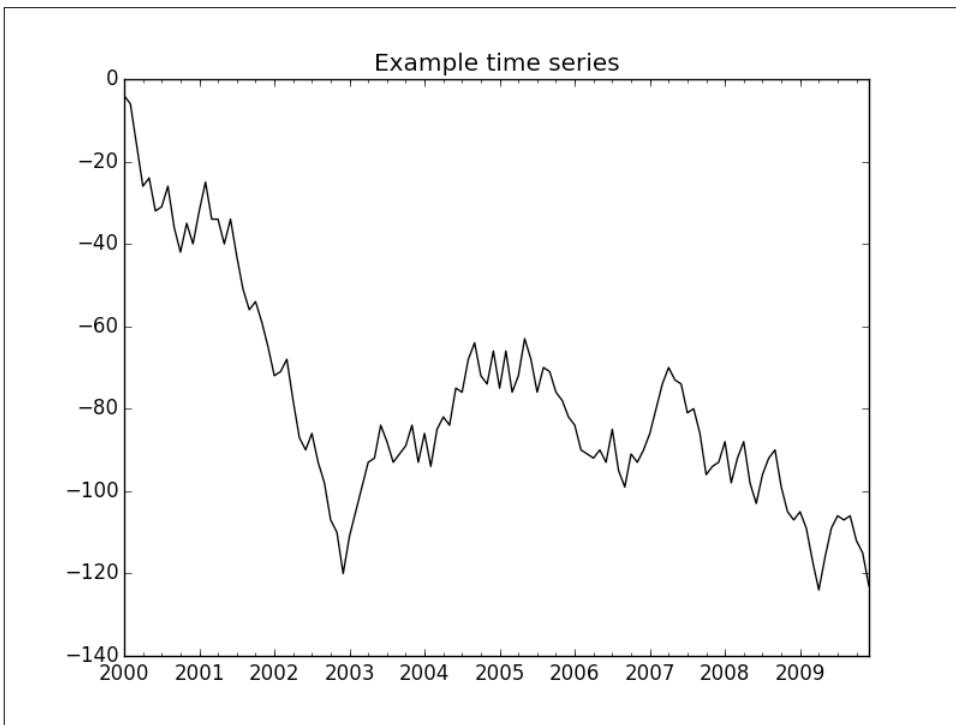
```
>>> rng = pd.date_range(start='2000', periods=120, freq='MS')
>>> ts = pd.Series(np.random.randint(-10, 10, size=len(rng)), rng).
cumsum()
>>> ts.head()
2000-01-01    -4
2000-02-01    -6
```

```
2000-03-01    -16
2000-04-01    -26
2000-05-01    -24
Freq: MS, dtype: int64
```

Since matplotlib is used under the hood, we can pass a familiar parameter to plot, such as c for color, or title for the chart title:

```
>>> ts.plot(c='k', title='Example time series')
>>> plt.show()
```

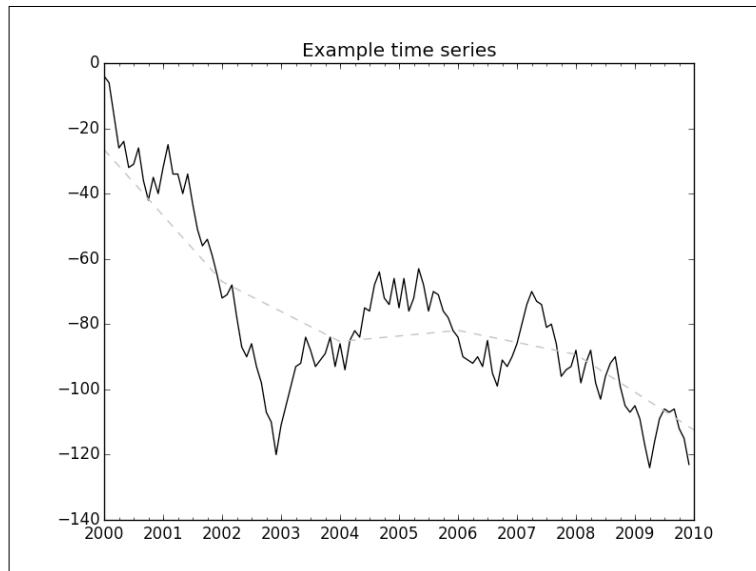
The following figure shows an example time series plot:



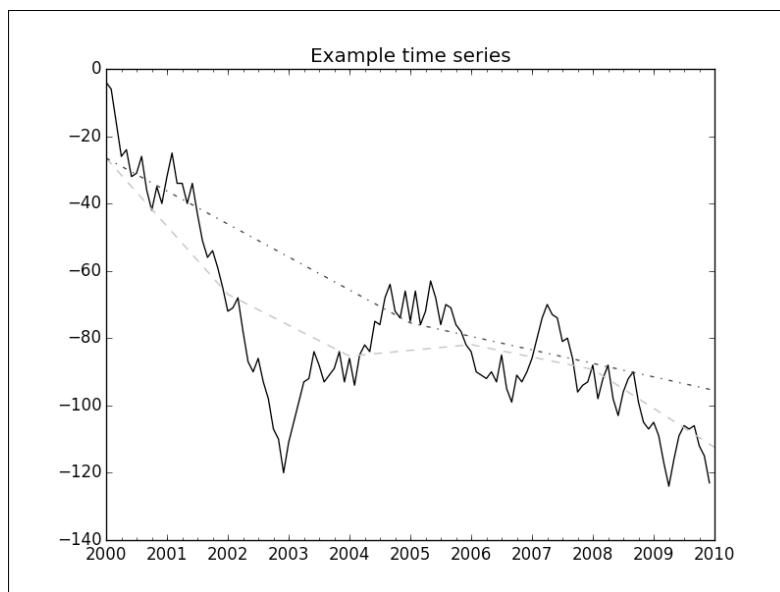
We can overlay an aggregate plot over 2 and 5 years:

```
>>> ts.resample('2A').plot(c='0.75', ls='--')
>>> ts.resample('5A').plot(c='0.25', ls='-.')
```

The following figure shows the resampled 2-year plot:

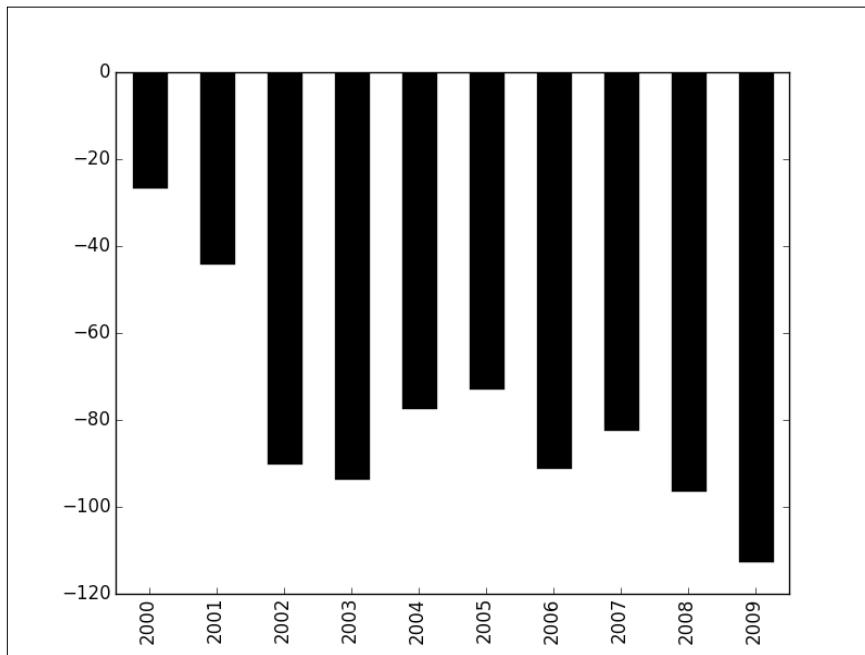


The following figure shows the resample 5-year plot:



We can pass the kind of chart to the plot method as well. The return value of the plot method is an AxesSubplot, which allows us to customize many aspects of the plot. Here we are setting the label values on the x axis to the year values from our time series:

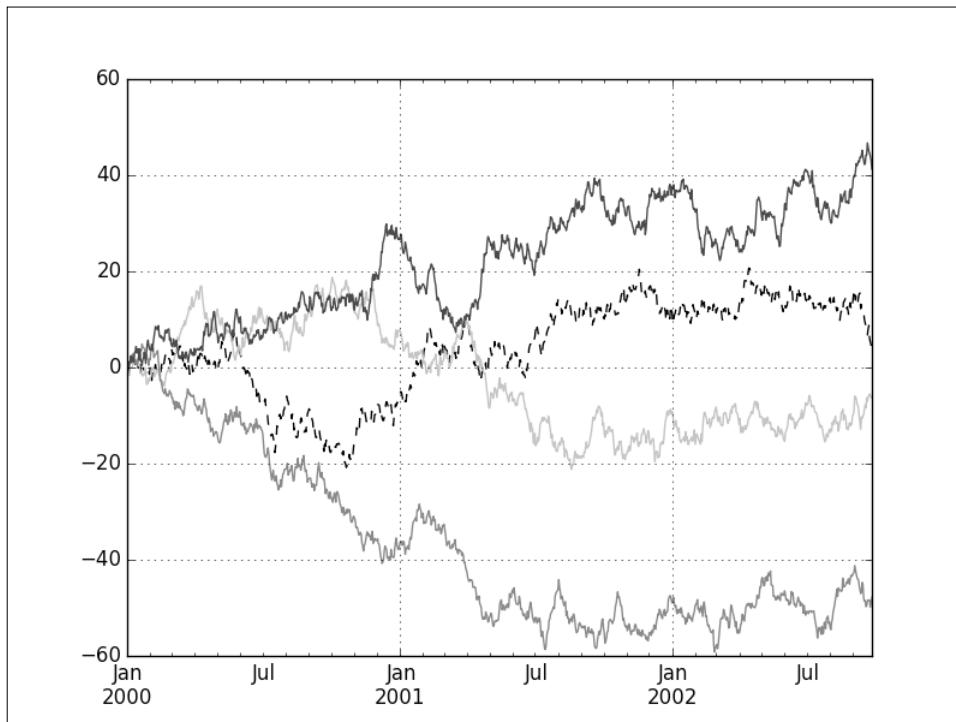
```
>>> plt.clf()  
>>> tsx = ts.resample('1A')  
>>> ax = tsx.plot(kind='bar', color='k')  
>>> ax.set_xticklabels(tsx.index.year)
```



Let's imagine we have four time series that we would like to plot simultaneously. We generate a matrix of 1000×4 random values and treat each column as a separated time series:

```
>>> plt.clf()  
>>> ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000',  
periods=1000))  
>>> df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,  
columns=['A', 'B', 'C', 'D'])
```

```
>>> df = df.cumsum()  
>>> df.plot(color=['k', '0.75', '0.5', '0.25'], ls='--')
```



Summary

In this chapter we showed how you can work with time series in Pandas. We introduced two index types, the `DatetimeIndex` and the `TimedeltaIndex` and explored their building blocks in depth. Pandas comes with versatile helper functions that take much of the pain out of parsing dates of various formats or generating fixed frequency sequences. Resampling data can help get a more condensed picture of the data, or it can help align various datasets of different frequencies to one another. One of the explicit goals of Pandas is to make it easy to work with missing data, which is also relevant in the context of upsampling.

Finally, we showed how time series can be visualized. Since matplotlib and Pandas are natural companions, we discovered that we can reuse our previous knowledge about matplotlib for time series data as well.

In the next chapter, we will explore ways to load and store data in text files and databases.

Practice examples

Exercise 1: Find one or two real-world examples for data sets, which could – in a sensible way – be assigned to the following groups:

- Fixed frequency data
- Variable frequency data
- Data where frequency is usually measured in seconds
- Data where frequency is measured in nanoseconds
- Data, where a `TimedeltaIndex` would be preferable

Create various fixed frequency ranges:

- Every minute between 1 AM and 2 AM on 2000-01-01
- Every two hours for a whole week starting 2000-01-01
- An entry for every Saturday and Sunday during the year 2000
- An entry for every Monday of a month, if it was a business day, for the years 2000, 2001 and 2002

6

Interacting with Databases

Data analysis starts with data. It is therefore beneficial to work with data storage systems that are simple to set up, operate and where the data access does not become a problem in itself. In short, we would like to have database systems that are easy to embed into our data analysis processes and workflows. In this book, we focus mostly on the Python side of the database interaction, and we will learn how to get data into and out of Pandas data structures.

There are numerous ways to store data. In this chapter, we are going to learn to interact with three main categories: text formats, binary formats and databases. We will focus on two storage solutions, MongoDB and Redis. MongoDB is a document-oriented database, which is easy to start with, since we can store JSON documents and do not need to define a schema upfront. Redis is a popular in-memory data structure store on top of which many applications can be built. It is possible to use Redis as a fast key-value store, but Redis supports lists, sets, hashes, bit arrays and even advanced data structures such as HyperLogLog out of the box as well.

Interacting with data in text format

Text is a great medium and it's a simple way to exchange information. The following statement is taken from a quote attributed to *Doug McIlroy*:
Write programs to handle text streams, because that is the universal interface.

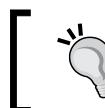
In this section we will start reading and writing data from and to text files.

Reading data from text format

Normally, the raw data logs of a system are stored in multiple text files, which can accumulate a large amount of information over time. Thankfully, it is simple to interact with these kinds of files in Python.

Pandas supports a number of functions for reading data from a text file into a DataFrame object. The most simple one is the `read_csv()` function. Let's start with a small example file:

```
$ cat example_data/ex_06-01.txt
Name,age,major_id,sex,hometown
Nam,7,1,male,hcm
Mai,11,1,female,hcm
Lan,25,3,female,hn
Hung,42,3,male,tn
Nghia,26,3,male,dn
Vinh,39,3,male,vl
Hong,28,4,female,dn
```



The `cat` is the Unix shell command that can be used to print the content of a file to the screen.

In the above example file, each column is separated by comma and the first row is a header row, containing column names. To read the data file into the DataFrame object, we type the following command:

```
>>> df_ex1 = pd.read_csv('example_data/ex_06-01.txt')
>>> df_ex1
   Name  age  major_id      sex hometown
0    Nam    7         1    male      hcm
1     Mai   11         1  female      hcm
2     Lan   25         3  female       hn
3    Hung   42         3    male       tn
4   Nghia   26         3    male       dn
5    Vinh   39         3    male       vl
6    Hong   28         4  female       dn
```

We see that the `read_csv` function uses a comma as the default delimiter between columns in the text file and the first row is automatically used as a header for the columns. If we want to change this setting, we can use the `sep` parameter to change the separated symbol and set `header=None` in case the example file does not have a caption row.

See the below example:

```
$ cat example_data/ex_06-02.txt
Nam    7      1      male   hcm
Mai    11     1      female  hcm
Lan    25     3      female  hn
Hung   42     3      male   tn
Nghia  26     3      male   dn
Vinh   39     3      male   vl
Hong   28     4      female  dn

>>> df_ex2 = pd.read_csv('example_data/ex_06-02.txt',
                           sep = '\t', header=None)
>>> df_ex2
      0    1    2      3    4
0    Nam   7   1    male  hcm
1    Mai  11   1  female  hcm
2    Lan  25   3  female  hn
3   Hung  42   3    male  tn
4  Nghia  26   3    male  dn
5   Vinh  39   3    male  vl
6   Hong  28   4  female  dn
```

We can also set a specific row as the caption row by using the header that's equal to the index of the selected row. Similarly, when we want to use any column in the data file as the column index of DataFrame, we set `index_col` to the name or index of the column. We again use the second data file `example_data/ex_06-02.txt` to illustrate this:

```
>>> df_ex3 = pd.read_csv('example_data/ex_06-02.txt',
                           sep = '\t', header=None,
                           index_col=0)
>>> df_ex3
      1    2      3    4
0
Nam    7   1    male  hcm
Mai   11   1  female  hcm
Lan   25   3  female  hn
```

```
Hung  42  3   male   tn
Nghia 26  3   male   dn
Vinh  39  3   male   vl
Hong  28  4   female dn
```

Apart from those parameters, we still have a lot of useful ones that can help us load data files into Pandas objects more effectively. The following table shows some common parameters:

Parameter	Value	Description
dtype	Type name or dictionary of type of columns	Sets the data type for data or columns. By default it will try to infer the most appropriate data type.
skiprows	List-like or integer	The number of lines to skip (starting from 0).
na_values	List-like or dict, default None	Values to recognize as NA/NaN. If a dict is passed, this can be set on a per-column basis.
true_values	List	A list of values to be converted to Boolean True as well.
false_values	List	A list of values to be converted to Boolean False as well.
keep_default_na	Bool, default True	If the na_values parameter is present and keep_default_na is False, the default NaN values are ignored, otherwise they are appended to
thousands	Str, default None	The thousands separator
nrows	Int, default None	Limits the number of rows to read from the file.
error_bad_lines	Boolean, default True	If set to True, a DataFrame is returned, even if an error occurred during parsing.

Besides the `read_csv()` function, we also have some other parsing functions in Pandas:

Function	Description
<code>read_table</code>	Read the general delimited file into DataFrame
<code>read_fwf</code>	Read a table of fixed-width formatted lines into DataFrame
<code>read_clipboard</code>	Read text from the clipboard and pass to <code>read_table</code> . It is useful for converting tables from web pages

In some situations, we cannot automatically parse data files from the disk using these functions. In that case, we can also open files and iterate through the reader, supported by the CSV module in the standard library:

```
$ cat example_data/ex_06-03.txt
Nam    7      1      male   hcm
Mai    11     1      female  hcm
Lan    25     3      female  hn
Hung   42     3      male   tn    single
Nghia  26     3      male   dn    single
Vinh   39     3      male   vl
Hong   28     4      female dn

>>> import csv
>>> f = open('data/ex_06-03.txt')
>>> r = csv.reader(f, delimiter='\t')
>>> for line in r:
>>>     print(line)
['Nam', '7', '1', 'male', 'hcm']
['Mai', '11', '1', 'female', 'hcm']
['Lan', '25', '3', 'female', 'hn']
['Hung', '42', '3', 'male', 'tn', 'single']
[['Nghia', '26', '3', 'male', 'dn', 'single']]
[['Vinh', '39', '3', 'male', 'vl']]
[['Hong', '28', '4', 'female', 'dn']]
```

Writing data to text format

We saw how to load data from a text file into a Pandas data structure. Now, we will learn how to export data from the data object of a program to a text file.

Corresponding to the `read_csv()` function, we also have the `to_csv()` function, supported by Pandas. Let's see an example below:

```
>>> df_ex3.to_csv('example_data/ex_06-02.out', sep = ';')
```

The result will look like this:

```
$ cat example_data/ex_06-02.out
0;1;2;3;4
Nam;7;1;male;hcm
Mai;11;1;female;hcm
Lan;25;3;female;hn
Hung;42;3;male;tn
Nghia;26;3;male;dn
Vinh;39;3;male;vl
Hong;28;4;female;dn
```

If we want to skip the header line or index column when writing out data into a disk file, we can set a `False` value to the `header` and `index` parameters:

```
>>> import sys
>>> df_ex3.to_csv(sys.stdout, sep='\t',
                    header=False, index=False)
7      1      male    hcm
11     1      female  hcm
25     3      female  hn
42     3      male    tn
26     3      male    dn
39     3      male    vl
28     4      female  dn
```

We can also write a subset of the columns of the DataFrame to the file by specifying them in the `columns` parameter:

```
>>> df_ex3.to_csv(sys.stdout, columns=[3,1,4],
                    header=False, sep='\t')
```

Nam	male	7	hcm
Mai	female	11	hcm
Lan	female	25	hn
Hung	male	42	tn
Nghia	male	26	dn
Vinh	male	39	vl
Hong	female	28	dn

With series objects, we can use the same function to write data into text files, with mostly the same parameters as above.

Interacting with data in binary format

We can read and write binary serialization of Python objects with the pickle module, which can be found in the standard library. Object serialization can be useful, if you work with objects that take a long time to create, like some machine learning models. By pickling such objects, subsequent access to this model can be made faster. It also allows you to distribute Python objects in a standardized way.

Pandas includes support for pickling out of the box. The relevant methods are the `read_pickle()` and `to_pickle()` functions to read and write data from and to files easily. Those methods will write data to disk in the pickle format, which is a convenient short-term storage format:

```
>>> df_ex3.to_pickle('example_data/ex_06-03.out')
>>> pd.read_pickle('example_data/ex_06-03.out')

      1   2       3   4
0
Nam    7   1   male  hcm
Mai   11   1 female  hcm
Lan   25   3   female  hn
Hung  42   3   male  tn
Nghia 26   3   male  dn
Vinh  39   3   male  vl
Hong  28   4   female  dn
```

HDF5

HDF5 is not a database, but a data model and file format. It is suited for write-one, read-many datasets. An HDF5 file includes two kinds of objects: data sets, which are array-like collections of data, and groups, which are folder-like containers what hold data sets and other groups. There are some interfaces for interacting with HDF5 format in Python, such as h5py which uses familiar NumPy and Python constructs, such as dictionaries and NumPy array syntax. With h5py, we have high-level interface to the HDF5 API which helps us to get started. However, in this book, we will introduce another library for this kind of format called PyTables, which works well with Pandas objects:

```
>>> store = pd.HDFStore('hdf5_store.h5')
>>> store
<class 'pandas.io.pytables.HDFStore'>
File path: hdf5_store.h5
Empty
```

We created an empty HDF5 file, named `hdf5_store.h5`. Now, we can write data to the file just like adding key-value pairs to a dict:

```
>>> store['ex3'] = df_ex3
>>> store['name'] = df_ex2[0]
>>> store['hometown'] = df_ex3[4]
>>> store
<class 'pandas.io.pytables.HDFStore'>
File path: hdf5_store.h5
/ex3           frame      (shape->[7, 4])
/hometown      series     (shape->[1])
/name          series     (shape->[1])
```

Objects stored in the HDF5 file can be retrieved by specifying the object keys:

```
>>> store['name']
0    Nam
1    Mai
2    Lan
3    Hung
4    Nghia
5    Vinh
6    Hong
Name: 0, dtype: object
```

Once we have finished interacting with the HDF5 file, we close it to release the file handle:

```
>>> store.close()
>>> store
<class 'pandas.io.pytables.HDFStore'>
File path: hdf5_store.h5
File is CLOSED
```

There are other supported functions that are useful for working with the HDF5 format. You should explore, in more detail, two libraries – pytables and h5py – if you need to work with huge quantities of data.

Interacting with data in MongoDB

Many applications require more robust storage systems than text files, which is why many applications use databases to store data. There are many kinds of databases, but there are two broad categories: relational databases, which support a standard declarative language called SQL, and so called NoSQL databases, which are often able to work without a predefined schema and where a data instance is more properly described as a document, rather as a row.

MongoDB is a kind of NoSQL database that stores data as documents, which are grouped together in collections. Documents are expressed as JSON objects. It is fast and scalable in storing, and also flexible in querying, data. To use MongoDB in Python, we need to import the pymongo package and open a connection to the database by passing a hostname and port. We suppose that we have a MongoDB instance, running on the default host (localhost) and port (27017):

```
>>> import pymongo
>>> conn = pymongo.MongoClient(host='localhost', port=27017)
```

If we do not put any parameters into the pymongo.MongoClient() function, it will automatically use the default host and port.

In the next step, we will interact with databases inside the MongoDB instance. We can list all databases that are available in the instance:

```
>>> conn.database_names()
['local']
>>> lc = conn.local
>>> lc
Database(MongoClient('localhost', 27017), 'local')
```

The above snippet says that our MongoDB instance only has one database, named 'local'. If the databases and collections we point to do not exist, MongoDB will create them as necessary:

```
>>> db = conn.db  
>>> db  
Database(MongoClient('localhost', 27017), 'db')
```

Each database contains groups of documents, called collections. We can understand them as tables in a relational database. To list all existing collections in a database, we use `collection_names()` function:

```
>>> lc.collection_names()  
['startup_log', 'system.indexes']  
>>> db.collection_names()  
[]
```

Our db database does not have any collections yet. Let's create a collection, named `person`, and insert data from a DataFrame object to it:

```
>>> collection = db.person  
>>> collection  
Collection(Database(MongoClient('localhost', 27017), 'db'), 'person')  
>>> # insert df_ex2 DataFrame into created collection  
>>> import json  

```

The `df_ex2` is transposed and converted to a JSON string before loading into a dictionary. The `insert()` function receives our created dictionary from `df_ex2` and saves it to the collection.

If we want to list all data inside the collection, we can execute the following commands:

```
>>> for cur in collection.find():
>>>     print(cur)
{'4': 'v1', '2': 3, '3': 'male', '1': 39, '_id':
ObjectId('557da218f21c761d7c176
a40'), '0': 'Vinh'}
{'4': 'dn', '2': 3, '3': 'male', '1': 26, '_id':
ObjectId('557da218f21c761d7c176
a41'), '0': 'Nghia'}
{'4': 'dn', '2': 4, '3': 'female', '1': 28, '_id':
ObjectId('557da218f21c761d7c1
76a42'), '0': 'Hong'}
{'4': 'hn', '2': 3, '3': 'female', '1': 25, '_id':
ObjectId('557da218f21c761d7c1
76a43'), '0': 'Lan'}
{'4': 'tn', '2': 3, '3': 'male', '1': 42, '_id':
ObjectId('557da218f21c761d7c176
a44'), '0': 'Hung'}
{'4': 'hcm', '2': 1, '3': 'male', '1': 7, '_id':
ObjectId('557da218f21c761d7c176
a45'), '0': 'Nam'}
{'4': 'hcm', '2': 1, '3': 'female', '1': 11, '_id':
ObjectId('557da218f21c761d7c
176a46'), '0': 'Mai'}
```

If we want to query data from the created collection with some conditions, we can use the `find()` function and pass in a dictionary describing the documents we want to retrieve. The returned result is a cursor type, which supports the iterator protocol:

```
>>> cur = collection.find({'3' : 'male'})
>>> type(cur)
pymongo.cursor.Cursor
>>> result = pd.DataFrame(list(cur))
```

```
>>> result
      0   1   2   3   4           _id
0  Vinh  39  3 male  vl  557da218f21c761d7c176a40
1  Nghia  26  3 male  dn  557da218f21c761d7c176a41
2   Hung  42  3 male  tn  557da218f21c761d7c176a44
3    Nam   7  1 male  hcm 557da218f21c761d7c176a45
```

Sometimes, we want to delete data in MongoDB. All we need to do is to pass a query to the `remove()` method on the collection:

```
>>> # before removing data
>>> pd.DataFrame(list(collection.find()))
      0   1   2   3   4           _id
0  Vinh  39  3 male  vl  557da218f21c761d7c176a40
1  Nghia  26  3 male  dn  557da218f21c761d7c176a41
2   Hong  28  4 female  dn  557da218f21c761d7c176a42
3    Lan  25  3 female  hn  557da218f21c761d7c176a43
4   Hung  42  3 male  tn  557da218f21c761d7c176a44
5    Nam   7  1 male  hcm 557da218f21c761d7c176a45
6    Mai  11  1 female  hcm 557da218f21c761d7c176a46

>>> # after removing records which have '2' column as 1 and '3' column as
'male'
>>> collection.remove({'2': 1, '3': 'male'})
{'n': 1, 'ok': 1}
>>> cur_all = collection.find();
>>> pd.DataFrame(list(cur_all))
      0   1   2   3   4           _id
0  Vinh  39  3 male  vl  557da218f21c761d7c176a40
1  Nghia  26  3 male  dn  557da218f21c761d7c176a41
2   Hong  28  4 female  dn  557da218f21c761d7c176a42
3    Lan  25  3 female  hn  557da218f21c761d7c176a43
4   Hung  42  3 male  tn  557da218f21c761d7c176a44
5    Mai  11  1 female  hcm 557da218f21c761d7c176a46
```

We learned step by step how to insert, query and delete data in a collection. Now, we will show how to update existing data in a collection in MongoDB:

```
>>> doc = collection.find_one({'1' : 42})
>>> doc['4'] = 'hcm'
>>> collection.save(doc)
ObjectId('557da218f21c761d7c176a44')
>>> pd.DataFrame(list(collection.find()))
   0   1   2      3   4          _id
0  Vinh  39  3    male   vl  557da218f21c761d7c176a40
1  Nghia  26  3    male   dn  557da218f21c761d7c176a41
2   Hong  28  4  female   dn  557da218f21c761d7c176a42
3    Lan  25  3  female   hn  557da218f21c761d7c176a43
4   Hung  42  3    male  hcm  557da218f21c761d7c176a44
5    Mai  11  1  female  hcm  557da218f21c761d7c176a46
```

The following table shows methods that provide shortcuts to manipulate documents in MongoDB:

Update Method	Description
inc()	Increment a numeric field
set()	Set certain fields to new values
unset()	Remove a field from the document
push()	Append a value onto an array in the document
pushAll()	Append several values onto an array in the document
addToSet()	Add a value to an array, only if it does not exist
pop()	Remove the last value of an array
pull()	Remove all occurrences of a value from an array
pullAll()	Remove all occurrences of any set of values from an array
rename()	Rename a field
bit()	Update a value by bitwise operation

Interacting with data in Redis

Redis is an advanced kind of key-value store where the values can be of different types: string, list, set, sorted set or hash. Redis stores data in memory like memcached but it can be persisted on disk, unlike memcached, which has no such option. Redis supports fast reads and writes, in the order of 100,000 set or get operations per second.

To interact with Redis, we need to install the `redis-py` module to Python, which is available on `pypi` and can be installed with `pip`:

```
$ pip install redis
```

Now, we can connect to Redis via the host and port of the DB server. We assume that we have already installed a Redis server, which is running with the default host (`localhost`) and port (`6379`) parameters:

```
>>> import redis
>>> r = redis.StrictRedis(host='127.0.0.1', port=6379)
>>> r
StrictRedis<ConnectionPool<Connection<host=localhost, port=6379, db=0>>>
```

As a first step to storing data in Redis, we need to define which kind of data structure is suitable for our requirements. In this section, we will introduce four commonly used data structures in Redis: simple value, list, set and ordered set. Though data is stored into Redis in many different data structures, each value must be associated with a key.

The simple value

This is the most basic kind of value in Redis. For every key in Redis, we also have a value that can have a data type, such as string, integer or double. Let's start with an example for setting and getting data to and from Redis:

```
>>> r.set('gender:An', 'male')
True
>>> r.get('gender:An')
b'male'
```

In this example we want to store the gender info of a person, named An into Redis. Our key is `gender:An` and our value is `male`. Both of them are a type of string.

The `set()` function receives two parameters: the key and the value. The first parameter is the key and the second parameter is value. If we want to update the value of this key, we just call the function again and change the value of the second parameter. Redis automatically updates it.

The `get()` function will retrieve the value of our key, which is passed as the parameter. In this case, we want to get gender information of the key `gender:An`.

In the second example, we show you another kind of value type, an integer:

```
>>> r.set('visited_time:An', 12)
True
>>> r.get('visited_time:An')
b'12'
>>> r.incr('visited_time:An', 1)
13
>>> r.get('visited_time:An')
b'13'
```

We saw a new function, `incr()`, which used to increment the value of key by a given amount. If our key does not exist, RedisDB will create the key with the given increment as the value.

List

We have a few methods for interacting with list values in Redis. The following example uses `rpush()` and `lrange()` functions to put and get list data to and from the DB:

```
>>> r.rpush('name_list', 'Tom')
1L
>>> r.rpush('name_list', 'John')
2L
>>> r.rpush('name_list', 'Mary')
3L
>>> r.rpush('name_list', 'Jan')
4L
>>> r.lrange('name_list', 0, -1)
[b'Tom', b'John', b'Mary', b'Jan']
>>> r.llen('name_list')
4
>>> r.lindex('name_list', 1)
b'John'
```

Besides the `rpush()` and `lrange()` functions we used in the example, we also want to introduce two others functions. First, the `llen()` function is used to get the length of our list in the Redis for a given key. The `lindex()` function is another way to retrieve an item of the list. We need to pass two parameters into the function: a key and an index of item in the list. The following table lists some other powerful functions in processing list data structure with Redis:

Function	Description
<code>rpushx(name, value)</code>	Push value onto the tail of the list name if name exists
<code>rpop(name)</code>	Remove and return the last item of the list name
<code>lset(name, index, value)</code>	Set item at the index position of the list name to input value
<code>lpushx(name, value)</code>	Push value on the head of the list name if name exists
<code>lpop(name)</code>	Remove and return the first item of the list name

Set

This data structure is also similar to the list type. However, in contrast to a list, we cannot store duplicate values in our set:

```
>>> r.sadd('country', 'USA')
1
>>> r.sadd('country', 'Italy')
1
>>> r.sadd('country', 'Singapore')
1
>>> r.sadd('country', 'Singapore')
0
>>> r.smembers('country')
{b'Italy', b'Singapore', b'USA'}
>>> r.srem('country', 'Singapore')
1
>>> r.smembers('country')
{b'Italy', b'USA'}
```

Corresponding to the list data structure, we also have a number of functions to get, set, update or delete items in the set. They are listed in the supported functions for set data structure, in the following table:

Function	Description
sadd(name, values)	Add value(s) to the set with key name
scard(name)	Return the number of element in the set with key name
smembers(name)	Return all members of the set with key name
srem(name, values)	Remove value(s) from the set with key name

Ordered set

The ordered set data structure takes an extra attribute when we add data to a set called **score**. An ordered set will use the score to determine the order of the elements in the set:

```
>>> r.zadd('person:A', 10, 'sub:Math')
1
>>> r.zadd('person:A', 7, 'sub:Bio')
1
>>> r.zadd('person:A', 8, 'sub:Chem')
1
>>> r.zrange('person:A', 0, -1)
[b'sub:Bio', b'sub:Chem', b'sub:Math']
>>> r.zrange('person:A', 0, -1, withscores=True)
[(b'sub:Bio', 7.0), (b'sub:Chem', 8.0), (b'sub:Math', 10.0)]
```

By using the zrange(name, start, end) function, we can get a range of values from the sorted set between the start and end score sorted in ascending order by default. If we want to change the way method of sorting, we can set the desc parameter to True. The withscore parameter is used in case we want to get the scores along with the return values. The return type is a list of (value, score) pairs as you can see in the above example.

See the below table for more functions available on ordered sets:

Function	Description
<code>zcard(name)</code>	Return the number of elements in the sorted set with key name
<code>zincrby(name, value, amount=1)</code>	Increment the score of value in the sorted set with key name by amount
<code>zrangebyscore(name, min, max, withscores=False, start=None, num=None)</code>	Return a range of values from the sorted set with key name with a score between min and max. If withscores is true, return the scores along with the values. If start and num are given, return a slice of the range
<code>zrank(name, value)</code>	Return a 0-based value indicating the rank of value in the sorted set with key name
<code>zrem(name, values)</code>	Remove member value(s) from the sorted set with key name

Summary

We finished covering the basics of interacting with data in different commonly used storage mechanisms from the simple ones, such as text files, over more structured ones, such as HDF5, to more sophisticated data storage systems, such as MongoDB and Redis. The most suitable type of storage will depend on your use case. The choice of the data storage layer technology plays an important role in the overall design of data processing systems. Sometimes, we need to combine various database systems to store our data, such as complexity of the data, performance of the system or computation requirements.

Practice exercise

- Take a data set of your choice and design storage options for it. Consider text files, HDF5, a document database, and a data structure store as possible persistent options. Also evaluate how difficult (by some metric, for example, how many lines of code) it would be to update or delete a specific item. Which storage type is the easiest to set up? Which storage type supports the most flexible queries?
- In *Chapter 3, Data Analysis with Pandas* we saw that it is possible to create hierarchical indices with Pandas. As an example, assume that you have data on each city with more than 1 million inhabitants and that we have a two level index, so we can address individual cities, but also whole countries. How would you represent this hierarchical relationship with the various storage options presented in this chapter: text files, HDF5, MongoDB, and Redis? What do you believe would be most convenient to work with in the long run?

7

Data Analysis Application Examples

In this chapter, we want to get you acquainted with typical data preparation tasks and analysis techniques, because being fluent in preparing, grouping, and reshaping data is an important building block for successful data analysis.

While preparing data seems like a mundane task – and often it is – it is a step we cannot skip, although we can strive to simplify it by using tools such as Pandas.

Why is preparation necessary at all? Because most useful data will come from the real world and will have deficiencies, contain errors or will be fragmentary.

There are more reasons why data preparation is useful: it gets you in close contact with the raw material. Knowing your input helps you to spot potential errors early and build confidence in your results.

Here are a few data preparation scenarios:

- A client hands you three files, each containing time series data about a single geological phenomenon, but the observed data is recorded on different intervals and uses different separators
- A machine learning algorithm can only work with numeric data, but your input only contains text labels
- You are handed the raw logs of a web server of an up and coming service and your task is to make suggestions on a growth strategy, based on existing visitor behavior

Data munging

The arsenal of tools for data munging is huge, and while we will focus on Python we want to mention some useful tools as well. If they are available on your system and you expect to work a lot with data, they are worth learning.

One group of tools belongs to the UNIX tradition, which emphasizes text processing and as a consequence has, over the last four decades, developed many high-performance and battle-tested tools for dealing with text. Some common tools are: sed, grep, awk, sort, uniq, tr, cut, tail, and head. They do very elementary things, such as filtering out lines (grep) or columns (cut) from files, replacing text (sed, tr) or displaying only parts of files (head, tail).

We want to demonstrate the power of these tools with a single example only.

Imagine you are handed the log files of a web server and you are interested in the distribution of the IP addresses.

Each line of the log file contains an entry in the common log server format (you can download this data set from <http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html>):

```
$ cat epa-html.txt
wpbf12-45.gate.net [29:23:56:12] "GET /Access/ HTTP/1.0" 200 2376ebaca.
icsi.net [30:00:22:20] "GET /Info.html HTTP/1.0" 200 884
```

For instance, we want to know how often certain users have visited our site.

We are interested in the first column only, since this is where the IP address or hostname can be found. After that, we need to count the number of occurrences of each host and finally display the results in a friendly way.

The `sort | uniq -c` stanza is our workhorse here: it sorts the data first and `uniq -c` will save the number of occurrences along with the value. The `sort -nr | head -15` is our formatting part; we sort numerically (-n) and in reverse (-r), and keep only the top 15 entries.

Putting it all together with pipes:

```
$ cut -d ' ' -f 1 epa-http.txt | sort | uniq -c | sort -nr | head -15
294 sandy.rtptok1.epa.gov
292 e659229.boeing.com
266 wicdgserv.wic.epa.gov
263 keyhole.es.dupont.com
248 dwilson.pr.mcs.net
176 oea4.r8stw56.epa.gov
174 macip26.nacion.co.cr
172 dcimsd23.dcimsd.epa.gov
167 www-b1.proxy.aol.com
158 piweba3y.prodigy.com
152 wictrn13.dcwictrn.epa.gov
151 nntp1.reach.com
151 inetg1.arco.com
149 canto04.nmsu.edu
146 weisman.metrokc.gov
```

With one command, we get to convert a sequential server log into an ordered list of the most common hosts that visited our site. We also see that we do not seem to have large differences in the number of visits among our top users.

There are more little helpful tools of which the following are just a tiny selection:

- `csvkit`: This is the suite of utilities for working with CSV, the king of tabular file formats
- `jq`: This is a lightweight and flexible command-line JSON processor
- `xmlstarlet`: This is a tool that supports XML queries with XPath, among other things
- `q`: This runs SQL on text files

Where the UNIX command line ends, lightweight languages take over. You might be able to get an impression from text only, but your colleagues might appreciate visual representations, such as charts or pretty graphs, generated by matplotlib, much more.

Python and its data tools ecosystem are much more versatile than the command line, but for first explorations and simple operations the effectiveness of the command line is often unbeatable.

Cleaning data

Most real-world data will have some defects and therefore will need to go through a cleaning step first. We start with a small file. Although this file contains only four rows, it will allow us to demonstrate the process up to a cleaned data set:

```
$ cat small.csv
22,6.1
41,5.7
18,5.3*
29,NaN
```

Note that this file has a few issues. The lines that contain values are all comma-separated, but we have missing (NA) and probably unclean (5.3*) values. We can load this file into a data frame, nevertheless:

```
>>> import pandas as pd
>>> df = pd.read_csv("small.csv")
>>> df
   22    6.1
0  41    5.7
1  18  5.3*
2  29    NaN
```

Pandas used the first row as header, but this is not what we want:

```
>>> df = pd.read_csv("small.csv", header=None)
>>> df
   0    1
0  22  6.1
1  41  5.7
2  18  5.3*
3  29  NaN
```

This is better, but instead of numeric values, we would like to supply our own column names:

```
>>> df = pd.read_csv("small.csv", names=["age", "height"])
>>> df
   age  height
0    22     6.1
1    41     5.7
2    18  5.3*
3    29     NaN
```

The age column looks good, since Pandas already inferred the intended type, but the height cannot be parsed into numeric values yet:

```
>>> df.age.dtype
dtype('int64')
>>> df.height.dtype
dtype('O')
```

If we try to coerce the height column into float values, Pandas will report an exception:

```
>>> df.height.astype('float')
ValueError: invalid literal for float(): 5.3*
```

We could use whatever value is parseable as a float and throw away the rest with the convert_objects method:

```
>>> df.height.convert_objects(convert_numeric=True)
0      6.1
1      5.7
2      NaN
3      NaN
Name: height, dtype: float64
```

If we know in advance the undesirable characters in our data set, we can augment the read_csv method with a custom converter function:

```
>>> remove_stars = lambda s: s.replace("*", "")
>>> df = pd.read_csv("small.csv", names=["age", "height"],
                     converters={"height": remove_stars})
>>> df
   age  height
0    22     6.1
1    41     5.7
2    18     5.3
3    29     NA
```

Now we can finally make the height column a bit more useful. We can assign it the updated version, which has the favored type:

```
>>> df.height = df.height.convert_objects(convert_numeric=True)
>>> df
   age  height
0    22      6.1
1    41      5.7
2    18      5.3
3    29      NaN
```

If we wanted to only keep the complete entries, we could drop any row that contains undefined values:

```
>>> df.dropna()
   age  height
0    22      6.1
1    41      5.7
2    18      5.3
```

We could use a default height, maybe a fixed value:

```
>>> df.fillna(5.0)
   age  height
0    22      6.1
1    41      5.7
2    18      5.3
3    29      5.0
```

On the other hand, we could also use the average of the existing values:

```
>>> df.fillna(df.height.mean())
   age  height
0    22      6.1
1    41      5.7
2    18      5.3
3    29      5.7
```

The last three data frames are complete and correct, depending on your definition of correct when dealing with missing values. Especially, the columns have the requested types and are ready for further analysis. Which of the data frames is best suited will depend on the task at hand.

Filtering

Even if we have clean and probably correct data, we might want to use only parts of it or we might want to check for outliers. An outlier is an observation point that is distant from other observations because of variability or measurement errors. In both cases, we want to reduce the number of elements in our data set to make it more relevant for further processing.

In this example, we will try to find potential outliers. We will use the Europe Brent Crude Oil Spot Price as recorded by the U.S. Energy Information Administration. The raw Excel data is available from http://www.eia.gov/dnav/pet/hist_xls/rbrted.xls (it can be found in the second worksheet). We cleaned the data slightly (the cleaning process is part of an exercise at the end of this chapter) and will work with the following data frame, containing 7160 entries, ranging from 1987 to 2015:

```
>>> df.head()
      date   price
0 1987-05-20  18.63
1 1987-05-21  18.45
2 1987-05-22  18.55
3 1987-05-25  18.60
4 1987-05-26  18.63
>>> df.tail()
      date   price
7155 2015-08-04  49.08
7156 2015-08-05  49.04
7157 2015-08-06  47.80
7158 2015-08-07  47.54
7159 2015-08-10  48.30
```

While many people know about oil prices – be it from the news or the filling station – let us forget anything we know about it for a minute. We could first ask for the extremes:

```
>>> df[df.price==df.price.min()]
      date   price
2937 1998-12-10    9.1
>>> df[df.price==df.price.max()]
      date   price
5373 2008-07-03 143.95
```

Another way to find potential outliers would be to ask for values that deviate most from the mean. We can use the `np.abs` function to calculate the deviation from the mean first:

```
>>> np.abs(df.price - df.price.mean())
0      26.17137  1      26.35137  7157      2.99863
7158    2.73863  7159      3.49863
```

We can now compare this deviation from a multiple – we choose 2.5 – of the standard deviation:

```
>>> import numpy as np
>>> df[np.abs(df.price - df.price.mean()) > 2.5 * df.price.std()]
   date   price
5354 2008-06-06  132.81
5355 2008-06-09  134.43
5356 2008-06-10  135.24
5357 2008-06-11  134.52
5358 2008-06-12  132.11
5359 2008-06-13  134.29
5360 2008-06-16  133.90
5361 2008-06-17  131.27
5363 2008-06-19  131.84
5364 2008-06-20  134.28
5365 2008-06-23  134.54
5366 2008-06-24  135.37
5367 2008-06-25  131.59
5368 2008-06-26  136.82
5369 2008-06-27  139.38
5370 2008-06-30  138.40
5371 2008-07-01  140.67
5372 2008-07-02  141.24
5373 2008-07-03  143.95
5374 2008-07-07  139.62
5375 2008-07-08  134.15
5376 2008-07-09  133.91
5377 2008-07-10  135.81
5378 2008-07-11  143.68
5379 2008-07-14  142.43
5380 2008-07-15  136.02
5381 2008-07-16  133.31
5382 2008-07-17  134.16
```

We see that those few days in summer 2008 must have been special. Sure enough, it is not difficult to find articles and essays with titles like *Causes and Consequences of the Oil Shock of 2007–08*. We have discovered a trace to these events solely by looking at the data.

We could ask the above question for each decade separately. We first make our data frame look more like a time series:

```
>>> df.index = df.date
>>> del df["date"]
>>> df.head()
   price
date
1987-05-20  18.63  1987-05-21  18.45
1987-05-22  18.55  1987-05-25  18.60
1987-05-26  18.63
```

We could filter out the eighties:

```
>>> decade = df["1980":"1989"]
>>> decade[np.abs(decade.price - decade.price.mean()) > 2.5 * decade.
price.std()]
   price
date
1988-10-03  11.60  1988-10-04  11.65  1988-10-05  11.20  1988-10-06
11.30  1988-10-07  11.35
```

We observe that within the data available (1987–1989), the fall of 1988 exhibits a slight spike in the oil prices. Similarly, during the nineties, we see that we have a larger deviation, in the fall of 1990:

```
>>> decade = df["1990":"1999"]
>>> decade[np.abs(decade.price - decade.price.mean()) > 5 * decade.price.
std()]
   price
date
1990-09-24  40.75  1990-09-26  40.85  1990-09-27  41.45  1990-09-28
41.00  1990-10-09  40.90  1990-10-10  40.20  1990-10-11  41.15
```

There are many more use cases for filtering data. Space and time are typical units: you might want to filter census data by state or city, or economical data by quarter. The possibilities are endless and will be driven by your project.

Merging data

The situation is common: you have multiple data sources, but in order to make statements about the content, you would rather combine them. Fortunately, Pandas' concatenation and merge functions abstract away most of the pain, when combining, joining, or aligning data. It does so in a highly optimized manner as well.

In a case where two data frames have a similar shape, it might be useful to just append one after the other. Maybe A and B are products and one data frame contains the number of items sold per product in a store:

```
>>> df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})  
>>> df1  
     A   B  
0   1   4  
1   2   5  
2   3   6  
  
>>> df2 = pd.DataFrame({'A': [4, 5, 6], 'B': [7, 8, 9]})  
>>> df2  
     A   B  
0   4   7  
1   5   8  
2   6   9  
  
>>> df1.append(df2)  
     A   B  
0   1   4  
1   2   5  
2   3   6  
0   4   7  
1   5   8  
2   6   9
```

Sometimes, we won't care about the indices of the originating data frames:

```
>>> df1.append(df2, ignore_index=True)  
     A   B  
0   1   4  
1   2   5  
2   3   6  
3   4   7  
4   5   8  
5   6   9
```

A more flexible way to combine objects is offered by the `pd.concat` function, which takes an arbitrary number of series, data frames, or panels as input. The default behavior resembles an append:

```
>>> pd.concat([df1, df2])  
   A   B  
0  1   4  
1  2   5  
2  3   6  
0  4   7  
1  5   8  
2  6   9
```

The default `concat` operation appends both frames along the rows – or index, which corresponds to axis 0. To concatenate along the columns, we can pass in the `axis` keyword argument:

```
>>> pd.concat([df1, df2], axis=1)  
   A   B   A   B  
0  1   4   4   7  
1  2   5   5   8  
2  3   6   6   9
```

We can add keys to create a hierarchical index.

```
>>> pd.concat([df1, df2], keys=['UK', 'DE'])  
    A   B  
UK 0  1   4  
    1   2   5  
    2   3   6  
DE 0  4   7  
    1   5   8  
    2   6   9
```

This can be useful if you want to refer back to parts of the data frame later. We use the `.ix` indexer:

```
>>> df3 = pd.concat([df1, df2], keys=['UK', 'DE'])  
>>> df3.ix["UK"]  
   A   B  
0  1   4  
1  2   5  
2  3   6
```

Data frames resemble database tables. It is therefore not surprising that Pandas implements SQL-like join operations on them. What is positively surprising is that these operations are highly optimized and extremely fast:

```
>>> import numpy as np
>>> df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
   ...:                 'value': range(4)})
>>> df1
   key  value
0    A      0
1    B      1
2    C      2
3    D      3
>>> df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
   ...:                 'value': range(10, 14)})
>>> df2
   key  value
0    B     10
1    D     11
2    D     12
3    E     13
```

If we merge on key, we get an inner join. This creates a new data frame by combining the column values of the original data frames based upon the join predicate, here the key attribute is used:

```
>>> df1.merge(df2, on='key')
   key  value_x  value_y
0    B         1       10
1    D         3       11
2    D         3       12
```

A left, right and full join can be specified by the how parameter:

```
>>> df1.merge(df2, on='key', how='left')
   key  value_x  value_y
0    A         0       NaN
1    B         1       10
2    C         2       NaN
3    D         3       11
4    D         3       12
```

```
>>> df1.merge(df2, on='key', how='right')
   key  value_x  value_y
0    B         1      10
1    D         3      11
2    D         3      12
3    E        NaN      13

>>> df1.merge(df2, on='key', how='outer')
   key  value_x  value_y
0    A         0      NaN
1    B         1      10
2    C         2      NaN
3    D         3      11
4    D         3      12
5    E        NaN      13
```

The merge methods can be specified with the how parameter. The following table shows the methods in comparison with SQL:

Merge Method	SQL Join Name	Description
left	LEFT OUTER JOIN	Use keys from the left frame only.
right	RIGHT OUTER JOIN	Use keys from the right frame only.
outer	FULL OUTER JOIN	Use a union of keys from both frames.
inner	INNER JOIN	Use an intersection of keys from both frames.

Reshaping data

We saw how to combine data frames but sometimes we have all the right data in a single data structure, but the format is impractical for certain tasks. We start again with some artificial weather data:

```
>>> df
      date     city  value
0  2000-01-03  London     6
1  2000-01-04  London     3
2  2000-01-05  London     4
3  2000-01-03  Mexico     3
4  2000-01-04  Mexico     9
5  2000-01-05  Mexico     8
6  2000-01-03  Mumbai    12
7  2000-01-04  Mumbai     9
```

```
8 2000-01-05 Mumbai 8
9 2000-01-03 Tokyo 5
10 2000-01-04 Tokyo 5
11 2000-01-05 Tokyo 6
```

If we want to calculate the maximum temperature per city, we could just group the data by city and then take the `max` function:

```
>>> df.groupby('city').max()
              date  value
city
London 2000-01-05      6
Mexico 2000-01-05      9
Mumbai 2000-01-05     12
Tokyo  2000-01-05      6
```

However, if we have to bring our data into form every time, we could be a little more effective, by creating a reshaped data frame first, having the dates as an index and the cities as columns.

We can create such a data frame with the `pivot` function. The arguments are the index (we use date), the columns (we use the cities), and the values (which are stored in the value column of the original data frame):

```
>>> pv = df.pivot("date", "city", "value")
>>> pv
city          London  Mexico  Mumbai  Tokyo
date
2000-01-03      6      3     12      5
2000-01-04      3      9      9      5
2000-01-05      4      8      8      6
```

We can use `max` function on this new data frame directly:

```
>>> pv.max()
city
London      6
Mexico      9
Mumbai     12
Tokyo      6
dtype: int64
```

With a more suitable shape, other operations become easier as well. For example, to find the maximum temperature per day, we can simply provide an additional axis argument:

```
>>> pv.max(axis=1)
date
2000-01-03    12
2000-01-04     9
2000-01-05     8
dtype: int64
```

Data aggregation

As a final topic, we will look at ways to get a condensed view of data with aggregations. Pandas comes with a lot of aggregation functions built-in. We already saw the `describe` function in *Chapter 3, Data Analysis with Pandas*. This works on parts of the data as well. We start with some artificial data again, containing measurements about the number of sunshine hours per city and date:

```
>>> df.head()
   country      city       date  hours
0  Germany    Hamburg  2015-06-01     8
1  Germany    Hamburg  2015-06-02    10
2  Germany    Hamburg  2015-06-03     9
3  Germany    Hamburg  2015-06-04     7
4  Germany    Hamburg  2015-06-05     3
```

To view a summary per `city`, we use the `describe` function on the grouped data set:

```
>>> df.groupby("city").describe()
                hours
city
Berlin      count    10.000000
              mean     6.000000
              std      3.741657
              min     0.000000
              25%     4.000000
              50%     6.000000
              75%     9.750000
              max    10.000000
```

```
Birmingham count    10.000000
               mean     5.100000
               std      2.078995
               min      2.000000
             25%      4.000000
             50%      5.500000
             75%      6.750000
               max     8.000000
```

On certain data sets, it can be useful to group by more than one attribute. We can get an overview about the sunny hours per country and date by passing in two column names:

```
>>> df.groupby(["country", "date"]).describe()
                hours
country date
France  2015-06-01  count    5.000000
                  mean     6.200000
                  std      1.095445
                  min      5.000000
                 25%      5.000000
                 50%      7.000000
                 75%      7.000000
                  max     7.000000
2015-06-02  count    5.000000
                  mean     3.600000
                  std      3.577709
                  min      0.000000
                 25%      0.000000
                 50%      4.000000
                 75%      6.000000
                  max     8.000000
UK       2015-06-07  std      3.872983
                  min      0.000000
                 25%      2.000000
                 50%      6.000000
                 75%      8.000000
                  max     9.000000
```

We can compute single statistics as well:

```
>>> df.groupby("city").mean()
          hours
city
Berlin      6.0
Birmingham   5.1
Bordeax      4.7
Edinburgh    7.5
Frankfurt    5.8
Glasgow      4.8
Hamburg      5.5
Leipzig       5.0
London        4.8
Lyon          5.0
Manchester   5.2
Marseille    6.2
Munich       6.6
Nice          3.9
Paris         6.3
```

Finally, we can define any function to be applied on the groups with the `agg` method. The above could have been written in terms of `agg` like this:

```
>>> df.groupby("city").agg(np.mean)
          hours
city
Berlin      6.0
Birmingham   5.1
Bordeax      4.7
Edinburgh    7.5
Frankfurt    5.8
Glasgow      4.8
...
...
```

But arbitrary functions are possible. As a last example, we define a `custom` function, which takes an input of a series object and computes the difference between the smallest and the largest element:

```
>>> df.groupby("city").agg(lambda s: abs(min(s) - max(s)))
          hours
```

```
city
Berlin      10
Birmingham   6
Bordeax      10
Edinburgh    8
Frankfurt    9
Glasgow      10
Hamburg      10
Leipzig      9
London       10
Lyon         8
Manchester   10
Marseille   10
Munich       9
Nice         10
Paris        9
```

Grouping data

One typical workflow during data exploration looks as follows:

- You find a criterion that you want to use to group your data. Maybe you have GDP data for every country along with the continent and you would like to ask questions about the continents. These questions usually lead to some function applications- you might want to compute the mean GDP per continent. Finally, you want to store this data for further processing in a new data structure.
- We use a simpler example here. Imagine some fictional weather data about the number of sunny hours per day and city:

```
>>> df
      date     city  value
0  2000-01-03  London    6
1  2000-01-04  London    3
2  2000-01-05  London    4
3  2000-01-03  Mexico    3
4  2000-01-04  Mexico    9
5  2000-01-05  Mexico    8
6  2000-01-03  Mumbai   12
7  2000-01-04  Mumbai    9
8  2000-01-05  Mumbai    8
```

```
9  2000-01-03    Tokyo      5
10 2000-01-04    Tokyo      5
11 2000-01-05    Tokyo      6
```

The groups attributes return a dictionary containing the unique groups and the corresponding values as axis labels:

```
>>> df.groupby("city").groups
{'London': [0, 1, 2],
'Mexico': [3, 4, 5],
'Mumbai': [6, 7, 8],
'Tokyo': [9, 10, 11]}
```

- Although the result of a groupby is a GroupBy object, not a DataFrame, we can use the usual indexing notation to refer to columns:

```
>>> grouped = df.groupby(["city", "value"])
>>> grouped["value"].max()
city
London      6
Mexico      9
Mumbai     12
Tokyo       6
Name: value, dtype: int64
>>> grouped["value"].sum()
city
London     13
Mexico     20
Mumbai     29
Tokyo      16
Name: value, dtype: int64
```

- We see that, according to our data set, Mumbai seems to be a sunny city. An alternative – and more verbose – way to achieve the above would be:

```
>>> df['value'].groupby(df['city']).sum()
city
London     13
Mexico     20
Mumbai     29
Tokyo      16
Name: value, dtype: int64
```

Summary

In this chapter we have looked at ways to manipulate data frames, from cleaning and filtering, to grouping, aggregation, and reshaping. Pandas makes a lot of the common operations very easy and more complex operations, such as pivoting or grouping by multiple attributes, can often be expressed as one-liners as well. Cleaning and preparing data is an essential part of data exploration and analysis.

The next chapter explains a brief of machine learning algorithms that is applying data analysis result to make decisions or build helpful products.

Practice exercises

Exercise 1: Cleaning: In the section about filtering, we used the Europe Brent Crude Oil Spot Price, which can be found as an Excel document on the internet. Take this Excel spreadsheet and try to convert it into a CSV document that is ready to be imported with Pandas.

Hint: There are many ways to do this. We used a small tool called `xls2csv.py` and we were able to load the resulting CSV file with a helper method:

```
import datetime
import pandas as pd
def convert_date(s):
    parts = s.replace("(", "").replace(")", "").split(",")
    if len(parts) < 6:
        return datetime.date(1970, 1, 1)
    return datetime.datetime(*[int(p) for p in parts])
df = pd.read_csv("RBRTEd.csv", sep=',', names=["date", "price"],
converters={"date": convert_date}).dropna()
```

Take a data set that is important for your work – or if you do not have any at hand, a data set that interests you and that is available online. Ask one or two questions about the data in advance. Then use cleaning, filtering, grouping, and plotting techniques to answer your question.

8

Machine Learning Models with scikit-learn

In the previous chapter, we saw how to perform data munging, data aggregation, and grouping. In this chapter, we will see the working of different scikit-learn modules for different models in brief, data representation in scikit-learn, understand supervised and unsupervised learning using an example, and measure prediction performance.

An overview of machine learning models

Machine learning is a subfield of artificial intelligence that explores how machines can learn from data to analyze structures, help with decisions, and make predictions. In 1959, Arthur Samuel defined machine learning as the, "Field of study that gives computers the ability to learn without being explicitly programmed."

A wide range of applications employ machine learning methods, such as spam filtering, optical character recognition, computer vision, speech recognition, credit approval, search engines, and recommendation systems.

One important driver for machine learning is the fact that data is generated at an increasing pace across all sectors; be it web traffic, texts or images, and sensor data or scientific datasets. The larger amounts of data give rise to many new challenges in storage and processing systems. On the other hand, many learning algorithms will yield better results with more data to learn from. The field has received a lot of attention in recent years due to significant performance increases in various hard tasks, such as speech recognition or object detection in images. Understanding large amounts of data without the help of intelligent algorithms seems unpromising.

A learning problem typically uses a set of samples (usually denoted with an N or n) to build a model, which is then validated and used to predict the properties of unseen data.

Each sample might consist of single or multiple values. In the context of machine learning, the properties of data are called features.

Machine learning can be arranged by the nature of the input data:

- Supervised learning
- Unsupervised learning

In supervised learning, the input data (typically denoted with x) is associated with a target label (y), whereas in unsupervised learning, we only have unlabeled input data.

Supervised learning can be further broken down into the following problems:

- Classification problems
- Regression problems

Classification problems have a fixed set of target labels, classes, or categories, while regression problems have one or more continuous output variables. Classifying e-mail messages as spam or not spam is a classification task with two target labels. Predicting house prices – given the data about houses, such as size, age, and nitric oxides concentration – is a regression task, since the price is continuous.

Unsupervised learning deals with datasets that do not carry labels. A typical case is clustering or automatic classification. The goal is to group similar items together. What similarity means will depend on the context, and there are many similarity metrics that can be employed in such a task.

The scikit-learn modules for different models

The scikit-learn library is organized into submodules. Each submodule contains algorithms and helper methods for a certain class of machine learning models and approaches.

Here is a sample of those submodules, including some example models:

Submodule	Description	Example models
cluster	This is the unsupervised clustering	KMeans and Ward
decomposition	This is the dimensionality reduction	PCA and NMF
ensemble	This involves ensemble-based methods	AdaBoostClassifier, AdaBoostRegressor, RandomForestClassifier, RandomForestRegressor
lda	This stands for latent discriminant analysis	LDA
linear_model	This is the generalized linear model	LinearRegression, LogisticRegression, Lasso and Perceptron
mixture	This is the mixture model	GMM and VBGMM
naive_bayes	This involves supervised learning based on Bayes' theorem	BaseNB and BernoulliNB, GaussianNB
neighbors	These are k-nearest neighbors	KNeighborsClassifier, KNeighborsRegressor, LSHForest
neural_network	This involves models based on neural networks	BernoulliRBM
tree	decision trees	DecisionTreeClassifier, DecisionTreeRegressor

While these approaches are diverse, a scikit-learn library abstracts away a lot of differences by exposing a regular interface to most of these algorithms. All of the example algorithms listed in the table implement a `fit` method, and most of them implement `predict` as well. These methods represent two phases in machine learning. First, the model is trained on the existing data with the `fit` method. Once trained, it is possible to use the model to predict the class or value of unseen data with `predict`. We will see both the methods at work in the next sections.

The scikit-learn library is part of the PyData ecosystem. Its codebase has seen steady growth over the past six years, and with over hundred contributors, it is one of the most active and popular among the scikit toolkits.

Data representation in scikit-learn

In contrast to the heterogeneous domains and applications of machine learning, the data representation in scikit-learn is less diverse, and the basic format that many algorithms expect is straightforward – a matrix of samples and features.

The underlying data structure is a numpy and the ndarray. Each row in the matrix corresponds to one sample and each column to the value of one feature.

There is something like Hello World in the world of machine learning datasets as well; for example, the Iris dataset whose origins date back to 1936. With the standard installation of scikit-learn, you already have access to a couple of datasets, including Iris that consists of 150 samples, each consisting of four measurements taken from three different Iris flower species:

```
>>> import numpy as np
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
```

The dataset is packaged as a bunch, which is only a thin wrapper around a dictionary:

```
>>> type(iris)
sklearn.datasets.base.Bunch
>>> iris.keys()
['target_names', 'data', 'target', 'DESCR', 'feature_names']
```

Under the data key, we can find the matrix of samples and features, and can confirm its shape:

```
>>> type(iris.data)
numpy.ndarray
>>> iris.data.shape
(150, 4)
```

Each entry in the data matrix has been labeled, and these labels can be looked up in the target attribute:

```
>>> type(iris.target)
numpy.ndarray
>>> iris.target.shape
(150,)
```

```
>>> iris.target[:10]
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>> np.unique(iris.target)
array([0, 1, 2])
```

The target names are encoded. We can look up the corresponding names in the `target_names` attribute:

```
>>> iris.target_names
>>> array(['setosa', 'versicolor', 'virginica'], dtype='|S10')
```

This is the basic anatomy of many datasets, such as example data, target values, and target names.

What are the features of a single entry in this dataset?:

```
>>> iris.data[0]
array([ 5.1,  3.5,  1.4,  0.2])
```

The four features are the measurements taken of real flowers: their sepal length and width, and petal length and width. Three different species have been examined: the **Iris-Setosa**, **Iris-Versicolour**, and **Iris-Virginica**.

Machine learning tries to answer the following question: can we predict the species of the flower, given only the measurements of its sepal and petal length?

In the next section, we will see how to answer this question with scikit-learn.

Besides the data about flowers, there are a few other datasets included in the scikit-learn distribution, as follows:

- The Boston House Prices dataset (506 samples and 13 attributes)
- The Optical Recognition of Handwritten Digits dataset (5620 samples and 64 attributes)
- The Iris Plants Database (150 samples and 4 attributes)
- The Linnerud dataset (30 samples and 3 attributes)

A few datasets are not included, but they can easily be fetched on demand (as these are usually a bit bigger). Among these datasets, you can find a real estate dataset and a news corpus:

```
>>> ds = datasets.fetch_california_housing()
downloading Cal. housing from http://lib.stat.cmu.edu/modules.php?op=...
>>> ds.data.shape
```

```
(20640, 8)
>>> ds = datasets.fetch_20newsgroups()
>>> len(ds.data)
11314
>>> ds.data[0][:50]
u"From: lerxst@wam.umd.edu (where's my thing)\nSubjec"
>>> sum([len([w for w in sample.split()]) for sample in ds.data])
3252437
```

These datasets are a great way to get started with the scikit-learn library, and they will also help you to test your own algorithms. Finally, scikit-learn includes functions (prefixed with `datasets.make_`) to create artificial datasets as well.

If you work with your own datasets, you will have to bring them in a shape that scikit-learn expects, which can be a task of its own. Tools such as Pandas make this task much easier, and Pandas DataFrames can be exported to `numpy.ndarray` easily with the `as_matrix()` method on DataFrame.

Supervised learning – classification and regression

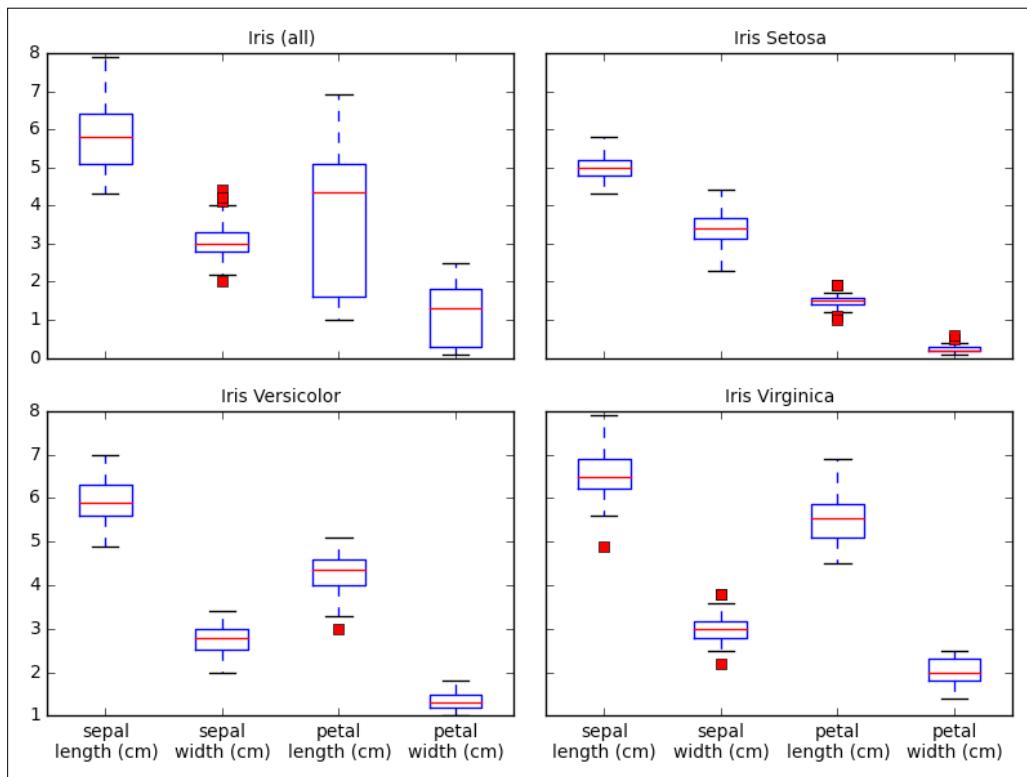
In this section, we will show short examples for both classification and regression.

Classification problems are pervasive: document categorization, fraud detection, market segmentation in business intelligence, and protein function prediction in bioinformatics.

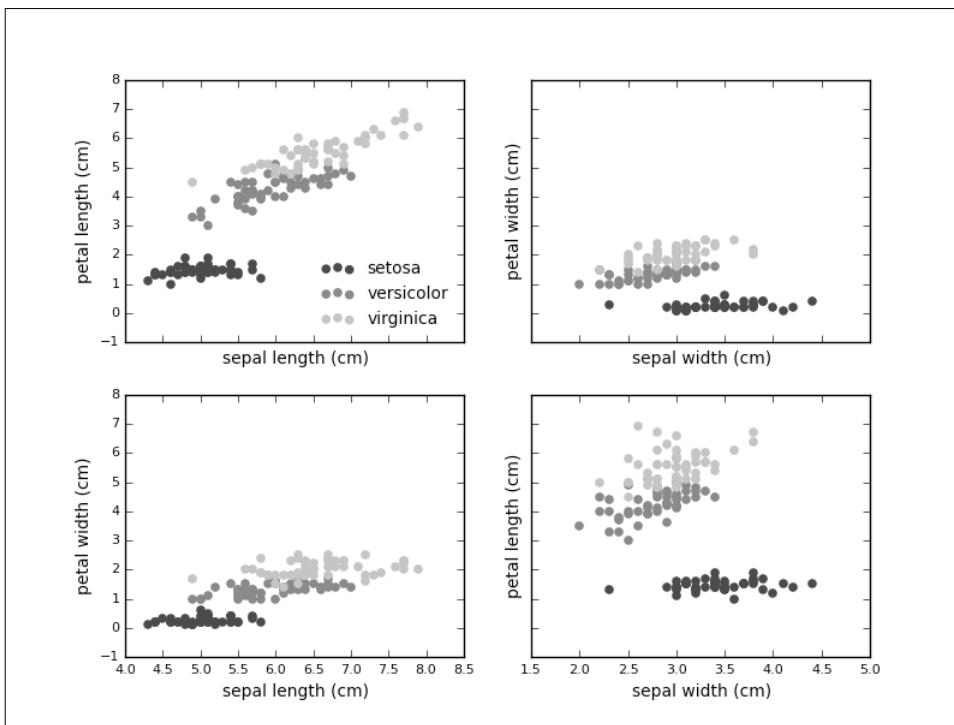
While it might be possible for hand-craft rules to assign a category or label to new data, it is faster to use algorithms to learn and generalize from the existing data.

We will continue with the Iris dataset. Before we apply a learning algorithm, we want to get an intuition of the data by looking at some values and plots.

All measurements share the same dimension, which helps to visualize the variance in various boxplots:



We see that the petal length (the third feature) exhibits the biggest variance, which could indicate the importance of this feature during classification. It is also insightful to plot the data points in two dimensions, using one feature for each axis. Also, indeed, our previous observation reinforced that the petal length might be a good indicator to tell apart the various species. The Iris setosa also seems to be more easily separable than the other two species:



From the visualizations, we get an intuition of the solution to our problem. We will use a supervised method called a **Support Vector Machine (SVM)** to learn about a classifier for the Iris data. The API separates models and data, therefore, the first step is to instantiate the model. In this case, we pass an optional keyword parameter to be able to query the model for probabilities later:

```
>>> from sklearn.svm import SVC
>>> clf = SVC(probability=True)
```

The next step is to fit the model according to our training data:

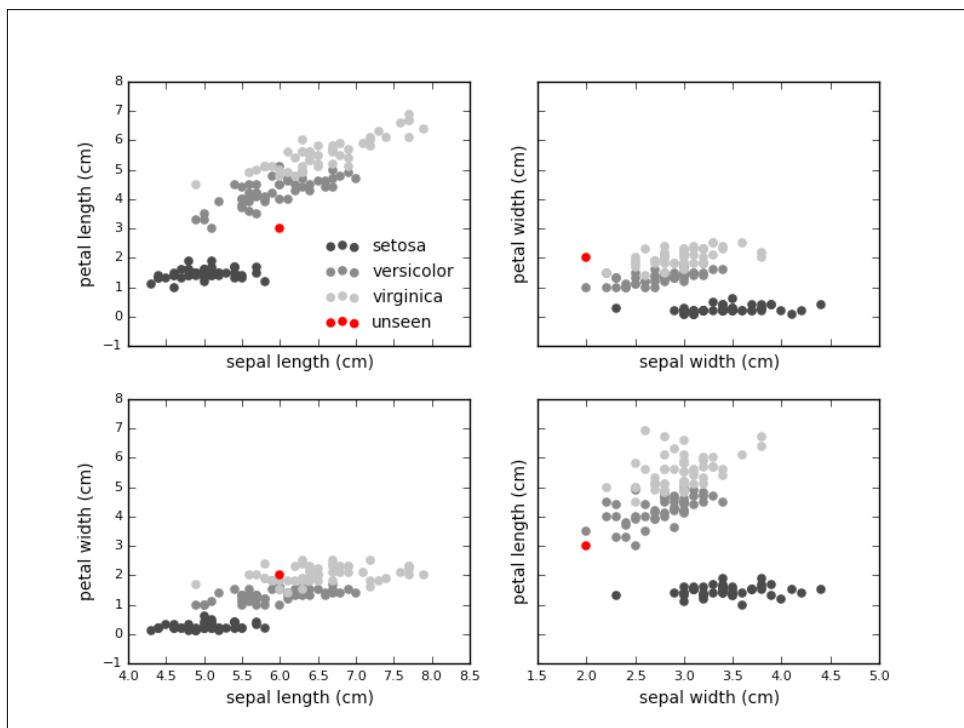
```
>>> clf.fit(iris.data, iris.target)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
```

```
degree=3, gamma=0.0, kernel='rbf', max_iter=-1,
probability=True, random_state=None, shrinking=True,
tol=0.001, verbose=False)
```

With this one line, we have trained our first machine learning model on a dataset. This model can now be used to predict the species of unknown data. If given some measurement that we have never seen before, we can use the predict method on the model:

```
>>> unseen = [6.0, 2.0, 3.0, 2.0]
>>> clf.predict(unseen)
array([1])
>>> iris.target_names[clf.predict(unseen)]
array(['versicolor',
       dtype='|S10')
```

We see that the classifier has given the `versicolor` label to the measurement. If we visualize the unknown point in our plots, we see that this seems like a sensible prediction:



In fact, the classifier is relatively sure about this label, which we can inquire into by using the `predict_proba` method on the classifier:

```
>>> clf.predict_proba(unseen)
array([[ 0.03314121,  0.90920125,  0.05765754]])
```

Our example consisted of four features, but many problems deal with higher-dimensional datasets and many algorithms work fine on these datasets as well.

We want to show another algorithm for supervised learning problems: linear regression. In linear regression, we try to predict one or more continuous output variables, called regress ands, given a D-dimensional input vector. Regression means that the output is continuous. It is called linear since the output will be modeled with a linear function of the parameters.

We first create a sample dataset as follows:

```
>>> import matplotlib.pyplot as plt
>>> X = [[1], [2], [3], [4], [5], [6], [7], [8]]
>>> y = [1, 2.5, 3.5, 4.8, 3.9, 5.5, 7, 8]
>>> plt.scatter(X, y, c='0.25')
>>> plt.show()
```

Given this data, we want to learn a linear function that approximates the data and minimizes the prediction error, which is defined as the sum of squares between the observed and predicted responses:

```
>>> from sklearn.linear_model import LinearRegression
>>> clf = LinearRegression()
>>> clf.fit(X, y)
```

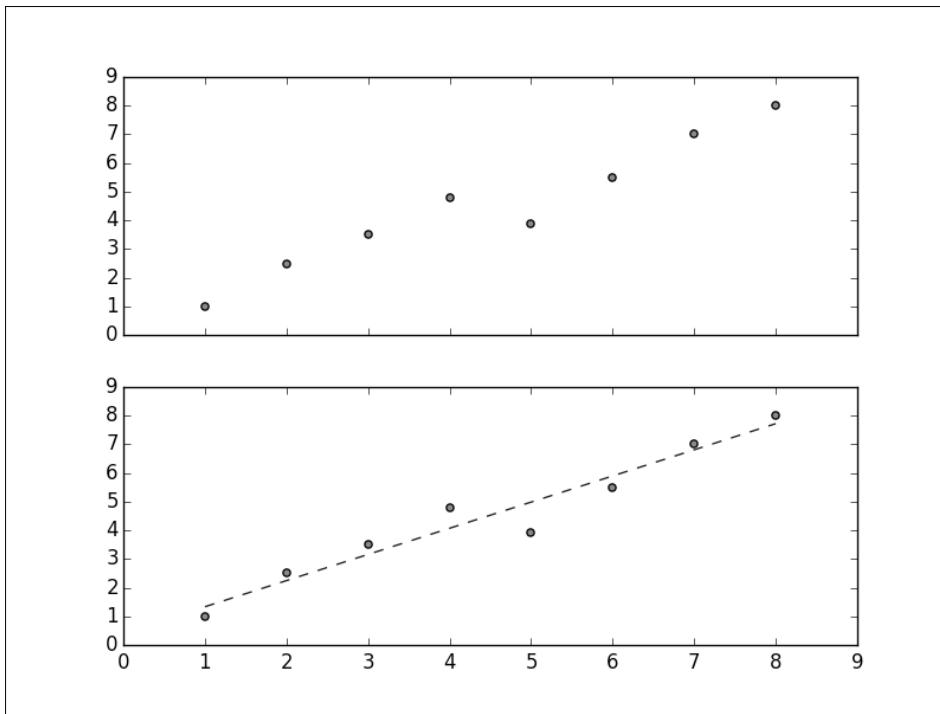
Many models will learn parameters during training. These parameters are marked with a single underscore at the end of the attribute name. In this model, the `coef_` attribute will hold the estimated coefficients for the linear regression problem:

```
>>> clf.coef_
array([ 0.91190476])
```

We can plot the prediction over our data as well:

```
>>> plt.plot(X, clf.predict(X), '--', color='0.10', linewidth=1)
```

The output of the plot is as follows:

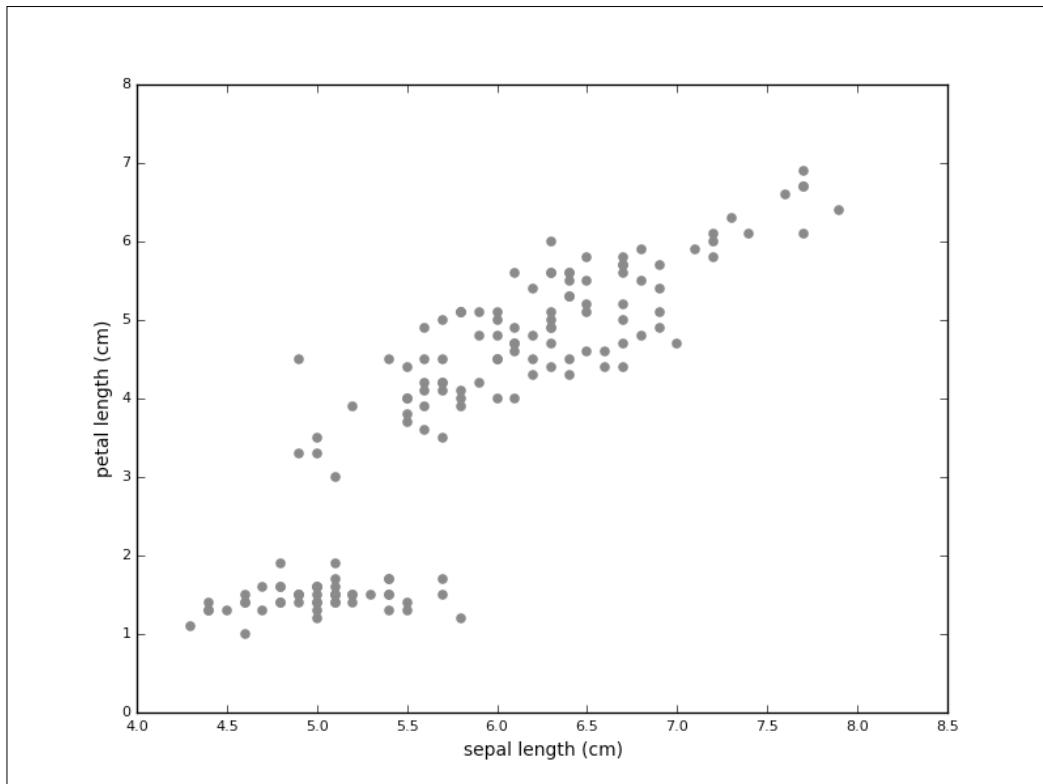


The above graph is a simple example with artificial data, but linear regression has a wide range of applications. If given the characteristics of real estate objects, we can learn to predict prices. If given the features of the galaxies, such as size, color, or brightness, it is possible to predict their distance. If given the data about household income and education level of parents, we can say something about the grades of their children.

There are numerous applications of linear regression everywhere, where one or more independent variables might be connected to one or more dependent variables.

Unsupervised learning – clustering and dimensionality reduction

A lot of existing data is not labeled. It is still possible to learn from data without labels with unsupervised models. A typical task during exploratory data analysis is to find related items or clusters. We can imagine the Iris dataset, but without the labels:



While the task seems much harder without labels, one group of measurements (in the lower-left) seems to stand apart. The goal of clustering algorithms is to identify these groups.

We will use K-Means clustering on the Iris dataset (without the labels). This algorithm expects the number of clusters to be specified in advance, which can be a disadvantage. K-Means will try to partition the dataset into groups, by minimizing the within-cluster sum of squares.

For example, we instantiate the `KMeans` model with `n_clusters` equal to 3:

```
>>> from sklearn.cluster import KMeans
>>> km = KMeans(n_clusters=3)
```

Similar to supervised algorithms, we can use the `fit` methods to train the model, but we only pass the data and not target labels:

```
>>> km.fit(iris.data)
KMeans(copy_x=True, init='k-means++', max_iter=300, n_clusters=3,
n_init=10, n_jobs=1, precompute_distances='auto', random_state=None,
tol=0.0001, verbose=0)
```

We already saw attributes ending with an underscore. In this case, the algorithm assigned a label to the training data, which can be inspected with the `labels_` attribute:

```
>>> km.labels_
array([1, 1, 1, 1, 1, 1, ..., 0, 2, 0, 0, 2], dtype=int32)
```

We can already compare the result of these algorithms with our known target labels:

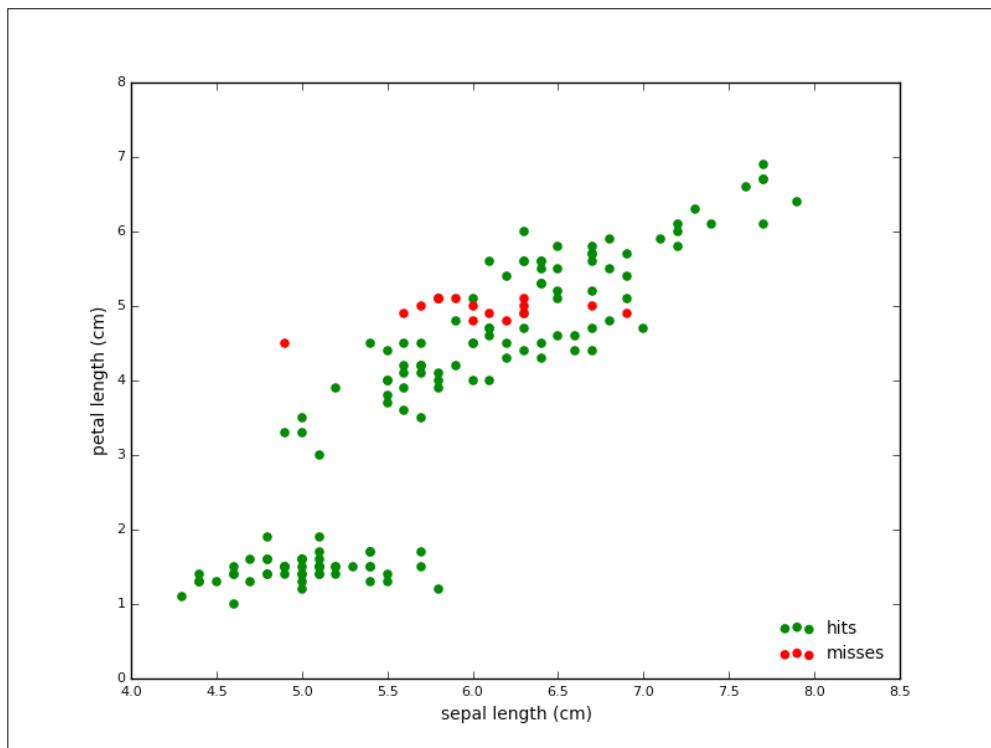
```
>>> iris.target
array([0, 0, 0, 0, 0, 0, ..., 2, 2, 2, 2, 2])
```

We quickly relabel the result to simplify the prediction error calculation:

```
>>> tr = {1: 0, 2: 1, 0: 2}
>>> predicted_labels = np.array([tr[i] for i in km.labels_])
>>> sum([p == t for (p, t) in zip(predicted_labels, iris.target)])
```

134

From 150 samples, K-Mean assigned the correct label to 134 samples, which is an accuracy of about 90 percent. The following plot shows the points of the algorithm predicted correctly in grey and the mislabeled points in red:



As another example for an unsupervised algorithm, we will take a look at **Principal Component Analysis (PCA)**. The PCA aims to find the directions of the maximum variance in high-dimensional data. One goal is to reduce the number of dimensions by projecting a higher-dimensional space onto a lower-dimensional subspace while keeping most of the information.

The problem appears in various fields. You have collected many samples and each sample consists of hundreds or thousands of features. Not all the properties of the phenomenon at hand will be equally important. In our Iris dataset, we saw that the petal length alone seemed to be a good discriminator of the various species. PCA aims to find principal components that explain most of the variation in the data. If we sort our components accordingly (technically, we sort the eigenvectors of the covariance matrix by eigenvalue), we can keep the ones that explain most of the data and ignore the remaining ones, thereby reducing the dimensionality of the data.

It is simple to run PCA with scikit-learn. We will not go into the implementation details, but instead try to give you an intuition of PCA by running it on the Iris dataset, in order to give you yet another angle.

The process is similar to the ones we implemented so far. First, we instantiate our model; this time, the PCA from the decomposition submodule. We also import a standardization method, called `StandardScaler`, that will remove the mean from our data and scale to the unit variance. This step is a common requirement for many machine learning algorithms:

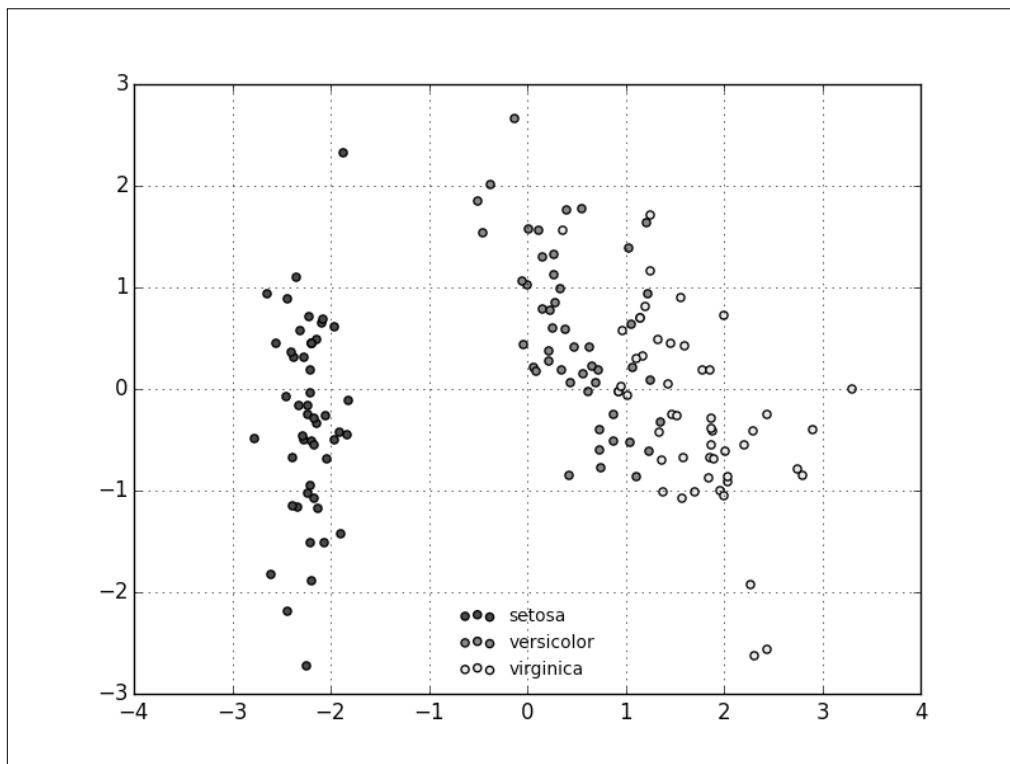
```
>>> from sklearn.decomposition import PCA  
>>> from sklearn.preprocessing import StandardScaler
```

First, we instantiate our model with a parameter (which specifies the number of dimensions to reduce to), standardize our input, and run the `fit_transform` function that will take care of the mechanics of PCA:

```
>>> pca = PCA(n_components=2)  
>>> X = StandardScaler().fit_transform(iris.data)  
>>> Y = pca.fit_transform(X)
```

The result is a dimensionality reduction in the Iris dataset from four (sepal and petal width and length) to two dimensions. It is important to note that this projection is not onto the two existing dimensions, so our new dataset does not consist of, for example, only petal length and width. Instead, the two new dimensions will represent a mixture of the existing features.

The following scatter plot shows the transformed dataset; from a glance at the plot, it looks like we still kept the essence of our dataset, even though we halved the number of dimensions:



Dimensionality reduction is just one way to deal with high-dimensional datasets, which are sometimes effected by the so called **curse of dimensionality**.

Measuring prediction performance

We have already seen that the machine learning process consists of the following steps:

- **Model selection:** We first select a suitable model for our data. Do we have labels? How many samples are available? Is the data separable? How many dimensions do we have? As this step is nontrivial, the choice will depend on the actual problem. As of Fall 2015, the scikit-learn documentation contains a much appreciated flowchart called *choosing the right estimator*. It is short, but very informative and worth taking a closer look at.

- **Training:** We have to bring the model and data together, and this usually happens in the fit methods of the models in scikit-learn.
- **Application:** Once we have trained our model, we are able to make predictions about the unseen data.

So far, we omitted an important step that takes place between the training and application: the model testing and validation. In this step, we want to evaluate how well our model has learned.

One goal of learning, and machine learning in particular, is generalization. The question of whether a limited set of observations is enough to make statements about any possible observation is a deeper theoretical question, which is answered in dedicated resources on machine learning.

Whether or not a model generalizes well can also be tested. However, it is important that the training and the test input are separate. The situation where a model performs well on a training input but fails on an unseen test input is called **overfitting**, and this is not uncommon.

The basic approach is to split the available data into a training and test set, and scikit-learn helps to create this split with the `train_test_split` function.

We go back to the Iris dataset and perform SVC again. This time we will evaluate the performance of the algorithm on a training set. We set aside 40 percent of the data for testing:

```
>>> from sklearn.cross_validation import train_test_split  
>>> X_train, X_test, y_train, y_test = train_test_split(  
    iris.data, iris.target, test_size=0.4, random_state=0)  
>>> clf = SVC()  
>>> clf.fit(X_train, y_train)
```

The score function returns the mean accuracy of the given data and labels. We pass the test set for evaluation:

```
>>> clf.score(X_test, y_test)  
0.9499999999999996
```

The model seems to perform well, with about 94 percent accuracy on unseen data. We can now start to tweak model parameters (also called hyper parameters) to increase prediction performance. This cycle would bring back the problem of overfitting. One solution is to split the input data into three sets: one for training, validation, and testing. The iterative model of hyper-parameters tuning would take place between the training and the validation set, while the final evaluation would be done on the test set. Splitting the dataset into three reduces the number of samples we can learn from as well.

Cross-validation (CV) is a technique that does not need a validation set, but still counteracts overfitting. The dataset is split into k parts (called folds). For each fold, the model is trained on $k-1$ folds and tested on the remaining folds. The accuracy is taken as the average over the folds.

We will show a five-fold cross-validation on the Iris dataset, using SVC again:

```
>>> from sklearn.cross_validation import cross_val_score
>>> clf = SVC()
>>> scores = cross_val_score(clf, iris.data, iris.target, cv=5)
>>> scores
array([ 0.96666667,  1.        ,  0.96666667,  0.96666667,  1.        ])
>>> scores.mean()
0.9800000000000009
```

There are various strategies implemented by different classes to split the dataset for cross-validation: KFold, StratifiedKFold, LeaveOneOut, LeavePOut, LeaveOneLabelOut, LeavePLableOut, ShuffleSplit, StratifiedShuffleSplit, and PredefinedSplit.

Model verification is an important step and it is necessary for the development of robust machine learning solutions.

Summary

In this chapter, we took a whirlwind tour through one of the most popular Python machine learning libraries: scikit-learn. We saw what kind of data this library expects. Real-world data will seldom be ready to be fed into an estimator right away. With powerful libraries, such as Numpy and, especially, Pandas, you already saw how data can be retrieved, combined, and brought into shape. Visualization libraries, such as matplotlib, help along the way to get an intuition of the datasets, problems, and solutions.

During this chapter, we looked at a canonical dataset, the Iris dataset. We also looked at it from various angles: as a problem in supervised and unsupervised learning and as an example for model verification.

In total, we have looked at four different algorithms: the Support Vector Machine, Linear Regression, K-Means clustering, and Principal Component Analysis. Each of these alone is worth exploring, and we barely scratched the surface, although we were able to implement all the algorithms with only a few lines of Python.

There are numerous ways in which you can take your knowledge of the data analysis process further. Hundreds of books have been published on machine learning, so we only want to highlight a few here: *Building Machine Learning Systems with Python* by Richert and Coelho, will go much deeper into scikit-learn as we couldn't in this chapter. *Learning from Data* by Abu-Mostafa, Magdon-Ismail, and Lin, is a great resource for a solid theoretical foundation of learning problems in general.

The most interesting applications will be found in your own field. However, if you would like to get some inspiration, we recommend that you look at the www.kaggle.com website that runs predictive modeling and analytics competitions, which are both fun and insightful.

Practice exercises

Are the following problems supervised or unsupervised? Regression or classification problems?:

- Recognizing coins inside a vending machine
- Recognizing handwritten digits
- If given a number of facts about people and economy, we want to estimate consumer spending
- If given the data about geography, politics, and historical events, we want to predict when and where a human right violation will eventually take place
- If given the sounds of whales and their species, we want to label yet unlabeled whale sound recordings

Look up one of the first machine learning models and algorithms: the perceptron. Try the perceptron on the Iris dataset and estimate the accuracy of the model. How does the perceptron compare to the SVC from this chapter?

Module 2

Python Data Analysis Cookbook

*Over 140 practical recipes to help you make sense of your data
with ease and build production-ready data apps*

1

Laying the Foundation for Reproducible Data Analysis

In this chapter, we will cover the following recipes:

- ▶ Setting up Anaconda
- ▶ Installing the Data Science Toolbox
- ▶ Creating a virtual environment with virtualenv and virtualenvwrapper
- ▶ Sandboxing Python applications with Docker images
- ▶ Keeping track of package versions and history in IPython Notebooks
- ▶ Configuring IPython
- ▶ Learning to log for robust error checking
- ▶ Unit testing your code
- ▶ Configuring pandas
- ▶ Configuring matplotlib
- ▶ Seeding random number generators and NumPy print options
- ▶ Standardizing reports, code style, and data access

Introduction

Reproducible data analysis is a cornerstone of good science. In today's rapidly evolving world of science and technology, reproducibility is a hot topic. Reproducibility is about lowering barriers for other people. It may seem strange or unnecessary, but reproducible analysis is essential to get your work acknowledged by others. If a lot of people confirm your results, it will have a positive effect on your career. However, reproducible analysis is hard. It has important economic consequences, as you can read in *Freedman LP, Cockburn IM, Simcoe TS (2015) The Economics of Reproducibility in Preclinical Research.* PLoS Biol 13(6): e1002165. doi:10.1371/journal.pbio.1002165.

So reproducibility is important for society and for you, but how does it apply to Python users? Well, we want to lower barriers for others by:

- ▶ Giving information about the software and hardware we used, including versions.
- ▶ Sharing virtual environments.
- ▶ Logging program behavior.
- ▶ Unit testing the code. This also serves as documentation of sorts.
- ▶ Sharing configuration files.
- ▶ Seeding random generators and making sure program behavior is as deterministic as possible.
- ▶ Standardizing reporting, data access, and code style.

I created the `dautil` package for this book, which you can install with pip or from the source archive provided in this book's code bundle. If you are in a hurry, run `$ python install_ch1.py` to install most of the software for this chapter, including `dautil`. I created a test Docker image, which you can use if you don't want to install anything except Docker (see the recipe, *Sandboxing Python applications with Docker images*).

Setting up Anaconda

Anaconda is a free Python distribution for data analysis and scientific computing. It has its own package manager, **conda**. The distribution includes more than 200 Python packages, which makes it very convenient. For casual users, the **Miniconda** distribution may be the better choice. Miniconda contains the conda package manager and Python. The technical editors use Anaconda, and so do I. But don't worry, I will describe in this book alternative installation instructions for readers who are not using Anaconda. In this recipe, we will install Anaconda and Miniconda and create a virtual environment.

Getting ready

The procedures to install Anaconda and Miniconda are similar. Obviously, Anaconda requires more disk space. Follow the instructions on the Anaconda website at <http://conda.pydata.org/docs/install/quick.html> (retrieved Mar 2016). First, you have to download the appropriate installer for your operating system and Python version. Sometimes, you can choose between a GUI and a command-line installer. I used the Python 3.4 installer, although my system Python version is v2.7. This is possible because Anaconda comes with its own Python. On my machine, the Anaconda installer created an anaconda directory in my home directory and required about 900 MB. The Miniconda installer installs a miniconda directory in your home directory.

How to do it...

1. Now that Anaconda or Miniconda is installed, list the packages with the following command:
`$ conda list`
2. For reproducibility, it is good to know that we can export packages:
`$ conda list --export`
3. The preceding command prints packages and versions on the screen, which you can save in a file. You can install these packages with the following command:
`$ conda create -n ch1env --file <export file>`
This command also creates an environment named ch1env.
4. The following command creates a simple testenv environment:
`$ conda create --name testenv python=3`
5. On Linux and Mac OS X, switch to this environment with the following command:
`$ source activate testenv`
6. On Windows, we don't need source. The syntax to switch back is similar:
`$ [source] deactivate`
7. The following command prints export information for the environment in the YAML (explained in the following section) format:
`$ conda env export -n testenv`
8. To remove the environment, type the following (note that even after removing, the name of the environment still exists in `~/.conda/environments.txt`):
`$ conda remove -n testenv --all`

9. Search for a package as follows:

```
$ conda search numpy
```

In this example, we searched for the NumPy package. If NumPy is already present, Anaconda shows an asterisk in the output at the corresponding entry.

10. Update the distribution as follows:

```
$ conda update conda
```

There's more...

The `.condarc` configuration file follows the **YAML** syntax.



YAML is a human-readable configuration file format with the extension `.yaml` or `.yml`. YAML was initially released in 2011, with the latest release in 2009. The YAML homepage is at <http://yaml.org/> (retrieved July 2015).

You can find a sample configuration file at <http://conda.pydata.org/docs/install/sample-condarc.html> (retrieved July 2015). The related documentation is at <http://conda.pydata.org/docs/install/config.html> (retrieved July 2015).

See also

- ▶ Martins, L. Felipe (November 2014). *IPython Notebook Essentials* (1st Edition.). Packt Publishing. p. 190. ISBN 1783988347
- ▶ The conda user cheat sheet at http://conda.pydata.org/docs/_downloads/conda-cheatsheet.pdf (retrieved July 2015)

Installing the Data Science Toolbox

The **Data Science Toolbox (DST)** is a virtual environment based on Ubuntu for data analysis using Python and R. Since DST is a virtual environment, we can install it on various operating systems. We will install DST locally, which requires **VirtualBox** and **Vagrant**. VirtualBox is a virtual machine application originally created by Innotek GmbH in 2007. Vagrant is a wrapper around virtual machine applications such as VirtualBox created by Mitchell Hashimoto.

Getting ready

You need to have in the order of 2 to 3 GB free for VirtualBox, Vagrant, and DST itself. This may vary by operating system.

How to do it...

Installing DST requires the following steps:

1. Install VirtualBox by downloading an installer for your operating system and architecture from <https://www.virtualbox.org/wiki/Downloads> (retrieved July 2015) and running it. I installed VirtualBox 4.3.28-100309 myself, but you can just install whatever the most recent VirtualBox version at the time is.
2. Install Vagrant by downloading an installer for your operating system and architecture from <https://www.vagrantup.com/downloads.html> (retrieved July 2015). I installed Vagrant 1.7.2 and again you can install a more recent version if available.
3. Create a directory to hold the DST and navigate to it with a terminal. Run the following command:

```
$ vagrant init data-science-toolbox/dst
```

The first command creates a `Vagrantfile` configuration file. Most of the content is commented out, but the file does contain links to documentation that might be useful. The second command creates the DST and initiates a download that could take a couple of minutes.

4. Connect to the virtual environment as follows (on Windows use putty):

```
$ vagrant ssh
```

5. View the preinstalled Python packages with the following command:

```
vagrant@data-science-toolbox:~$ pip freeze
```

The list is quite long; in my case it contained 32 packages. The DST Python version as of July 2015 was 2.7.6.

6. When you are done with the DST, log out and suspend (you can also halt it completely) the VM:

```
vagrant@data-science-toolbox:~$ logout  
Connection to 127.0.0.1 closed.  
$ vagrant suspend  
==> default: Saving VM state and suspending execution...
```

How it works...

Virtual machines (VMs) emulate computers in software. **VirtualBox** is an application that creates and manages VMs. VirtualBox stores its VMs in your home folder, and this particular VM takes about 2.2 GB of storage.

Ubuntu is an open source Linux operating system, and we are allowed by its license to create virtual machines. Ubuntu has several versions; we can get more info with the `lsb_release` command:

```
vagrant@data-science-toolbox:~$ lsb_release -a
No LSB modules are available.
Distributor ID:    Ubuntu
Description:    Ubuntu 14.04 LTS
Release:    14.04
Codename:   trusty
```

Vagrant used to only work with VirtualBox, but currently it also supports VMware, KVM, Docker, and Amazon EC2. Vagrant calls virtual machines boxes. Some of these boxes are available for everyone at <http://www.vagrantbox.es/> (retrieved July 2015).

See also

- ▶ *Run Ubuntu Linux Within Windows Using VirtualBox* at <http://linux.about.com/od/howtos/ss/Run-Ubuntu-Linux-Within-Windows-Using-VirtualBox.htm#step11> (retrieved July 2015)
- ▶ *VirtualBox manual chapter 10 Technical Information* at <https://www.virtualbox.org/manual/ch10.html> (retrieved July 2015)

Creating a virtual environment with virtualenv and virtualenvwrapper

Virtual environments provide dependency isolation for small projects. They also keep your `site-packages` directory small. Since Python 3.3, `virtualenv` has been part of the standard Python distribution. The `virtualenvwrapper` Python project has some extra convenient features for virtual environment management. I will demonstrate `virtualenv` and `virtualenvwrapper` functionality in this recipe.

Getting ready

You need Python 3.3 or later. You can install `virtualenvwrapper` with `pip` command as follows:

```
$ [sudo] pip install virtualenvwrapper
```

On Linux and Mac, it's necessary to do some extra work—specifying a directory for the virtual environments and sourcing a script:

```
$ export WORKON_HOME=/tmp/envs
$ source /usr/local/bin/virtualenvwrapper.sh
```

Windows has a separate version, which you can install with the following command:

```
$ pip install virtualenvwrapper-win
```

How to do it...

1. Create a virtual environment for a given directory with the `pyvenv` script part of your Python distribution:

```
$ pyvenv /tmp/testenv
$ ls
bin      include      lib      pyvenv.cfg
```

2. In this example, we created a `testenv` directory in the `/tmp` directory with several directories and a configuration file. The configuration file `pyvenv.cfg` contains the Python version and the home directory of the Python distribution.
3. Activate the environment on Linux or Mac by sourcing the `activate` script, for example, with the following command:

```
$ source bin/activate
```

On Windows, use the `activate.bat` file.

4. You can now install packages in this environment in isolation. When you are done with the environment, switch back on Linux or Mac with the following command:

```
$ deactivate
```

On Windows, use the `deactivate.bat` file.

5. Alternatively, you could use `virtualenvwrapper`. Create and switch to a virtual environment with the following command:

```
vagrant@data-science-toolbox:~$ mkvirtualenv env2
```

6. Deactivate the environment with the deactivate command:

```
(env2)vagrant@data-science-toolbox:~$ deactivate
```

7. Delete the environment with the rmvirtualenv command:

```
vagrant@data-science-toolbox:~$ rmvirtualenv env2
```

See also

- ▶ The Python standard library documentation for virtual environments at <https://docs.python.org/3/library/venv.html#creating-virtual-environments> (retrieved July 2015)
- ▶ The virtualenvwrapper documentation is at <https://virtualenvwrapper.readthedocs.org/en/latest/index.html> (retrieved July 2015)

Sandboxing Python applications with Docker images

Docker uses Linux kernel features to provide an extra virtualization layer. Docker was created in 2013 by Solomon Hykes. **Boot2Docker** allows us to install Docker on Windows and Mac OS X too. Boot2Docker uses a VirtualBox VM that contains a Linux environment with Docker. In this recipe, we will set up Docker and download the continuumio/miniconda3 Docker image.

Getting ready

The Docker installation docs are saved at <https://docs.docker.com/index.html> (retrieved July 2015). I installed Docker 1.7.0 with Boot2Docker. The installer requires about 133 MB. However, if you want to follow the whole recipe, you will need several gigabytes.

How to do it...

1. Once Boot2Docker is installed, you need to initialize the environment. This is only necessary once, and Linux users don't need this step:

```
$ boot2docker init
Latest release for github.com/boot2docker/boot2docker is v1.7.0
Downloading boot2docker ISO image...
Success: downloaded https://github.com/boot2docker/boot2docker/
releases/download/v1.7.0/boot2docker.iso
```

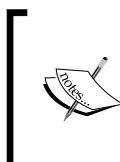
2. In the preceding step, you downloaded a VirtualBox VM to a directory such as `/VirtualBox\ VMs/boot2docker-vm/`.

The next step for Mac OS X and Windows users is to start the VM:

```
$ boot2docker start
```

3. Check the Docker environment by starting a sample container:

```
$ docker run hello-world
```



Some people reported a hopefully temporary issue of not being able to connect. The issue can be resolved by issuing commands with an extra argument, for instance:

```
$ docker [--tlsverify=false] run hello-world
```

4. Docker images can be made public. We can search for such images and download them. In *Setting up Anaconda*, we installed Anaconda; however, Anaconda and Miniconda Docker images also exist. Use the following command:

```
$ docker search continuumio
```

5. The preceding command shows a list of Docker images from **Continuum Analytics** – the company that developed Anaconda and Miniconda. Download the Miniconda 3 Docker image as follows (if you prefer using my container, skip this):

```
$ docker pull continuumio/miniconda3
```

6. Start the image with the following command:

```
$ docker run -t -i continuumio/miniconda3 /bin/bash
```

We start out as root in the image.

7. The command `$ docker images` should list the `continuumio/miniconda3` image as well. If you prefer not to install too much software (possibly only Docker and Boot2Docker) for this book, you should use the image I created. It uses the `continuumio/miniconda3` image as template. This image allows you to execute Python scripts in the current working directory on your computer, while using installed software from the Docker image:

```
$ docker run -it -p 8888:8888 -v $(pwd) :/usr/data -w /usr/data "ivanidris/pydacbk:latest" python <somefile>.py
```

8. You can also run a IPython notebook in your current working directory with the following command:

```
$ docker run -it -p 8888:8888 -v $(pwd) :/usr/data -w /usr/data "ivanidris/pydacbk:latest" sh -c "ipython notebook --ip=0.0.0.0 --no-browser"
```

9. Then, go to either `http:// 192.168.59.103:8888` or `http:// localhost :8888` to view the IPython home screen. You might have noticed that the command lines are quite long, so I will post additional tips and tricks to make life easier on <https://pythonhosted.org/dautil> (work in progress).

The Boot2Docker VM shares the `/Users` directory on Mac OS X and the `C:\Users` directory on Windows. In general and on other operating systems, we can mount directories and copy files from the container as described in <https://docs.docker.com/userguide/dockervolumes/> (retrieved July 2015).

10. Shut down the VM (unless you are on Linux, where you use the `docker` command instead) with the following command:

```
$ boot2docker down
```

How it works...

Docker Hub acts as a central registry for public and private Docker images. In this recipe, we downloaded images via this registry. To push an image to Docker Hub, we need to create a local registry first. The way Docker Hub works is in many ways comparable to the way source code repositories such as GitHub work. You can commit changes as well as push, pull, and tag images. The `continuumio/miniconda3` image is configured with a special file, which you can find at <https://github.com/ContinuumIO/docker-images/blob/master/miniconda3/Dockerfile> (retrieved July 2015). In this file, you can read which image was used as base, the name of the maintainer, and the commands used to build the image.

See also

- ▶ The Docker user guide at [http://docs.docker.com/userguide/](https://docs.docker.com/userguide/) (retrieved July 2015)

Keeping track of package versions and history in IPython Notebook

The **IPython Notebook** was added to IPython 0.12 in December 2011. Many Pythonistas feel that the IPython Notebook is essential for reproducible data analysis. The IPython Notebook is comparable to commercial products such as Mathematica, MATLAB, and Maple. It is an interactive web browser-based environment. In this recipe, we will see how to keep track of package versions and store IPython sessions in the context of reproducible data analysis. By the way, the IPython Notebook has been renamed Jupyter Notebook.

Getting ready

For this recipe, you will need a recent IPython installation. The instructions to install IPython are at <http://ipython.org/install.html> (retrieved July 2015). Install it using the pip command:

```
$ [sudo] pip install ipython/jupyter
```

If you have installed IPython via Anaconda already, check for updates with the following commands:

```
$ conda update conda
$ conda update ipython ipython-notebook ipython-qtconsole
```

I have IPython 3.2.0 as part of the Anaconda distribution.

How to do it...

We will install log a Python session and use the **watermark** extension to track package versions and other information. Start an IPython shell or notebook. When we start a session, we can use the command line switch `--logfile=<file name>.py`. In this recipe, we use the `%logstart` magic (IPython terminology) function:

```
In [1]: %logstart cookbook_log.py rotate
Activating auto-logging. Current session state plus future input saved.
Filename      : cookbook_log.py
Mode          : rotate
Output logging : False
Raw input log  : False
Timestamping   : False
State         : active
```

This example invocation started logging to a file in rotate mode. Both the filename and mode are optional. Turn logging off and back on again as follows:

```
In [2]: %logoff
Switching logging OFF
```

```
In [3]: %logon
Switching logging ON
```

Install the watermark magic from Github with the following command:

```
In [4]: %install_ext https://raw.githubusercontent.com/rasbt/watermark/master/watermark.py
```

The preceding line downloads a Python file, in my case, to `~/.ipython/extensions/watermark.py`. Load the extension by typing the following line:

```
%load_ext watermark
```

The extension can place timestamps as well as software and hardware information. Get additional usage documentation and version (I installed watermark 1.2.2) with the following command:

```
%watermark?
```

For example, call watermark without any arguments:

```
In [7]: %watermark  
... Omitting time stamp ...
```

```
CPython 3.4.3
```

```
IPython 3.2.0
```

```
compiler    : Omitting  
system      : Omitting  
release     : 14.3.0  
machine     : x86_64  
processor   : i386  
CPU cores   : 8  
interpreter: 64bit
```

I omitted the timestamp and other information for personal reasons. A more complete example follows with author name (-a), versions of packages specified as a comma-separated string (-p), and custom time (-c) in a `strftime()` based format:

```
In [8]: %watermark -a "Ivan Idris" -v -p numpy,scipy,matplotlib -c '%b
```

```
%Y' -w
```

```
Ivan Idris 'Jul 2015'
```

```
CPython 3.4.3
```

```
IPython 3.2.0
```

```
numpy 1.9.2
```

```
scipy 0.15.1
matplotlib 1.4.3
watermark v. 1.2.2
```

How it works...

The IPython logger writes commands you type to a Python file. Most of the lines are in the following format:

```
get_ipython().magic('STRING_YOU_TYPED')
```

You can replay the session with `%load <log_file>`. The logging modes are described in the following table:

Mode	Description
over	This mode overwrites existing log files.
backup	If a log file exists with the same name, the old file is renamed.
append	This mode appends lines to already existing files.
rotate	This mode rotates log files by incrementing numbers, so that log files don't get too big.

We used a custom magic function available on the Internet. The code for the function is in a single Python file and it should be easy for you to follow. If you want different behavior, you just need to modify the file.

See also

- ▶ The custom magics documentation at <http://ipython.org/ipython-doc/dev/config/custommagics.html> (retrieved July 2015)
- ▶ Helen Shen (2014). *Interactive notebooks: Sharing the code*. Nature 515 (7525): 151–152. doi:10.1038/515151a
- ▶ IPython reference documentation at <https://ipython.org/ipython-doc/dev/interactive/reference.html> (retrieved July 2015)

Configuring IPython

IPython has an elaborate configuration and customization system. The components of the system are as follows:

- ▶ IPython provides default profiles, but we can create our own profiles
- ▶ Various settable options for the shell, kernel, Qt console, and notebook

- ▶ Customization of prompts and colors
- ▶ Extensions we saw in *Keeping track of package versions and history in IPython notebooks*
- ▶ Startup files per profile

I will demonstrate some of these components in this recipe.

Getting ready

You need IPython for this recipe, so (if necessary) have a look at the *Getting ready* section of *Keeping track of package versions and history in IPython notebooks*.

How to do it...

Let's start with a startup file. I have a directory in my home directory at `.ipython/profile_default/startup`, which belongs to the default profile. This directory is meant for startup files. IPython detects Python files in this directory and executes them in lexical order of filenames. Because of the lexical order, it is convenient to name the startup files with a combination of digits and strings, for example, `0000-watermark.py`. Put the following code in the startup file:

```
get_ipython().magic('%load_ext watermark')
get_ipython().magic('watermark -a "Ivan Idris" -v -p
numpy,scipy,matplotlib -c \'%b %Y\' -w')
```

This startup file loads the extension we used in *Keeping track of package versions and history in IPython notebooks* and shows information about package versions. Other use cases include importing modules and defining functions. IPython stores commands in a SQLite database, so you could gather statistics to find common usage patterns. The following script prints source lines and associated counts from the database for the default profile sorted by counts (the code is in the `ipython_history.py` file in this book's code bundle):

```
import sqlite3
from IPython.utils.path import get_ipython_dir
import pprint
import os

def print_history(file):
    with sqlite3.connect(file) as con:
        c = con.cursor()
        c.execute("SELECT count(source_raw) as csr,\n            source_raw FROM history\n            GROUP BY source_raw\n            ORDER BY csr")
        result = c.fetchall()
```

```

pprint.pprint(result)
c.close()

hist_file = '%s/profile_default/history.sqlite' % get_ipython_dir()

if os.path.exists(hist_file):
    print_history(hist_file)
else:
    print("%s doesn't exist" % hist_file)

```

The highlighted SQL query does the bulk of the work. The code is self-explanatory. If it is not clear, I recommend reading *Chapter 8, Text Mining and Social Network*, of my book *Python Data Analysis*, Packt Publishing.

The other configuration option I mentioned is profiles. We can use the default profiles or create our own profiles on a per project or functionality basis. Profiles act as sandboxes and you can configure them separately. Here's the command to create a profile:

```
$ ipython profile create [newprofile]
```

The configuration files are Python files and their names end with `_config.py`. In these files, you can set various IPython options. Set the option to automatically log the IPython session as follows:

```

c = get_config()

c.TerminalInteractiveShell.logstart=True

```

The first line is usually included in configuration files and gets the root IPython configuration object. The last line tells IPython that we want to start logging immediately on startup so you don't have to type `%logstart`.

Alternatively, you can also set the log file name with the following command:

```
c.TerminalInteractiveShell.logfile='mylog_file.py'
```

You can also use the following configuration line that ensures logging in append mode:

```
c.TerminalInteractiveShell.logappend='mylog_file.py'
```

See also

- ▶ *Introduction to IPython configuration* at <http://ipython.org/ipython-doc/dev/config/intro.html#profiles> (retrieved July 2015)
- ▶ *Terminal IPython options documentation* at <http://ipython.org/ipython-doc/dev/config/options/terminal.html> (retrieved July 2015)

Learning to log for robust error checking

Notebooks are useful to keep track of what you did and what went wrong. Logging works in a similar fashion, and we can log errors and other useful information with the standard Python logging library.

For reproducible data analysis, it is good to know the modules our Python scripts import. In this recipe, I will introduce a minimal API from `dautil` that logs package versions of imported modules in a best effort manner.

Getting ready

In this recipe, we import NumPy and pandas, so you may need to import them. See the *Configuring pandas* recipe for pandas installation instructions. Installation instructions for NumPy can be found at <http://docs.scipy.org/doc/numpy/user/install.html> (retrieved July 2015). Alternatively, install NumPy with pip using the following command:

```
$ [sudo] pip install numpy
```

The command for Anaconda users is as follows:

```
$ conda install numpy
```

I have installed NumPy 1.9.2 via Anaconda. We also require `AppDirs` to find the appropriate directory to store logs. Install it with the following command:

```
$ [sudo] pip install appdirs
```

I have `AppDirs` 1.4.0 on my system.

How to do it...

To log, we need to create and set up loggers. We can either set up the loggers with code or use a configuration file. Configuring loggers with code is the more flexible option, but configuration files tend to be more readable. I use the `log.conf` configuration file from `dautil`:

```
[loggers]
keys=root

[handlers]
keys=consoleHandler,fileHandler

[formatters]
keys=simpleFormatter
```

```
[logger_root]
level=DEBUG
handlers=consoleHandler,fileHandler

[handler_consoleHandler]
class=StreamHandler
level=INFO
formatter=simpleFormatter
args=(sys.stdout,)

[handler_fileHandler]
class=dautil.log_api.VersionsLogFileHandler
formatter=simpleFormatter
args=('versions.log',)

[formatter_simpleFormatter]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
datefmt=%d-%b-%Y
```

The file configures a logger to log to a file with the DEBUG level and to the screen with the INFO level. So, the logger logs more to the file than to the screen. The file also specifies the format of the log messages. I created a tiny API in dautil, which creates a logger with its `get_logger()` function and uses it to log the package versions of a client program with its `log()` function. The code is in the `log_api.py` file of dautil:

```
from pkg_resources import get_distribution
from pkg_resources import resource_filename
import logging
import logging.config
import pprint
from appdirs import AppDirs
import os

def get_logger(name):
    log_config = resource_filename(__name__, 'log.conf')
    logging.config.fileConfig(log_config)
    logger = logging.getLogger(name)

    return logger

def shorten(module_name):
```

```
dot_i = module_name.find('.')

return module_name[:dot_i]

def log(modules, name):
    skiplist = ['pkg_resources', 'distutils']

    logger = get_logger(name)
    logger.debug('Inside the log function')

    for k in modules.keys():
        str_k = str(k)

        if '.version' in str_k:
            short = shorten(str_k)

            if short in skiplist:
                continue

            try:
                logger.info('%s=%s' % (short,
                                      get_distribution(short).version))
            except ImportError:
                logger.warn('Could not import', short)

class VersionsLogFileHandler(logging.FileHandler):
    def __init__(self, fName):
        dirs = AppDirs("PythonDataAnalysisCookbook",
                      "Ivan Idris")
        path = dirs.user_log_dir
        print(path)

        if not os.path.exists(path):
            os.mkdir(path)

        super(VersionsLogFileHandler, self).__init__(
            os.path.join(path, fName))
```

The program that uses the API is in the `log_demo.py` file in this book's code bundle:

```
import sys
import numpy as np
```

```
import matplotlib.pyplot as plt
import pandas as pd
from dautil import log_api

log_api.log(sys.modules, sys.argv[0])
```

How it works...

We configured a handler (`VersionsLogFileHandler`) that writes to file and a handler (`StreamHandler`) that displays messages on the screen. `StreamHandler` is a class in the Python standard library. To configure the format of the log messages, we used the `SimpleFormatter` class from the Python standard library.

The API I made goes through modules listed in the `sys.modules` variable and tries to get the versions of the modules. Some of the modules are not relevant for data analysis, so we skip them. The `log()` function of the API logs a `DEBUG` level message with the `debug()` method. The `info()` method logs the package version at `INFO` level.

See also

- ▶ The logging tutorial at <https://docs.python.org/3.5/howto/logging.html> (retrieved July 2015)
- ▶ The logging cookbook at <https://docs.python.org/3.5/howto/logging-cookbook.html#logging-cookbook> (retrieved July 2015)

Unit testing your code

If code doesn't do what you want, it's hard to do reproducible data analysis. One way to gain control of your code is to test it. If you have tested code manually, you know it is repetitive and boring. When a task is boring and repetitive, you should automate it.

Unit testing automates testing and I hope you are familiar with it. When you learn unit testing for the first time, you start with simple tests such as comparing strings or numbers. However, you hit a wall when file I/O or other resources come into the picture. It turns out that in Python we can mock resources or external APIs easily. The packages needed are even part of the standard Python library. In the *Learning to log for robust error checking* recipe, we logged messages to a file. If we unit test this code, we don't want to trigger logging from the test code. In this recipe, I will show you how to mock the logger and other software components we need.

Getting ready

Familiarize yourself with the code under test in `log_api.py`.

How to do it...

The code for this recipe is in the `test_log_api.py` file of `dautil`. We start by importing the module under test and the Python functionality we need for unit testing:

```
from dautil import log_api
import unittest
from unittest.mock import create_autospec
from unittest.mock import patch
```

Define a class that contains the test code:

```
class TestLogApi(unittest.TestCase):
```

Make the unit tests executable with the following lines:

```
if __name__ == '__main__':
    unittest.main()
```

If we call Python functions with the wrong number of arguments, we expect to get a `TypeError`. The following tests check for that:

```
def test_get_logger_args(self):
    mock_get_logger = create_autospec(log_api.get_logger,
return_value=None)
    mock_get_logger('test')
    mock_get_logger.assert_called_once_with('test')

def test_log_args(self):
    mock_log = create_autospec(log_api.log, return_value=None)
    mock_log([], 'test')
    mock_log.assert_called_once_with([], 'test')

    with self.assertRaises(TypeError):
        mock_log()

    with self.assertRaises(TypeError):
        mock_log('test')
```

We used the `unittest.create_autospec()` function to mock the functions under test. Mock the Python logging package as follows:

```
@patch('dautil.log_api.logging')
def test_get_logger_fileConfig(self, mock_logging):
    log_api.get_logger('test')
    self.assertTrue(mock_logging.config.fileConfig.called)
```

The `@patch` decorator replaces logging with a mock. We can also patch with similarly named functions. The patching trick is quite useful. Test our `get_logger()` function with the following method:

```
@patch('dautil.log_api.get_logger')
def test_log_debug(self, amock):
    log_api.log({}, 'test')
    self.assertTrue(amock.return_value.debug.called)
    amock.return_value.debug.assert_called_once_with(
        'Inside the log function')
```

The previous lines check whether `debug()` was called and with which arguments. The following two test methods demonstrate how to use multiple `@patch` decorators:

```
@patch('dautil.log_api.get_distribution')
@patch('dautil.log_api.get_logger')
def test_numpy(self, m_get_logger, m_get_distribution):
    log_api.log({'numpy.version': ''}, 'test')
    m_get_distribution.assert_called_once_with('numpy')
    self.assertTrue(m_get_logger.return_value.info.called)

@patch('dautil.log_api.get_distribution')
@patch('dautil.log_api.get_logger')
def test_distutils(self, amock, m_get_distribution):
    log_api.log({'distutils.version': ''}, 'test')
    self.assertFalse(m_get_distribution.called)
```

How it works...

Mocking is a technique to spy on objects and functions. We substitute them with our own spies, which we give just enough information to avoid detection. The spies report to us who contacted them and any useful information they received.

See also

- ▶ The `unittest.mock` library documentation at <https://docs.python.org/3/library/unittest.mock.html#patch-object> (retrieved July 2015)
- ▶ The `unittest` documentation at <https://docs.python.org/3/library/unittest.html> (retrieved July 2015)

Configuring pandas

The pandas library has more than a dozen configuration options, as described in <http://pandas.pydata.org/pandas-docs/dev/options.html> (retrieved July 2015).



The pandas library is Python open source software originally created for econometric data analysis. It uses data structures inspired by the R programming language.

You can set and get properties using dotted notation or via functions. It is also possible to reset options to defaults and get information about them. The `option_context()` function allows you to limit the scope of the option to a context using the `with` statement. In this recipe, I will demonstrate pandas configuration and a simple API to set and reset options I find useful. The two options are `precision` and `max_rows`. The first option specifies floating point precision of output. The second option specifies the maximum rows of a pandas `DataFrame` to print on the screen.

Getting ready

You need pandas and NumPy for this recipe. Instructions to install NumPy are given in *Learning to log for robust error checking*. The pandas installation documentation can be found at <http://pandas.pydata.org/pandas-docs/dev/install.html> (retrieved July 2015). The recommended way to install pandas is via Anaconda. I have installed pandas 0.16.2 via Anaconda. You can update your Anaconda pandas with the following command:

```
$ conda update pandas
```

How to do it...

The following code from the `options.py` file in `dautil` defines a simple API to set and reset options:

```
import pandas as pd

def set_pd_options():
    pd.set_option('precision', 4)
    pd.set_option('max_rows', 5)

def reset_pd_options():
    pd.reset_option('precision')
    pd.reset_option('max_rows')
```

The script in `configure_pd.py` in this book's code bundle uses the following API:

```
from dautil import options
import pandas as pd
import numpy as np
from dautil import log_api

printer = log_api.Printer()
print(pd.describe_option('precision'))
print(pd.describe_option('max_rows'))

printer.print('Initial precision', pd.get_option('precision'))
printer.print('Initial max_rows', pd.get_option('max_rows'))

# Random pi's, should use random state if possible
np.random.seed(42)
df = pd.DataFrame(np.pi * np.random.rand(6, 2))
printer.print('Initial df', df)

options.set_pd_options()
printer.print('df with different options', df)

options.reset_pd_options()
printer.print('df after reset', df)
```

If you run the script, you get descriptions for the options that are a bit too long to display here. The getter gives the following output:

```
'Initial precision'
```

```
7
```

```
'Initial max_rows'
```

```
60
```

Then, we create a pandas DataFrame table with random data. The initial printout looks like this:

```
'Initial df'
```

	0	1
0	1.176652	2.986757
1	2.299627	1.880741
2	0.490147	0.490071
3	0.182475	2.721173
4	1.888459	2.224476
5	0.064668	3.047062

The printout comes from the following class in `log_api.py`:

```
class Printer():
    def __init__(self, modules=None, name=None):
        if modules and name:
            log(modules, name)

    def print(self, *args):
        for arg in args:
            pprint.pprint(arg)
```

After setting the options with the `dautil` API, pandas hides some of the rows and the floating point numbers look different too:

```
'df with different options'
```

```
      0      1
0  1.177  2.987
1  2.300  1.881
...
4  1.888  2.224
5  0.065  3.047
```

```
[6 rows x 2 columns]
```

Because of the truncated rows, pandas tells us how many rows and columns the `DataFrame` table has. After we reset the options, we get the original printout back.

Configuring matplotlib

The `matplotlib` library allows configuration via the `matplotlibrc` files and Python code. The last option is what we are going to do in this recipe. Small configuration tweaks should not matter if your data analysis is strong. However, it doesn't hurt to have consistent and attractive plots. Another option is to apply stylesheets, which are files comparable to the `matplotlibrc` files. However, in my opinion, the best option is to use Seaborn on top of `matplotlib`. I will discuss Seaborn and `matplotlib` in more detail in *Chapter 2, Creating Attractive Data Visualizations*.

Getting ready

You need to install matplotlib for this recipe. Visit <http://matplotlib.org/users/installing.html> (retrieved July 2015) for more information. I have matplotlib 1.4.3 via Anaconda. Install Seaborn using Anaconda:

```
$ conda install seaborn
```

I have installed Seaborn 0.6.0 via Anaconda.

How to do it...

We can set options via a dictionary-like variable. The following function from the `options.py` file in `dautil` sets three options:

```
def set_mpl_options():
    mpl.rcParams['legend.fancybox'] = True
    mpl.rcParams['legend.shadow'] = True
    mpl.rcParams['legend.framealpha'] = 0.7
```

The first three options have to do with legends. The first option specifies rounded corners for the legend, the second option enables showing a shadow, and the third option makes the legend slightly transparent. The `matplotlib rcdefaults()` function resets the configuration.

To demonstrate these options, let's use sample data made available by matplotlib. The imports are as follows:

```
import matplotlib.cbook as cbook
import pandas as pd
import matplotlib.pyplot as plt
from dautil import options
import matplotlib as mpl
from dautil import plotting
import seaborn as sns
```

The data is in a CSV file and contains stock price data for AAPL. Use the following commands to read the data and stores them in a pandas `DataFrame`:

```
data = cbook.get_sample_data('aapl.csv', asfileobj=True)
df = pd.read_csv(data, parse_dates=True, index_col=0)
```

Resample the data to average monthly values as follows:

```
df = df.resample('M')
```

The full code is in the `configure_matplotlib.ipynb` file in this book's code bundle:

```
import matplotlib.cbook as cbook
import pandas as pd
import matplotlib.pyplot as plt
from dautil import options
import matplotlib as mpl
from dautil import plotting
import seaborn as sns

data = cbook.get_sample_data('aapl.csv', asfileobj=True)
df = pd.read_csv(data, parse_dates=True, index_col=0)
df = df.resample('M')
close = df['Close'].values
dates = df.index.values
fig, axes = plt.subplots(4)

def plot(title, ax):
    ax.set_title(title)
    ax.set_xlabel('Date')

    plotter = plotting.CyclePlotter(ax)
    plotter.plot(dates, close, label='Close')
    plotter.plot(dates, 0.75 * close, label='0.75 * Close')
    plotter.plot(dates, 1.25 * close, label='1.25 * Close')

    ax.set_ylabel('Price ($)')
    ax.legend(loc='best')

plot('Initial', axes[0])
sns.reset_orig()
options.set_mpl_options()

plot('After setting options', axes[1])

sns.reset_defaults()
plot('After resetting options', axes[2])

with plt.style.context(('dark_background')):
    plot('With dark_background stylesheet', axes[3])
    fig.autofmt_xdate()
plt.show()
```

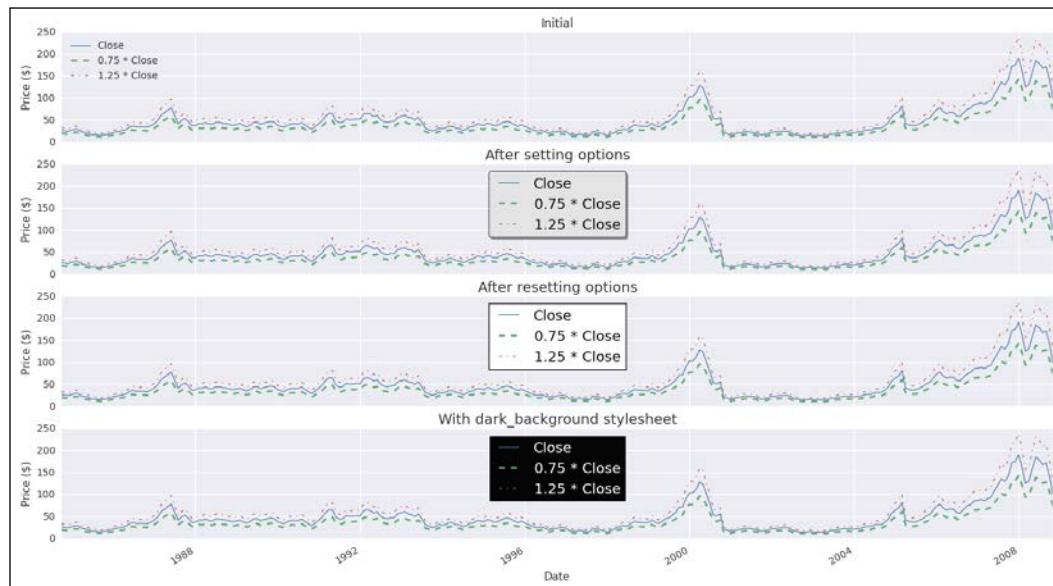
The program plots the data and arbitrary upper and lower band with the default options, custom options, and after a reset of the options. I used the following helper class from the `plotting.py` file of `dautil`:

```
from itertools import cycle

class CyclePlotter():
    def __init__(self, ax):
        self.STYLES = cycle(["-", "--", "-.", ":"])
        self.LW = cycle([1, 2])
        self.ax = ax

    def plot(self, x, y, *args, **kwargs):
        self.ax.plot(x, y, next(self.STYLES),
                     lw=next(self.LW), *args, **kwargs)
```

The class cycles through different line styles and line widths. Refer to the following plot for the end result:



How it works...

Importing Seaborn dramatically changes the look and feel of matplotlib plots. Just temporarily comment the `seaborn` lines out to convince yourself. However, Seaborn doesn't seem to play nicely with the matplotlib options we set, unless we use the Seaborn functions `reset_orig()` and `reset_defaults()`.

See also

- ▶ The matplotlib customization documentation at <http://matplotlib.org/users/customizing.html> (retrieved July 2015)
- ▶ The matplotlib documentation about stylesheets at http://matplotlib.org/users/style_sheets.html (retrieved July 2015)

Seeding random number generators and NumPy print options

For reproducible data analysis, we should prefer deterministic algorithms. Some algorithms use random numbers, but in practice we rarely use perfectly random numbers. The algorithms provided in `numpy.random` allow us to specify a seed value. For reproducibility, it is important to always provide a seed value but it is easy to forget. A utility function in `sklearn.utils` provides a solution for this issue.

NumPy has a `set_printoptions()` function, which controls how NumPy prints arrays. Obviously, printing should not influence the quality of your analysis too much. However, readability is important if you want people to understand and reproduce your results.

Getting ready

Install NumPy using the instructions in the *Learning to log for robust error checking* recipe. We will need scikit-learn, so have a look at <http://scikit-learn.org/dev/install.html> (retrieved July 2015). I have installed scikit-learn 0.16.1 via Anaconda.

How to do it...

The code for this example is in the `configure_numpy.py` file in this book's code bundle:

```
from sklearn.utils import check_random_state
import numpy as np
from dautil import options
```

```

from dautil import log_api

random_state = check_random_state(42)
a = random_state.rndn(5)

random_state = check_random_state(42)
b = random_state.rndn(5)

np.testing.assert_array_equal(a, b)

printer = log_api.Printer()
printer.print("Default options", np.get_printoptions())

pi_array = np.pi * np.ones(30)
options.set_np_options()
print(pi_array)

# Reset
options.reset_np_options()
print(pi_array)

```

The highlighted lines show how to get a NumPy RandomState object with 42 as the seed. In this example, the arrays a and b are equal, because we used the same seed and the same procedure to draw the numbers. The second part of the preceding program uses the following functions I defined in `options.py`:

```

def set_np_options():
    np.set_printoptions(precision=4, threshold=5,
                        linewidth=65)

def reset_np_options():
    np.set_printoptions(precision=8, threshold=1000,
                        linewidth=75)

```

Here's the output after setting the options:

```
[ 3.1416  3.1416  3.1416 ...,  3.1416  3.1416  3.1416]
```

As you can see, NumPy replaces some of the values with an ellipsis and it shows only four digits after the decimal sign. The NumPy defaults are as follows:

```
'Default options'
{'edgeitems': 3,
 'formatter': None,
```

```
'infstr': 'inf',
'linewidth': 75,
'nansstr': 'nan',
'precision': 8,
'suppress': False,
'threshold': 1000}
```

See also

- ▶ The scikit-learn documentation at <http://scikit-learn.org/stable/developers/utilities.html> (retrieved July 2015)
- ▶ The NumPy `set_printoptions()` documentation at http://docs.scipy.org/doc/numpy/reference/generated/numpy.set_printoptions.html (retrieved July 2015)
- ▶ The NumPy RandomState documentation at <http://docs.scipy.org/doc/numpy/reference/generated/numpy.random.RandomState.html> (retrieved July 2015)

Standardizing reports, code style, and data access

Following a code style guide helps improve code quality. Having high-quality code is important if you want people to easily reproduce your analysis. One way to adhere to a coding standard is to scan your code with static code analyzers. You can use many code analyzers. In this recipe, we will use the `pep8` analyzer. In general, code analyzers complement or maybe slightly overlap each other, so you are not limited to `pep8`.

Convenient data access is crucial for reproducible analysis. In my opinion, the best type of data access is with a specialized API and local data. I will introduce a `dautil` module I created to load weather data provided by the Dutch KNMI.

Reporting is often the last phase of a data analysis project. We can report our findings using various formats. In this recipe, we will focus on tabulating our report with the `tabulate` module. The `landslide` tool creates slide shows from various formats such as reStructured text.

Getting ready

You will need pep8 and tabulate. A quick guide to pep8 is available at <https://pep8.readthedocs.org/en/latest/intro.html> (retrieved July 2015). I have installed pep8 1.6.2 via Anaconda. You can install joblib, tabulate, and landslide with the pip command.

I have tabulate 0.7.5 and landslide 1.1.3.

How to do it...

Here's an example pep8 session:

```
$ pep8 --first log_api.py
log_api.py:21:1: E302 expected 2 blank lines, found 1
log_api.py:44:33: W291 trailing whitespace
log_api.py:50:60: E225 missing whitespace around operator
```

The --first switch finds the first occurrence of an error. In the previous example, pep8 reports the line number where the error occurred, an error code, and a short description of the error. I prepared a module dedicated to data access of datasets we will use in several chapters. We start with access to a pandas DataFrame stored in a pickle, which contains selected weather data from the De Bilt weather station in the Netherlands. I created the pickle by downloading a zip file, extracting the data file, and loading the data in a pandas DataFrame table. I applied minimal data transformation, including multiplication of values and converting empty fields to NaNs. The code is in the data.py file in dautil. I will not discuss this code in detail, because we only need to load data from the pickle. However, if you want to download the data yourself, you can use the static method I defined in data.py. Downloading the data will of course give you more recent data, but you will get slightly different results if you substitute my pickle. The following code shows the basic descriptive statistics with the pandas.DataFrame.describe() method in the report_weather.py file in this book's code bundle:

```
from dautil import data
from dautil import report
import pandas as pd
import numpy as np
from tabulate import tabulate

df = data.Weather.load()
headers = [data.Weather.get_header(header)
           for header in df.columns.values.tolist()]
df = df.describe()
```

Then, the code creates a `slides.rst` file in the reStructuredText format with `dautil.RSTWriter`. This is just a matter of simple string concatenation and writing to a file. The highlighted lines in the following code show the `tabulate()` calls that create table grids from the `pandas.DataFrame` objects:

```

writer = report.RSTWriter()
writer.h1('Weather Statistics')
writer.add(tabulate(df, headers=headers,
                    tablefmt='grid', floatfmt='.2f'))
writer.divider()
headers = [data.Weather.get_header(header)
           for header in df.columns.values.tolist()]
builder = report.DFBuilder(df.columns)
builder.row(df.iloc[7].values - df.iloc[3].values)
builder.row(df.iloc[6].values - df.iloc[4].values)
df = builder.build(['ptp', 'iqr'])

writer.h1('Peak-to-peak and Interquartile Range')

writer.add(tabulate(df, headers=headers,
                    tablefmt='grid', floatfmt='.2f'))
writer.write('slides.rst')
generator = report.Generator('slides.rst', 'weather_report.html')
generator.generate()

```

I use the `dautil.reportDFBuilder` class to create the `pandas.DataFrame` objects incrementally using a dictionary where the keys are columns of the final `DataFrame` table and the values are the rows:

```

import pandas as pd

class DFBuilder():
    def __init__(self, cols, *args):
        self.columns = cols
        self.df = {}

        for col in self.columns:
            self.df.update({col: []})

        for arg in args:
            self.row(arg)

    def row(self, row):
        assert len(row) == len(self.columns)

        for col, val in zip(self.columns, row):

```

```

        self.df[col].append(val)

    return self.df

def build(self, index=None):
    self.df = pd.DataFrame(self.df)

    if index:
        self.df.index = index

    return self.df

```

I eventually generate a HTML file using `landslide` and my own custom CSS. If you open `weather_report.html`, you will see the first slide with basic descriptive statistics:

Weather Statistics					
	Wind Dir	W Speed, m/s	Temp, °C	Rain, mm	Pres, hPa
count	37907.00	37907.00	39032.00	37176.00	38667.00
mean	190.02	4.29	9.52	2.34	1014.93
std	93.83	1.94	6.36	4.47	9.90
min	0.00	0.00	-14.90	0.00	962.10
25%	120.00	3.10	5.00	0.00	1008.90
50%	207.00	4.10	9.70	0.20	1015.60
75%	256.00	5.10	14.50	2.80	1021.60
max	360.00	16.50	27.90	63.90	1048.30

The second slide looks like this and contains the peak-to-peak (difference between minimum and maximum values) and the interquartile range (difference between the third and first quartile):

Peak-to-peak And Interquartile Range					
	Pres, hPa	Rain, mm	Temp, °C	Wind Dir	W Speed, m/s
ptp	86.20	63.90	42.80	360.00	16.50
iqr	12.70	2.80	9.50	136.00	2.00

See also

- ▶ The tabulate PyPi page at <https://pypi.python.org/pypi/tabulate> (retrieved July 2015)
- ▶ The landslide Github page at <https://github.com/adamzap/landslide> (retrieved July 2015)

2

Creating Attractive Data Visualizations

In this chapter, we will cover:

- ▶ Graphing Anscombe's quartet
- ▶ Choosing seaborn color palettes
- ▶ Choosing matplotlib color maps
- ▶ Interacting with IPython notebook widgets
- ▶ Viewing a matrix of scatterplots
- ▶ Visualizing with d3.js via mpld3
- ▶ Creating heatmaps
- ▶ Combining box plots and kernel density plots with violin plots
- ▶ Visualizing network graphs with hive plots
- ▶ Displaying geographical maps
- ▶ Using ggplot2-like plots
- ▶ Highlighting data points with influence plots

Introduction

Data analysis is more of an art than a science. Creating attractive visualizations is an integral part of this art. Obviously, what one person finds attractive, other people may find completely unacceptable. Just as in art, in the rapidly evolving world of data analysis, opinions, and taste change over time; however, in principle, nobody is absolutely right or wrong. As data artists and Pythonistas, we can choose from among several libraries of which I will cover matplotlib, seaborn, Bokeh, and ggplot. Installation instructions for some of the packages we use in this chapter were already covered in *Chapter 1, Laying the Foundation for Reproducible Data Analysis*, so I will not repeat them. I will provide an installation script (which uses pip only) for this chapter; you can even use the Docker image I described in the previous chapter. I decided to not include the Proj cartography library and the R-related libraries in the image because of their size. So for the two recipes involved in this chapter, you may have to do extra work.

Graphing Anscombe's quartet

Anscombe's quartet is a classic example that illustrates why visualizing data is important. The quartet consists of four datasets with similar statistical properties. Each dataset has a series of x values and dependent y values. We will tabulate these metrics in an IPython notebook. However, if you plot the datasets, they look surprisingly different compared to each other.

How to do it...

For this recipe, you need to perform the following steps:

1. Start with the following imports:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib as mpl
from dautil import report
from dautil import plotting
import numpy as np
from tabulate import tabulate
```

2. Define the following function to compute the mean, variance, and correlation of x and y within a dataset, the slope, and the intercept of a linear fit for each of the datasets:

```
df = sns.load_dataset("anscombe")

agg = df.groupby('dataset') \
    .agg([np.mean, np.var]) \
    .transpose()
```

```

groups = df.groupby('dataset')

corr = [g.corr()['x'][1] for _, g in groups]
builder = report.DFBuilder(agg.columns)
builder.row(corr)

fits = [np.polyfit(g['x'], g['y'], 1) for _, g in groups]
builder.row([f[0] for f in fits])
builder.row([f[1] for f in fits])
bottom = builder.build(['corr', 'slope', 'intercept'])

return df, pd.concat((agg, bottom))

```

3. The following function returns a string, which is partly Markdown, partly restructured text, and partly HTML, because core Markdown does not officially support tables:

```

def generate(table):
    writer = report.RSTWriter()
    writer.h1('Anscombe Statistics')
    writer.add(tabulate(table, tablefmt='html', floatfmt='.3f'))

    return writer.rst

```

4. Plot the data and corresponding linear fits with the Seaborn `lmplot()` function:

```

def plot(df):
    sns.set(style="ticks")
    g = sns.lmplot(x="x", y="y", col="dataset",
                    hue="dataset", data=df,
                    col_wrap=2, ci=None, palette="muted", size=4,
                    scatter_kws={"s": 50, "alpha": 1})

    plotting.embellish(g.fig.axes)

```

5. Display a table with statistics, as follows:

```

df, table = aggregate()
from IPython.display import display_markdown
display_markdown(generate(table), raw=True)

```

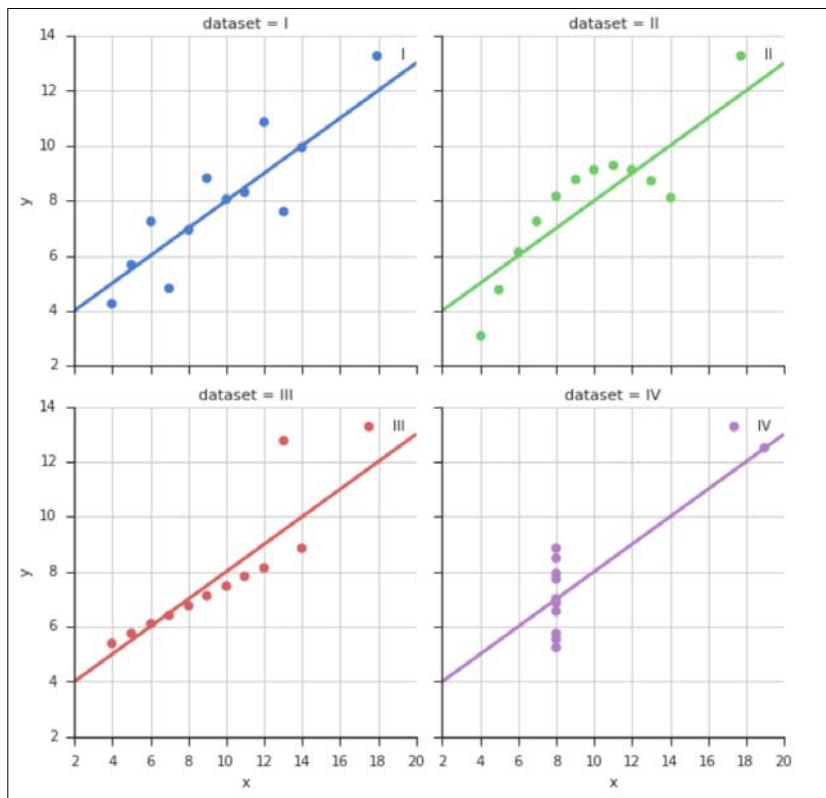
The following table shows practically identical statistics for each dataset (I modified the `custom.css` file in my IPython profile to get the colors):

Anscombe Statistics				
('x', 'mean')	9.000	9.000	9.000	9.000
('x', 'var')	11.000	11.000	11.000	11.000
('y', 'mean')	7.501	7.501	7.500	7.501
('y', 'var')	4.127	4.128	4.123	4.123
corr	0.816	0.816	0.816	0.817
slope	0.500	0.500	0.500	0.500
intercept	3.000	3.001	3.002	3.002

6. The following lines plot the datasets:

```
%matplotlib inline
plot(df)
```

Refer to the following plot for the end result:



A picture says more than a thousand words. The source code is in the `anscombe.ipynb` file in this book's code bundle.

See also

- ▶ The Anscombe's quartet Wikipedia page at https://en.wikipedia.org/wiki/Anscombe%27s_quartet (retrieved July 2015)
- ▶ The seaborn documentation for the `lmplot()` function at <https://web.stanford.edu/~mwaskom/software/seaborn/generated/seaborn.lmplot.html> (retrieved July 2015)

Choosing seaborn color palettes

Seaborn color palettes are similar to matplotlib colormaps. Color can help you discover patterns in data and is an important visualization component. Seaborn has a wide range of color palettes, which I will try to visualize in this recipe.

How to do it...

1. The imports are as follows:

```
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
from dautil import plotting
```

2. Use the following function that helps plot the palettes:

```
def plot_palette(ax, plotter, pal, i, label, ncol=1):
    n = len(pal)
    x = np.linspace(0.0, 1.0, n)
    y = np.arange(n) + i * n
    ax.scatter(x, y, c=x,
               cmap=matplotlib.colors.ListedColormap(list(pal)),
               s=200)
    plotter.plot(x,y, label=label)
    handles, labels = ax.get_legend_handles_labels()
    ax.legend(loc='best', ncol=ncol, fontsize=18)
```

3. **Categorical palettes** are useful for categorical data, for instance, gender or blood type. The following function plots some of the Seaborn categorical palettes:

```
def plot_categorical_palettes(ax):
    palettes = ['deep', 'muted', 'pastel', 'bright', 'dark',
    'colorblind']
```

```
plotter = plotting.CyclePlotter(ax)
ax.set_title('Categorical Palettes')

for i, p in enumerate(palettes):
    pal = sns.color_palette(p)
    plot_palette(ax, plotter, pal, i, p, 4)
```

4. **Circular color systems** usually use **HLS (Hue Lightness Saturation)** instead of **RGB (red green blue)** color spaces. They are useful if you have many categories. The following function plots palettes using HSL systems:

```
def plot_circular_palettes(ax):
    ax.set_title('Circular Palettes')
    plotter = plotting.CyclePlotter(ax)

    pal = sns.color_palette("hls", 6)
    plot_palette(ax, plotter, pal, 0, 'hls')

    sns.hls_palette(6, l=.3, s=.8)
    plot_palette(ax, plotter, pal, 1, 'hls l=.3 s=.8')

    pal = sns.color_palette("husl", 6)
    plot_palette(ax, plotter, pal, 2, 'husl')

    sns.husl_palette(6, l=.3, s=.8)
    plot_palette(ax, plotter, pal, 3, 'husl l=.3 s=.8')
```

5. Seaborn also has palettes, which are based on the online ColorBrewer tool (<http://colorbrewer2.org/>). Plot them as follows:

```
def plot_brewer_palettes(ax):
    ax.set_title('Brewer Palettes')
    plotter = plotting.CyclePlotter(ax)

    pal = sns.color_palette("Paired")
    plot_palette(ax, plotter, pal, 0, 'Paired')

    pal = sns.color_palette("Set2", 6)
    plot_palette(ax, plotter, pal, 1, 'Set2')
```

6. **Sequential palettes** are useful for wide ranging data, for instance, differing by orders of magnitude. Use the following function to plot them:

```
def plot_sequential_palettes(ax):
    ax.set_title('Sequential Palettes')
    plotter = plotting.CyclePlotter(ax)

    pal = sns.color_palette("Blues")
    plot_palette(ax, plotter, pal, 0, 'Blues')
```

```

pal = sns.color_palette("BuGn_r")
plot_palette(ax, plotter, pal, 1, 'BuGn_r')

pal = sns.color_palette("GnBu_d")
plot_palette(ax, plotter, pal, 2, 'GnBu_d')

pal = sns.color_palette("cubehelix", 6)
plot_palette(ax, plotter, pal, 3, 'cubehelix')

```

7. The following lines call the functions we defined:

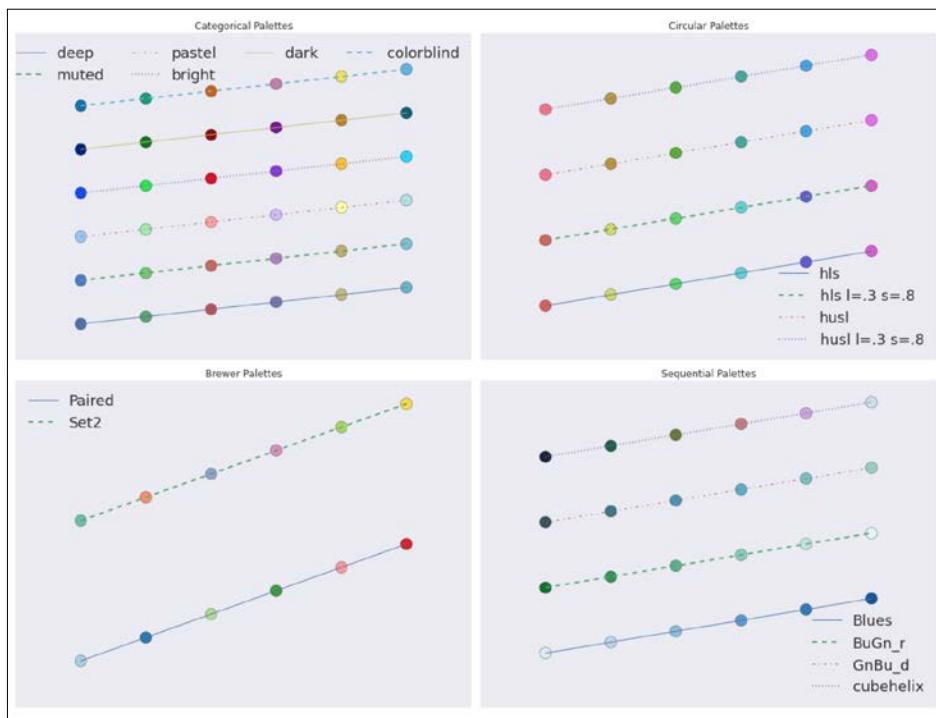
```

%matplotlib inline

fig, axes = plt.subplots(2, 2, figsize=(16, 12))
plot_categorical_palettes(axes[0][0])
plot_circular_palettes(axes[0][1])
plot_brewer_palettes(axes[1][0])
plot_sequential_palettes(axes[1][1])
plotting.hide_axes(axes)
plt.tight_layout()

```

The complete code is available in the choosing_palettes.ipynb file in this book's code bundle. Refer to the following plot for the end result:



See also

- ▶ The seaborn color palettes documentation at https://web.stanford.edu/~mwaskom/software/seaborn/tutorial/color_palettes.html (retrieved July 2015)

Choosing matplotlib color maps

The matplotlib color maps are getting a lot of criticism lately because they can be misleading; however, most colormaps are just fine in my opinion. The defaults are getting a makeover in matplotlib 2.0 as announced at http://matplotlib.org/style_changes.html (retrieved July 2015). Of course, there are some good arguments that do not support using certain matplotlib colormaps, such as jet. In art, as in data analysis, almost nothing is absolutely true, so I leave it up to you to decide. In practical terms, I think it is important to consider how to deal with print publications and the various types of color blindness. In this recipe, I visualize relatively safe colormaps with colorbars. This is a tiny selection of the many colormaps in matplotlib.

How to do it...

1. The imports are as follows:

```
import matplotlib.pyplot as plt
import matplotlib as mpl
from dautiful import plotting
```

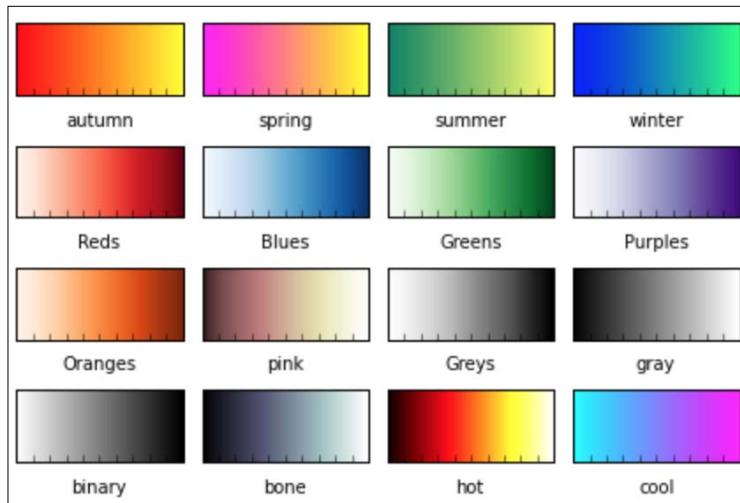
2. Plot the datasets with the following code:

```
fig, axes = plt.subplots(4, 4)
cmaps = ['autumn', 'spring', 'summer', 'winter',
          'Reds', 'Blues', 'Greens', 'Purples',
          'Oranges', 'pink', 'Greys', 'gray',
          'binary', 'bone', 'hot', 'cool']

for ax, cm in zip(axes.ravel(), cmaps):
    cmap = plt.cm.get_cmap(cm)
    cb = mpl.colorbar.ColorbarBase(ax, cmap=cmap,
                                    orientation='horizontal')
    cb.set_label(cm)
    ax.xaxis.set_ticklabels([])

plt.tight_layout()
plt.show()
```

Refer to the following plot for the end result:



The notebook is in the `choosing_colormaps.ipynb` file in this book's code bundle. The color maps are used in various visualizations in this book.

See also

- ▶ The related matplotlib documentation at <http://matplotlib.org/users/colormaps.html> (retrieved July 2015)

Interacting with IPython Notebook widgets

Interactive IPython notebook widgets are, at the time of writing (July 2015), an experimental feature. I, and as far as I know, many other people, hope that this feature will remain. In a nutshell, the widgets let you select values as you would with HTML forms. This includes sliders, drop-down boxes, and check boxes. As you can read, these widgets are very convenient for visualizing the weather data I introduced in *Chapter 1, Laying the Foundation for Reproducible Data Analysis*.

How to do it...

1. Import the following:

```
import seaborn as sns
import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt
from IPython.html.widgets import interact
from dautil import data
from dautil import ts
```

2. Load the data and request inline plots:

```
%matplotlib inline
df = data.Weather.load()
```

3. Define the following function, which displays bubble plots:

```
def plot_data(x='TEMP', y='RAIN', z='WIND_SPEED', f='A', size=10,
cmap='Blues'):
    dfx = df[x].resample(f)
    dfy = df[y].resample(f)
    dfz = df[z].resample(f)

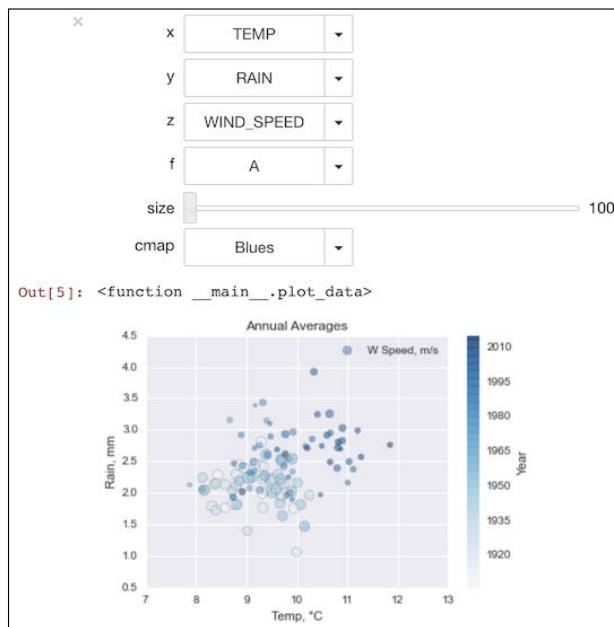
    bubbles = (dfz - dfz.min())/(dfz.max() - dfz.min())
    years = dfz.index.year
    sc = plt.scatter(dfx, dfy, s= size * bubbles + 9, c = years,
                     cmap=cmap, label=data.Weather.get_header(z),
                     alpha=0.5)
    plt.colorbar(sc, label='Year')

    freqs = {'A': 'Annual', 'M': 'Monthly', 'D': 'Daily'}
    plt.title(freqs[f] + ' Averages')
    plt.xlabel(data.Weather.get_header(x))
    plt.ylabel(data.Weather.get_header(y))
    plt.legend(loc='best')
```

4. Call the function we just defined with the following code:

```
vars = df.columns.tolist()
freqs = ('A', 'M', 'D')
cmaps = [cmap for cmap in plt.cm.datad if not cmap.endswith("_r")]
cmaps.sort()
interact(plot_data, x=vars, y=vars, z=vars, f=freqs,
size=(100,700), cmap=cmaps)
```

5. This is one of the recipes where you really should play with the code to understand how it works. The following is an example bubble plot:



6. Define another function (actually, it has the same name), but this time the function groups the data by day of year or month:

```
def plot_data(x='TEMP', y='RAIN', z='WIND_SPEED', groupby='ts.
groupby_yday', size=10, cmap='Blues'):
    if groupby == 'ts.groupby_yday':
        groupby = ts.groupby_yday
    elif groupby == 'ts.groupby_month':
        groupby = ts.groupby_month
    else:
        raise AssertionError('Unknown groupby ' + groupby)

    dfx = groupby(df[x]).mean()
    dfy = groupby(df[y]).mean()
    dfz = groupby(df[z]).mean()

    bubbles = (dfz - dfz.min())/(dfz.max() - dfz.min())
    colors = dfx.index.values
    sc = plt.scatter(dfx, dfy, s= size * bubbles + 9, c = colors,
                      cmap=cmap, label=data.Weather.get_header(z),
                      alpha=0.5)
```

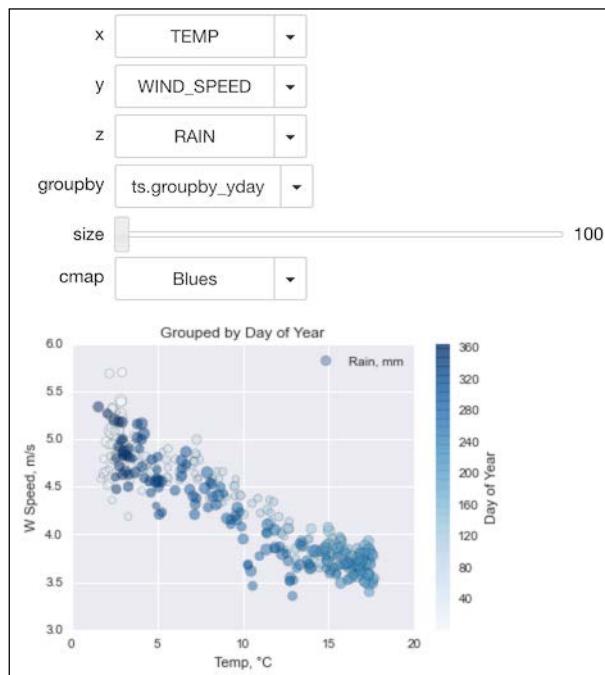
```
plt.colorbar(sc, label='Day of Year')

by_dict = {ts.groupby_yday: 'Day of Year', ts.groupby_month:
'Month'}
plt.title('Grouped by ' + by_dict[groupby])
plt.xlabel(data.Weather.get_header(x))
plt.ylabel(data.Weather.get_header(y))
plt.legend(loc='best')
```

7. Call this function with the following snippet:

```
groupbys = ('ts.groupby_yday', 'ts.groupby_month')
interact(plot_data, x=vars, y=vars, z=vars, groupby=groupbys,
size=(100,700), cmap=cmaps)
```

Refer to the following plot for the end result:



My first impression of this plot is that the temperature and wind speed seem to be correlated. The source code is in the `Interactive.ipynb` file in this book's code bundle.

See also

- ▶ The documentation on interactive IPython widgets at <https://ipython.org/ipython-doc/dev/api/generated/IPython.html.widgets.interaction.html> (retrieved July 2015)

Viewing a matrix of scatterplots

If you don't have many variables in your dataset, it is a good idea to view all the possible scatterplots for your data. You can do this with one function call from either seaborn or pandas. These functions display a matrix of plots with kernel density estimation plots or histograms on the diagonal.

How to do it...

1. Imports the following:

```
import pandas as pd
from dautil import data
from dautil import ts
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib as mpl
```

2. Load the weather data with the following lines:

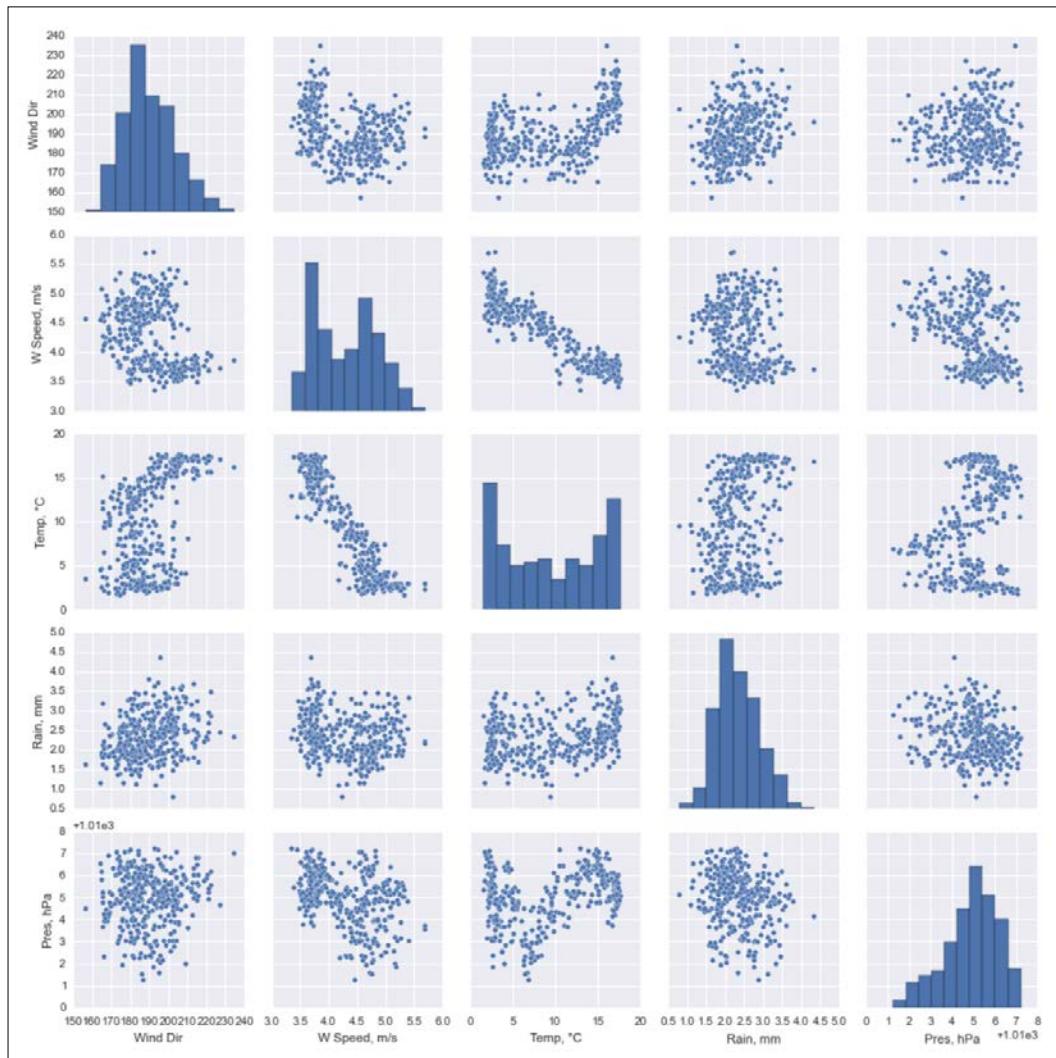
```
df = data.Weather.load()
df = ts.groupby_yday(df).mean()
df.columns = [data.Weather.get_header(c) for c in df.columns]
```

3. Plot with the Seaborn `pairplot()` function, which plots histograms on the diagonal by default:

```
%matplotlib inline

# Seaborn plotting, issues due to NaNs
sns.pairplot(df.fillna(0))
```

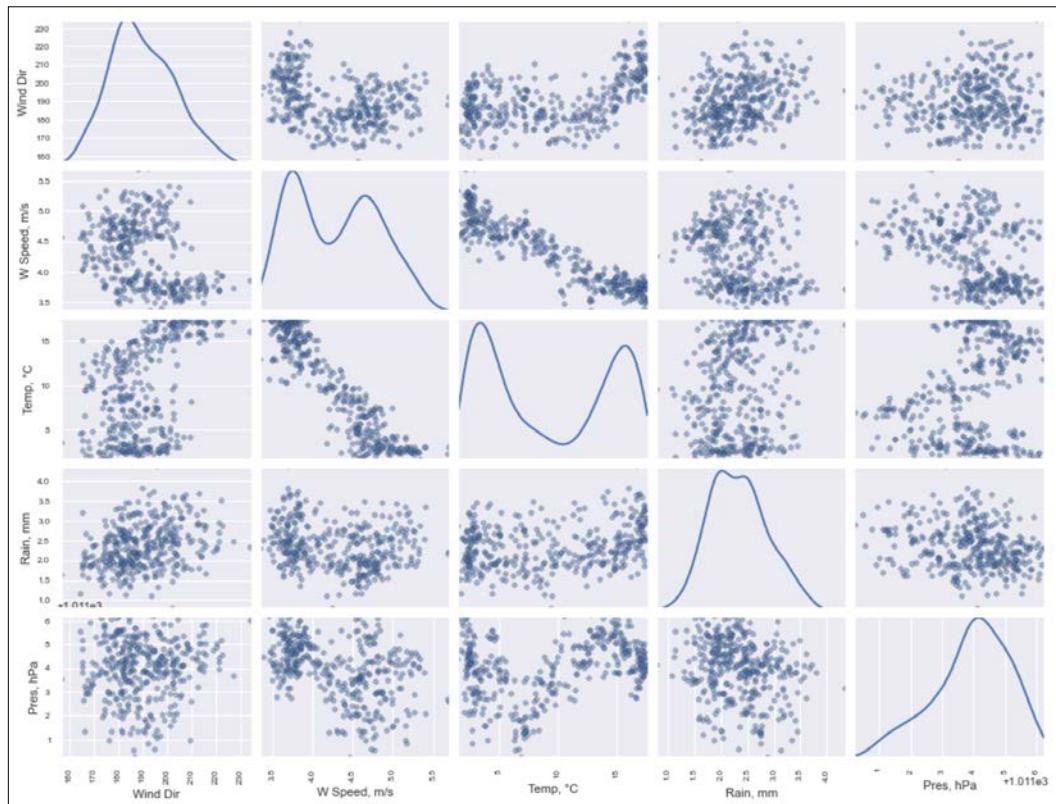
The following plots are the result:



4. Plot similarly with the pandas `scatter_matrix()` function and request kernel density estimation plots on the diagonal:

```
sns.set({'figure.figsize': '16, 12'})
mpl.rcParams['axes.linewidth'] = 9
mpl.rcParams['lines.linewidth'] = 2
plots = pd.scatter_matrix(df, marker='o', diagonal='kde')
plt.show()
```

Refer to the following plots for the end result:



The complete code is available in the `scatter_matrix.ipynb` file in this book's code bundle.

Visualizing with d3.js via mpld3

D3.js is a JavaScript data visualization library released in 2011, which we can also use in an IPython notebook. We will add hovering tooltips to a regular matplotlib plot. As a bridge, we need the `mpld3` package. This recipe doesn't require any JavaScript coding whatsoever.

Getting ready

I installed `mpld3` 0.2 with the following command:

```
$ [sudo] pip install mpld3
```

How to do it...

1. Start with the imports and enable `mpld3`:

```
%matplotlib inline
import matplotlib.pyplot as plt
import mpld3
mpld3.enable_notebook()
from mpld3 import plugins
import seaborn as sns
from dautil import data
from dautil import ts
```

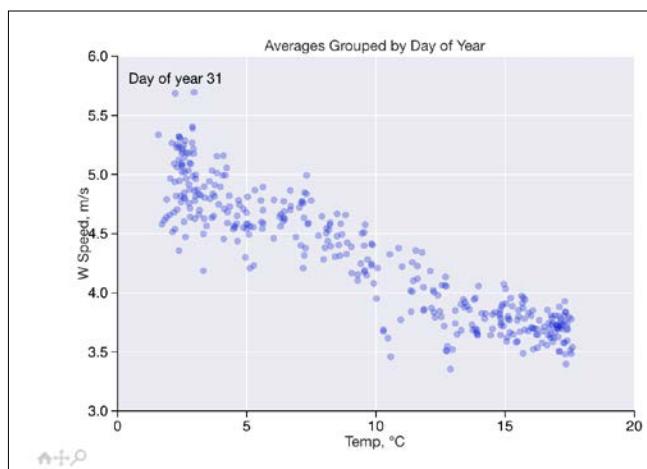
2. Load the weather data and plot it as follows:

```
df = data.Weather.load()
df = df[['TEMP', 'WIND_SPEED']]
df = ts.groupby_yday(df).mean()

fig, ax = plt.subplots()
ax.set_title('Averages Grouped by Day of Year')
points = ax.scatter(df['TEMP'], df['WIND_SPEED'],
                     s=30, alpha=0.3)
ax.set_xlabel(data.Weather.get_header('TEMP'))
ax.set_ylabel(data.Weather.get_header('WIND_SPEED'))
labels = ["Day of year {}".format(i) for i in range(366)]
tooltip = plugins.PointLabelTooltip(points, labels)

plugins.connect(fig, tooltip)
```

The highlighted lines are responsible for the tooltips. In the following screenshot, the **Day of year 31** text comes from the tooltip:



As you can see, at the bottom of the plot, you also have widgets for panning and zooming (refer to the `mpld3_demo.ipynb` file in this book's code bundle).

Creating heatmaps

Heat maps visualize data in a matrix using a set of colors. Originally, heat maps were used to represent prices of financial assets, such as stocks. Bokeh is a Python package that can display heatmaps in an IPython notebook or produce a standalone HTML file.

Getting ready

I have Bokeh 0.9.1 via Anaconda. The Bokeh installation instructions are available at <http://bokeh.pydata.org/en/latest/docs/installation.html> (retrieved July 2015).

How to do it...

1. The imports are as follows:

```
from collections import OrderedDict
from dautil import data
from dautil import ts
from dautil import plotting
import numpy as np
import bokeh.plotting as bkh_plt
from bokeh.models import HoverTool
```

2. The following function loads temperature data and groups it by year and month:

```
def load():
    df = data.Weather.load() ['TEMP']
    return ts.groupby_year_month(df)
```

3. Define a function that rearranges data in a special Bokeh structure:

```
def create_source():
    colors = plotting.sample_hex_cmap()

    month = []
    year = []
    color = []
    avg = []

    for year_month, group in load():
        month.append(ts.short_month(year_month[1]))
        year.append(str(year_month[0]))
```

```
monthly_avg = np.nanmean(group.values)
avg.append(monthly_avg)
color.append(colors[min(int(abs(monthly_avg)) - 2, 8)])
```

```
source = bkh_plt.ColumnDataSource(
    data=dict(month=month, year=year, color=color, avg=avg)
)

return year, source
```

4. Define a function that returns labels for the horizontal axis:

```
def all_years():
    years = set(year)
    start_year = min(years)
    end_year = max(years)

    return [str(y) for y in range(int(start_year), int(end_year),
      5)]
```

5. Define a plotting function for the heat map that also sets up hover tooltips:

```
def plot(year, source):
    fig = bkh_plt.figure(title="De Bilt, NL Temperature (1901 -
2014)",
                          x_range=all_years(),
                          y_range=list(reversed(ts.short_
months())),
                          toolbar_location="left",
                          tools="resize,hover,save,pan,box_
zoom,wheel_zoom")

    fig.rect("year", "month", 1, 1, source=source,
             color="color", line_color=None)

    fig.xaxis.major_label_orientation = np.pi/3

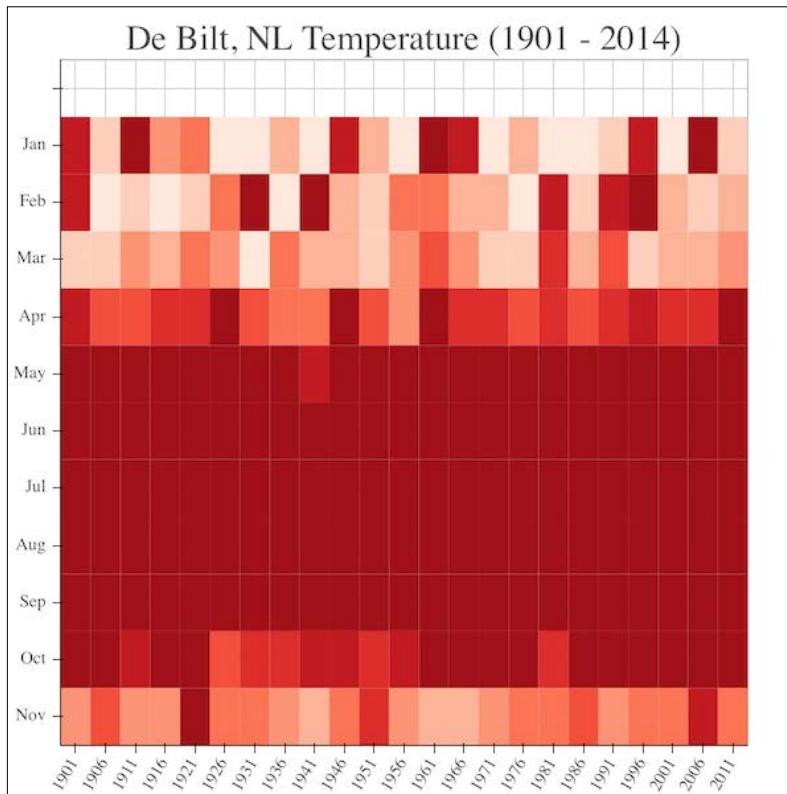
    hover = fig.select(dict(type=HoverTool))
    hover.tooltips = OrderedDict([
        ('date', '@month @year'),
        ('avg', '@avg'),
    ])

    bkh_plt.output_notebook()
    bkh_plt.show(fig)
```

6. Call the functions you defined:

```
year, source = create_source()  
plot(year, source)
```

Refer to the following plot for the end result:



The source code is available in the `heat_map.ipynb` file in this book's code bundle.

See also

- ▶ The Bokeh documentation about embedding Bokeh plots at http://bokeh.pydata.org/en/latest/docs/user_guide/embed.html (retrieved July 2015)

Combining box plots and kernel density plots with violin plots

Violin plots combine box plots and kernel density plots or histograms in one type of plot. Seaborn and matplotlib both offer violin plots. We will use Seaborn in this recipe on z-scores of weather data. The z-scoring is not essential, but without it, the violins will be more spread out.

How to do it...

1. Import the required libraries as follows:

```
import seaborn as sns  
from daultil import data  
import matplotlib.pyplot as plt
```

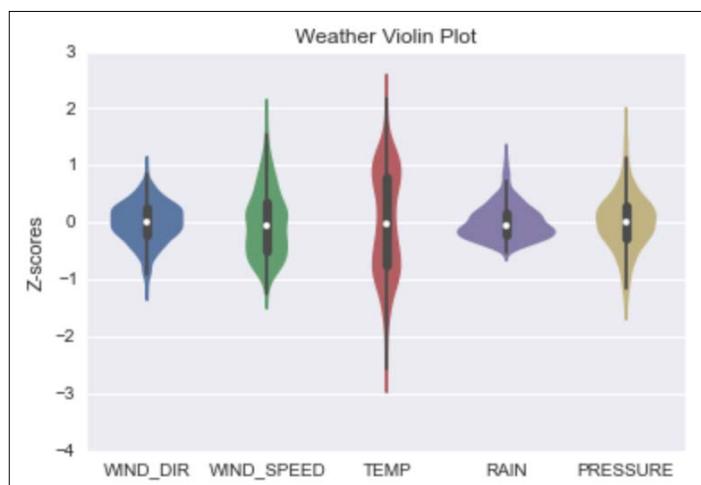
2. Load the weather data and calculate z-scores:

```
df = data.Weather.load()  
zscores = (df - df.mean()) / df.std()
```

3. Plot a violin plot of the z-scores:

```
%matplotlib inline  
plt.figure()  
plt.title('Weather Violin Plot')  
sns.violinplot(zscores.resample('M'))  
plt.ylabel('Z-scores')
```

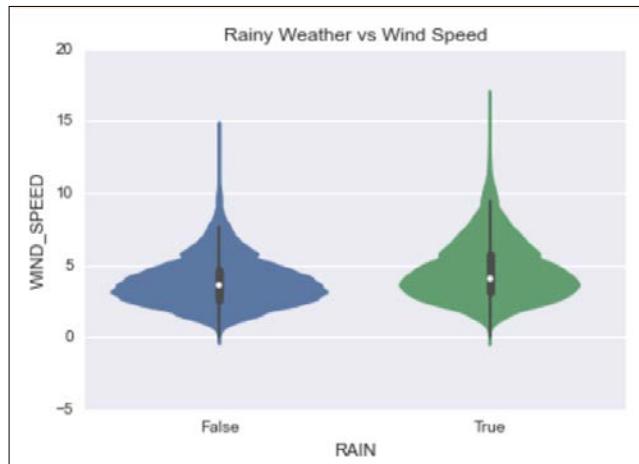
Refer to the following plot for the first violin plot:



4. Plot a violin plot of rainy and dry (the opposite of rainy) days against wind speed:

```
plt.figure()
plt.title('Rainy Weather vs Wind Speed')
categorical = df
categorical['RAIN'] = categorical['RAIN'] > 0
ax = sns.violinplot(x="RAIN", y="WIND_SPEED",
                     data=categorical)
```

Refer to the following plot for the second violin plot:



The source code is available in the `violins.ipynb` file in this book's code bundle.

See also

- ▶ The Seaborn documentation about violin plots at <https://web.stanford.edu/~mwaskom/software/seaborn/generated/seaborn.violinplot.html> (retrieved July 2015)

Visualizing network graphs with hive plots

A **hive plot** is a visualization technique for plotting network graphs. In hive plots, we draw edges as curved lines. We group nodes by some property and display them on radial axes. NetworkX is one of the most famous Python network graph libraries; however, it doesn't support hive plots yet (July 2015). Luckily, several libraries exist that specialize in hive plots. Also, we will use an API to partition the graph of Facebook users available at <https://snap.stanford.edu/data/egonets-Facebook.html> (retrieved July 2015). The data belongs to the **Stanford Network Analysis Project (SNAP)**, which also has a Python API. Unfortunately, the SNAP API doesn't support Python 3 yet.

Getting ready

I have NetworkX 1.9.1 via Anaconda. The instructions to install NetworkX are at <https://networkx.github.io/documentation/latest/install.html> (retrieved July 2015). We also need the community package at <https://bitbucket.org/taynaud/python-louvain> (retrieved July 2015). There is another package with the same name on PyPi, which is completely unrelated. Install the hiveplot package hosted at <https://github.com/ericmjl/hiveplot> (retrieved July 2015):

```
$ [sudo] pip install hiveplot
```

I wrote the code with hiveplot 0.1.7.4.

How to do it...

1. The imports are as follows:

```
import networkx as nx
import community
import matplotlib.pyplot as plt
from hiveplot import HivePlot
from collections import defaultdict
from dautil import plotting
from dautil import data
```

2. Load the data and create a NetworkX Graph object:

```
fb_file = data.SPANFB().load()
G = nx.read_edgelist(fb_file,
                     create_using = nx.Graph(),
                     nodetype = int)
print(nx.info(G))
```

3. Partition the graph and create a nodes dictionary as follows:

```
parts = community.best_partition(G)
nodes = defaultdict(list)

for n, d in parts.items():
    nodes[d].append(n)
```

-
4. The graph is pretty big, so we will just create three groups of edges:

```
edges = defaultdict(list)

for u, v in nx.edges(G, nodes[0]):
    edges[0].append((u, v, 0))

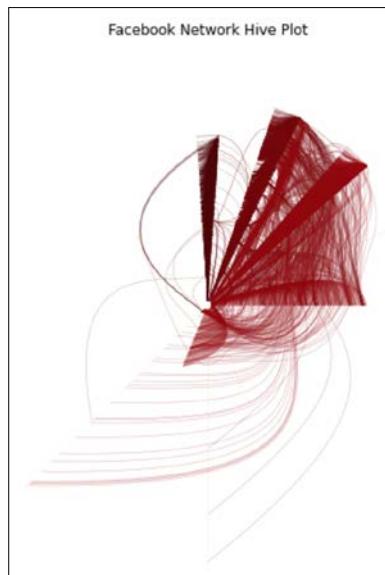
for u, v in nx.edges(G, nodes[1]):
    edges[1].append((u, v, 1))

for u, v in nx.edges(G, nodes[2]):
    edges[2].append((u, v, 2))
```

5. Plotting will take about six minutes:

```
%matplotlib inline
cmap = plotting.sample_hex_cmap(name='hot', ncolors=len(nodes.keys()))
h = HivePlot(nodes, edges, cmap, cmap)
h.draw()
plt.title('Facebook Network Hive Plot')
```

After the waiting period, we get the following plot:



The code is in the `hive_plot.ipynb` file in this book's code bundle.

Displaying geographical maps

Whether dealing with local or global data, geographical maps are a suitable visualization. To plot data on a map, we need coordinates, usually in the form of latitude and longitude values. Several file formats exist with which we can save geographical data. In this recipe, we will use the special **shapefile** format and the more common **tab separated values (TSV)** format. The shapefile format was created by the Esri company and uses three mandatory files with the extensions .shp, .shx, and .dbf. The .dbf file contains a database with extra information for each geographical location in the shapefile. The shapefile we will use contains information about country borders, population, and **Gross Domestic Product (GDP)**. We can download the shapefile with the `cartopy` library. The TSV file holds population data for more than 4000 cities as a timeseries. It comes from <https://nordpil.com/resources/world-database-of-large-cities/> (retrieved July 2015).

Getting ready

First, we need to install Proj.4 from source or, if you are lucky, using a binary distribution from <https://github.com/OSGeo/proj.4/wiki> (retrieved July 2015). The instructions to install Proj.4 are available at <https://github.com/OSGeo/proj.4> (retrieved July 2015). Then, install `cartopy` with pip—I wrote the code with `cartopy-0.13.0`. Alternatively, we can run the following command:

```
$ conda install -c scitools cartopy
```

How to do it...

1. The imports are as follows:

```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
import cartopy.io.shapereader as shpreader
import matplotlib as mpl
import pandas as pd
from dautil import options
from dautil import data
```

2. We will use color to visualize country populations and populous cities. Load the data as follows:

```
countries = shpreader.natural_earth(resolution='110m',
                                      category='cultural',
                                      name='admin_0_countries')
```

```
cities = pd.read_csv(data.Nordpil().load_urban_tsv(),
                     sep='\t', encoding='ISO-8859-1')
mill_cities = cities[cities['pop2005'] > 1000]

3. Draw a map, a corresponding colorbar, and mark populous cities on the map with the
following code:

%matplotlib inline
plt.figure(figsize=(16, 12))
gs = mpl.gridspec.GridSpec(2, 1,
                           height_ratios=[20, 1])
ax = plt.subplot(gs[0], projection=ccrs.PlateCarree())

norm = mpl.colors.Normalize(vmin=0, vmax=2 * 10 ** 9)
cmap = plt.cm.Blues
ax.set_title('Population Estimates by Country')

for country in shpreader.Reader(countries).records():
    ax.add_geometries(country.geometry, ccrs.PlateCarree(),
                      facecolor=cmap(
                          norm(country.attributes['pop_est'])))

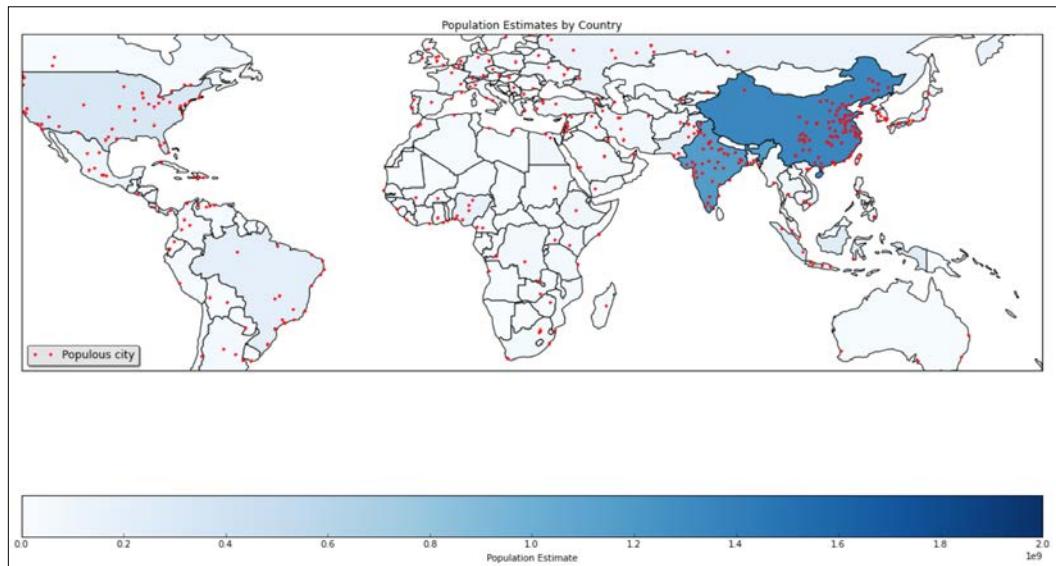
plt.plot(mill_cities['Longitude'],
         mill_cities['Latitude'], 'r.',
         label='Populous city',
         transform=ccrs.PlateCarree())

options.set_mpl_options()
plt.legend(loc='lower left')

cax = plt.subplot(gs[1])
cb = mpl.colorbar.ColorbarBase(cax,
                               cmap=cmap,
                               norm=norm,
                               orientation='horizontal')

cb.set_label('Population Estimate')
plt.tight_layout()
```

Refer to the following plot for the end result:



You can find the code in the `plot_map.ipynb` file in this book's code bundle.

Using ggplot2-like plots

Ggplot2 is an R library for data visualization popular among R users. The main idea of ggplot2 is that the product of data visualization consists of many layers. Like a painter, we start with an empty canvas and then gradually add layers of paint. Usually, we interface with R code from Python with `rpy2` (I will discuss several interoperability options in Chapter 11, of my book *Python Data Analysis*). However, if we only want to use `ggplot2`, it is more convenient to use the `pyggplot` library. In this recipe, we will visualize population growth for three countries using Worldbank data retrievable through `pandas`. The data consists of various indicators and related metadata. The spreadsheet at <http://api.worldbank.org/v2/en/topic/19?downloadformat=excel> (retrieved July 2015) has descriptions of the indicators. I think that we can consider the Worldbank dataset to be static; however, similar datasets have frequent changes quite often enough to keep an analyst busy almost full time. Obviously, changing the name of an indicator (probably) could break the code, so I decided to cache the data via the `jobjlib` library. The `jobjlib` library is related to **scikit-learn**, and we will discuss it in more detail in Chapter 9, *Ensemble Learning and Dimensionality Reduction*. Unfortunately, this approach has some limitations; in particular, we are not able to pickle all Python objects.

Getting ready

First, you need R with ggplot2 installed. If you are not going to seriously use ggplot2, maybe you should skip this recipe altogether. The homepage of R is <http://www.r-project.org/> (retrieved July 2015). The documentation of ggplot2 is at <http://docs.ggplot2.org/current/index.html> (retrieved July 2015). You can install pygplot with pip—used pygplot-2.3. To install joblib, visit <https://pythonhosted.org/joblib/installing.html> (retrieved July 2015). I have joblib 0.8.4 via Anaconda.

How to do it...

1. The imports are as follows:

```
import pygplot  
from dautil import data
```

2. Load the data with the following code:

```
dawb = data.Worldbank()  
pop_grow = dawb.get_name('pop_grow')  
df = dawb.download(indicator=pop_grow, start=1984, end=2014)  
df = dawb.rename_columns(df, use_longnames=True)
```

3. The following line initializes pygplot with the pandas DataFrame object we created:

```
p = pygplot.Plot(df)
```

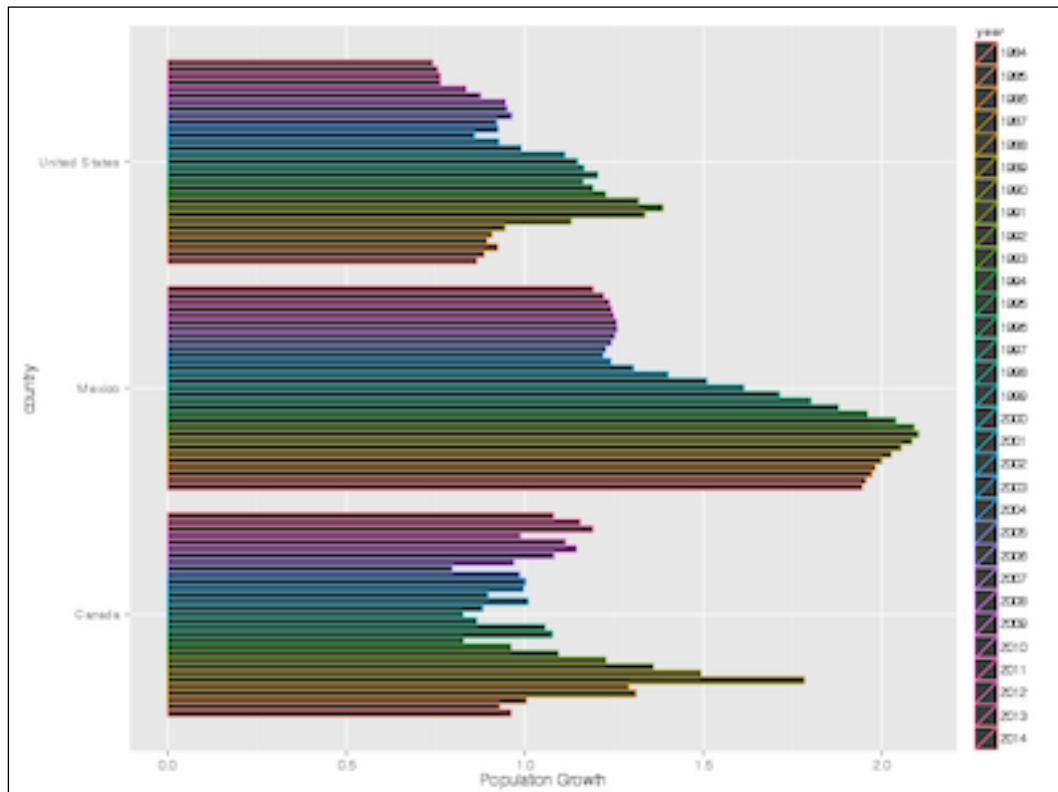
4. Add a bar chart with the following line:

```
p.add_bar('country', dawb.get_longname(pop_grow), color='year')
```

5. Flip the chart so that the bars point to the right and render:

```
p.coord_flip()  
p.render_notebook()
```

Refer to the following plot for the end result:



The code is in the `using_ggplot.ipynb` file in this book's code bundle.

Highlighting data points with influence plots

Influence plots take into account residuals after a fit, influence, and leverage for individual data points similar to bubble plots. The size of the residuals is plotted on the vertical axis and can indicate that a data point is an outlier. To understand influence plots, take a look at the following equations:

$$(2.1) \quad \text{var}(\hat{\varepsilon}_i) = \hat{\sigma}_i^2 (1 - h_{ii})$$

$$(2.2) \quad \hat{\sigma}_i^2 = \frac{1}{n-p-1} \sum_j^n \quad \forall \quad j \neq i$$

$$(2.3) \quad H = X(X'X)^{-1}X'$$

$$(2.4) \quad CookD = \frac{\sum_{j=1}^n (\hat{Y}_j - \hat{Y}_{j(i)})^2}{p \text{MSE}}$$

$$(2.5) \quad DFFITS = \frac{\hat{Y}_i - \hat{Y}_{i(i)}}{s_{(i)} \sqrt{h_{ii}}}$$

The residuals according to the `statsmodels` documentation are scaled by standard deviation **(2.1)**. In **(2.2)**, n is the number of observations and p is the number of regressors. We have a so-called **hat-matrix**, which is given by **(2.3)**.

The diagonal elements of the hat matrix give the special metric called leverage. **Leverage** serves as the horizontal axis and indicates potential influence of influence plots. In influence plots, influence determines the size of plotted points. Influential points tend to have high residuals and leverage. To measure influence, `statsmodels` can use either **Cook's distance** **(2.4)** or **DFFITS** **(2.5)**.

How to do it...

1. The imports are as follows:

```
import matplotlib.pyplot as plt
import statsmodels.api as sm
from statsmodels.formula.api import ols
from dautil import data
```

2. Get the available country codes:

```
dawb = data.Worldbank()

countries = dawb.get_countries() [['name', 'iso2c']]
```

3. Load the data from the Worldbank:

```
population = dawb.download(indicator=[dawb.get_name('pop_grow'),
dawb.get_name('gdp_pcip'),
dawb.get_name('primary_education')],
```

```
country=countries['iso2c'], start=2014,  
end=2014)  
  
population = dawb.rename_columns(population)
```

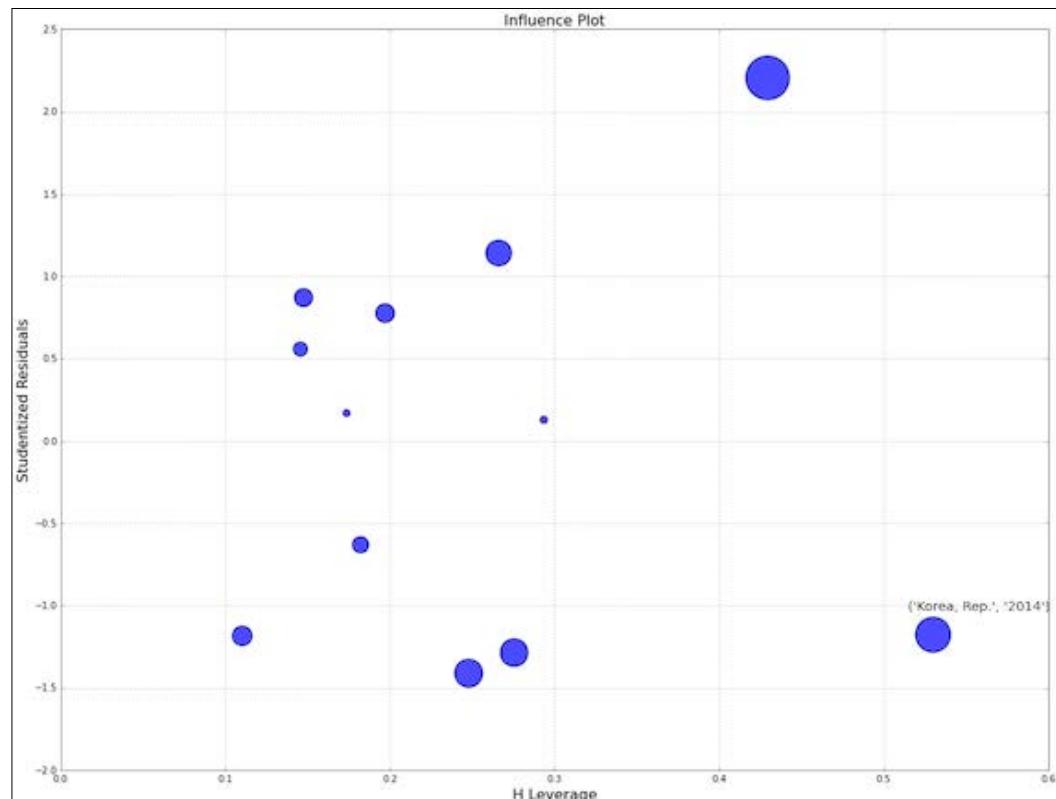
4. Define an ordinary least squares model, as follows:

```
population_model = ols("pop_grow ~ gdp_pcip + primary_education",  
data=population).fit()
```

5. Display an influence plot of the model using Cook's distance:

```
%matplotlib inline  
fig, ax = plt.subplots(figsize=(19.2, 14.4))  
fig = sm.graphics.influence_plot(population_model, ax=ax,  
criterion="cooks")  
plt.grid()
```

Refer to the following plot for the end result:



The code is in the `highlighting_influence.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about the Cook's distance at https://en.wikipedia.org/wiki/Cook%27s_distance (retrieved July 2015)
- ▶ The Wikipedia page about DFFITS at <https://en.wikipedia.org/wiki/DFFITS> (retrieved July 2015)

3

Statistical Data Analysis and Probability

We will cover the following recipes in this chapter:

- ▶ Fitting data to the exponential distribution
- ▶ Fitting aggregated data to the gamma distribution
- ▶ Fitting aggregated counts to the Poisson distribution
- ▶ Determining bias
- ▶ Estimating kernel density
- ▶ Determining confidence intervals for mean, variance, and standard deviation
- ▶ Sampling with probability weights
- ▶ Exploring extreme values
- ▶ Correlating variables with the Pearson's correlation
- ▶ Correlating variables with the Spearman rank correlation
- ▶ Correlating a binary and a continuous variable with the point-biserial correlation
- ▶ Evaluating relationships between variables with ANOVA

Introduction

Various statistical distributions have been invented, which are the equivalent of the wheel for data analysts. Just as whatever I think of comes out differently in print, data in our world doesn't follow strict mathematical laws. Nevertheless, after visualizing our data, we can see that the data follows (to certain extent) a distribution. Even without visualization, we can find a candidate distribution using rules of thumb. The next step is to try to fit the data to a known distribution. If the data is very complex, possibly due to a high number of variables, it is useful to estimate its kernel density (also useful with one variable). In all scenarios, it is good to estimate the confidence intervals or p-values of our results. When we have at least two variables, it is sometimes appropriate to have a look at the correlation between variables. In this chapter, we will apply three types of correlation.

Fitting data to the exponential distribution

The **exponential distribution** is a special case of the **gamma distribution**, which we will also encounter in this chapter. The exponential distribution can be used to analyze extreme values for rainfall. It can also be used to model the time it takes to serve a customer in a queue. For zero and negative values, the **probability distribution function (PDF)** of the exponential distribution is zero. For positive values, the PDF decays exponentially:

$$(3.1) \quad f(x; \lambda) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0, \\ 0 & x < 0. \end{cases}$$

We will use rain data as an example, which is a good candidate for an exponential distribution fit. Obviously, the amount of rain cannot be negative and we know that heavy rain is less likely than no rain at all. In fact, a day without rain is very likely.

How to do it...

The following steps fit the rain data to the exponential distribution:

1. The imports are as follows:

```
from scipy.stats.distributions import expon
import matplotlib.pyplot as plt
import dautil as dl
from IPython.display import HTML
```

2. I made a wrapper class that calls the `scipy.stats.expon` methods. First, call the `fit()` method:

```
rain = dl.data.Weather.load()['RAIN'].dropna()
dist = dl.stats.Distribution(rain, expon)
dl.options.set_pd_options()
html_builder = dl.report.HTMLBuilder()
html_builder.h1('Fitting Data to the Exponential Distribution')
loc, scale = dist.fit()
table = dl.report.DFBuilder(['loc', 'scale'])
table.row([loc, scale])
html_builder.h2('Distribution Parameters')
html_builder.add_df(table.build())
```

3. The following code calls the `scipy.stats.expon.pdf()` method and the `scipy.stats.describe()` function on the fit residuals:

```
pdf = dist.pdf(loc, scale)
html_builder.h2('Residuals of the Fit')
residuals = dist.describe_residuals()
html_builder.add(residuals.to_html())
```

4. To evaluate the fit, we can use metrics. Compute fit metrics with the following code snippet:

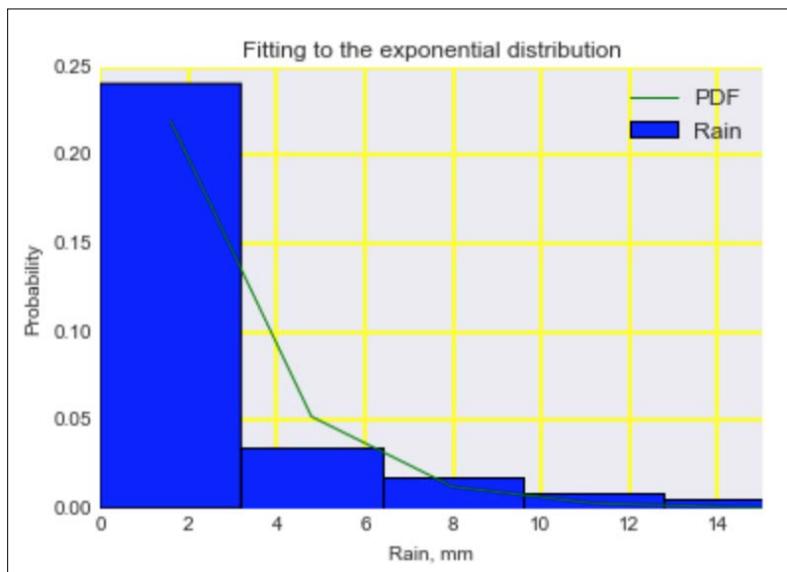
```
table2 = dl.report.DFBuilder(['Mean_AD', 'RMSE'])
table2.row([dist.mean_ad(), dist.rmse()])
html_builder.h2('Fit Metrics')
html_builder.add_df(table2.build())
```

5. Plot the fit and display the analysis report as follows:

```
plt.hist(rain, bins=dist.nbins, normed=True, label='Rain')
plt.plot(dist.x, pdf, label='PDF')
plt.title('Fitting to the exponential distribution')

# Limiting the x-axis for a better plot
plt.xlim([0, 15])
plt.xlabel(dl.data.Weather.get_header('RAIN'))
plt.ylabel('Probability')
plt.legend(loc='best')
HTML(html_builder.html)
```

Refer to the following screenshot for the end result (the code is in the `fitting_expon.ipynb` file in this book's code bundle):



How it works...

The `scale` parameter returned by `scipy.stats.expon.fit()` is the inverse of the decay parameter from **(3.1)**. We get about 2 for the `scale` value, so the decay is about half. The probability for no rain is therefore about half. The fit residuals should have a mean and skew close to 0. If we have a nonzero skew, something strange must be going on, because we don't expect the residuals to be skewed in any direction. The **mean absolute deviation (MAD)** and **root mean square error (RMSE)** are regression metrics, which we will cover in more detail in *Chapter 10, Evaluating Classifiers, Regressors, and Clusters*.

See also

- ▶ The exponential distribution Wikipedia page at https://en.wikipedia.org/wiki/Exponential_distribution (retrieved August 2015)
- ▶ The relevant SciPy documentation at <http://docs.scipy.org/doc/scipy-dev/reference/generated/scipy.stats.expon.html> (retrieved August 2015)

Fitting aggregated data to the gamma distribution

The gamma distribution can be used to model the size of insurance claims, rainfall, and the distribution of inter-spike intervals in brains. The PDF for the gamma distribution is defined by shape k and scale θ as follows:

$$(3.2) \quad f(x; k, \theta) = \frac{x^{k-1} e^{-\frac{x}{\theta}}}{\theta^k \Gamma(k)} \quad \text{for } x > 0 \text{ and } k, \theta > 0$$

$$(3.3) \quad E[X] = k\theta$$

$$(3.4) \quad Var[X] = k\theta^2$$

There is also a definition that uses an inverse scale parameter (used by SciPy). The mean and variance of the gamma distribution are described by (3.3) and (3.4). As you can see, we can estimate the shape parameter from the mean and variance using simple algebra.

How to do it...

Let's fit aggregates for the rain data for January to the gamma distribution:

1. Start with the following imports:

```
from scipy.stats.distributions import gamma
import matplotlib.pyplot as plt
import dautil as dl
import pandas as pd
from IPython.display import HTML
```

2. Load the data and select aggregates for January:

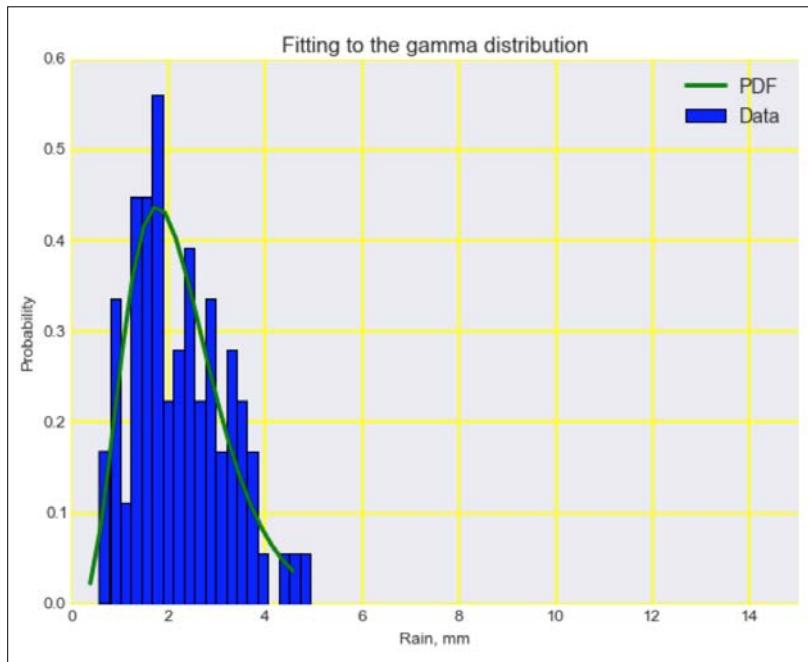
```
rain = dl.data.Weather.load() ['RAIN'].resample('M').dropna()
rain = dl.ts.groupby_month(rain)
rain = rain.get_group(1)
```

3. Derive a value for k from the mean and variance of the distribution, and use it to fit the data:

```
dist = dl.stats.Distribution(rain, gamma)

a = (dist.mean() ** 2)/dist.var()
shape, loc, scale = dist.fit(a)
```

The rest of the code is similar to the code in *Fitting data to the exponential distribution*. Refer to the following screenshot for the end result (the code is in the `fitting_gamma.ipynb` file in this book's code bundle):



See also

- ▶ The relevant SciPy documentation at <http://docs.scipy.org/doc/scipy-dev/reference/generated/scipy.stats.gamma.html#scipy.stats.gamma> (retrieved August 2015)
- ▶ The Wikipedia page for the gamma distribution at https://en.wikipedia.org/wiki/Gamma_distribution (retrieved August 2015)

Fitting aggregated counts to the Poisson distribution

The **Poisson distribution** is named after the French mathematician Poisson, who published a thesis about it in 1837. The Poisson distribution is a discrete distribution usually associated with counts for a fixed interval of time or space. It is only defined for integer values k . For instance, we could apply it to monthly counts of rainy days. In this case, we implicitly assume that the event of a rainy day occurs at a fixed monthly rate. The goal of fitting the data to the Poisson distribution is to find the fixed rate.

The following equations describe the probability mass function **(3.5)** and rate parameter **(3.6)** of the Poisson distribution:

$$(3.5) \quad f(k; \lambda) = \Pr(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}$$

$$(3.6) \quad \lambda = E(X) = \text{Var}(X)$$

How to do it...

The following steps fit using the **maximum likelihood estimation (MLE)** method:

1. The imports are as follows:

```
from scipy.stats.distributions import poisson
import matplotlib.pyplot as plt
import dautil as dl
from scipy.optimize import minimize
from IPython.html.widgets.interaction import interactive
from IPython.core.display import display
from IPython.core.display import HTML
```

2. Define the function to maximize:

```
def log_likelihood(k, mu):
    return poisson.logpmf(k, mu).sum()
```

3. Load the data and group it by month:

```
def count_rain_days(month):
    rain = dl.data.Weather.load()['RAIN']
    rain = (rain > 0).resample('M', how='sum')
    rain = dl.ts.groupby_month(rain)
    rain = rain.get_group(month)

    return rain
```

4. Define the following visualization function:

```
def plot(rain, dist, params, month):
    fig, ax = plt.subplots()
    plt.title('Fitting to the Poisson distribution ({})'.
format(dl.ts.short_month(month)))

    # Limiting the x-axis for a better plot
    plt.xlim([0, 15])
    plt.figtext(0.5, 0.7, 'rate {:.3f}'.format(params.x[0]),
alpha=0.7,
```

```

    fontsize=14)
plt.xlabel('# Rainy days in a month')
plt.ylabel('Probability')
ax.hist(dist.train, bins=dist.nbins, normed=True,
label='Data')
ax.plot(dist.x, poisson.pmf(dist.x, params.x))

```

5. Define a function to serve as the entry point:

```

def fit_poisson(month):
    month_index = dl.ts.month_index(month)
    rain = count_rain_days(month_index)

    dist = dl.stats.Distribution(rain, poisson, range=[-0.5,
19.5])
    params = minimize(log_likelihood, x0=rain.mean(),
args=(rain,))
    plot(rain, dist, params, month_index)

```

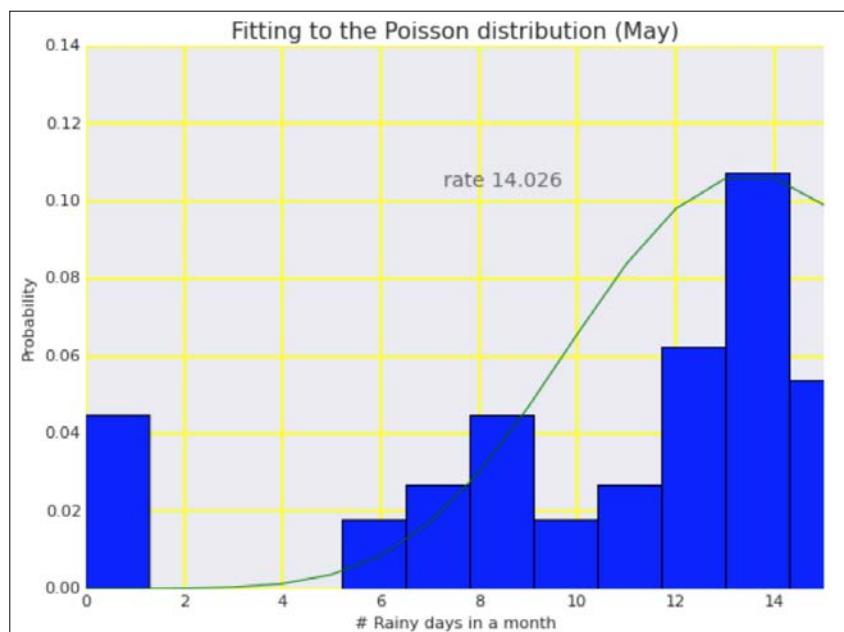
6. Use interactive widgets so we can display a plot for each month:

```

display(interactive(fit_poisson, month=dl.nb.create_month_
widget(month='May')))
HTML(dl.report.HTMLBuilder().watermark())

```

Refer to the following screenshot for the end result (see the fitting_poisson.ipynb file in this book's code bundle):



See also

- ▶ The Poisson distribution Wikipedia page at https://en.wikipedia.org/wiki/Poisson_distribution (retrieved August 2015)
- ▶ The related SciPy documentation at <http://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.poisson.html#scipy.stats.poisson> (retrieved August 2015)

Determining bias

When teaching probability, it is customary to give examples of coin tosses. Whether it is going to rain or not is more or less like a coin toss. If we have two possible outcomes, the **binomial distribution** is appropriate. This distribution requires two parameters: the probability and the sample size.

In statistics, there are two generally accepted approaches. In the **frequentist** approach, we measure the number of coin tosses and use that frequency for further analysis. **Bayesian** analysis is named after its founder the Reverend Thomas Bayes. The Bayesian approach is more incremental and requires a **prior distribution**, which is the distribution we assume before performing experiments. The **posterior distribution** is the distribution we are interested in and which we obtain after getting new data from experiments. Let's first have a look at the following equations:

$$(3.7) \quad f(k; n, p) = \Pr(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

$$(3.8) \quad \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

$$(3.9) \quad P(a < p < b | m; n) = \frac{\int_a^b \binom{n+m}{m} p^m (1-p)^n dp}{\int_0^1 \binom{n+m}{m} p^m (1-p)^n dp}$$

(3.7) and (3.8) describe the probability mass function for the binomial distribution. (3.9) comes from an essay published by Bayes. The equation is about an experiment with m successes and n failures and assumes a uniform prior distribution for the probability parameter of the binomial distribution.

How to do it...

In this recipe, we will apply the frequentist and Bayesian approach to rain data:

1. The imports are as follows:

```
import dutil as dl
from scipy import stats
import matplotlib.pyplot as plt
import numpy as np
from IPython.html.widgets.interaction import interact
from IPython.display import HTML
```

2. Define the following function to load the data:

```
def load():
    rainy = dl.data.Weather.rain_values() > 0
    n = len(rainy)
    nrains = np.cumsum(rainy)

    return n, nrains
```

3. Define the following function to compute the posterior:

```
def posterior(i, u, data):
    return stats.binom(i, u).pmf(data[i])
```

4. Define the following function to plot the posterior for the subset of the data:

```
def plot_posterior(ax, day, u, nrains):
    ax.set_title('Posterior distribution for day {}'.format(day))
    ax.plot(posterior(day, u, nrains),
            label='rainy days in period={}'.format(nrains[day]))
    ax.set_xlabel('Uniform prior parameter')
    ax.set_ylabel('Probability rain')
    ax.legend(loc='best')
```

5. Define the following function to do the plotting:

```
def plot(day1=1, day2=30):
    fig, [[upleft, upright], [downleft, downright]] = plt.
    subplots(2, 2)
    plt.suptitle('Determining bias of rain data')
    x = np.arange(n) + 1
    upleft.set_title('Frequentist Approach')
    upleft.plot(x, nrains/x, label='Probability rain')
    upleft.set_xlabel('Days')
    set_ylabel(upleft)

    max_p = np.zeros(n)
    u = np.linspace(0, 1, 100)
```

```

for i in x - 1:
    max_p[i] = posterior(i, u, nrains).argmax()/100

downleft.set_title('Bayesian Approach')
downleft.plot(x, max_p)
downleft.set_xlabel('Days')
set_ylabel(downleft)

plot_posterior(upright, day1, u, nrains)
plot_posterior(downright, day2, u, nrains)
plt.tight_layout()

```

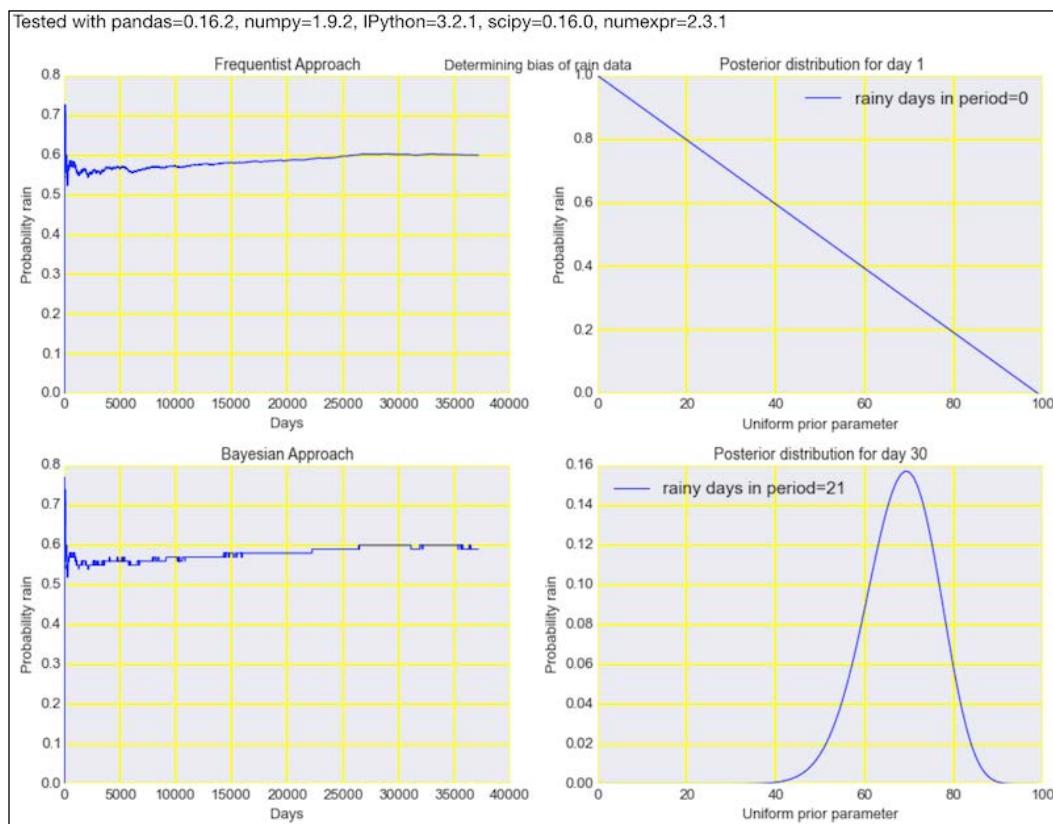
6. The following lines call the other functions and place a watermark:

```

interact(plot, day1=(1, n), day2=(1, n))
HTML(dl.report.HTMLBuilder().watermark())

```

Refer to the following screenshot for the end result (see the determining_bias.ipynb file in this book's code bundle):



See also

- The Wikipedia page about the essay mentioned in this recipe is at https://en.wikipedia.org/wiki/An_Essay_towards_solving_a_Problem_in_the_Doctrine_of_Chances (retrieved August 2015)

Estimating kernel density

Often, we have an idea about the kind of distribution that is appropriate for our data. If that is not the case, we can apply a procedure called **kernel density estimation**. This method doesn't make any assumptions and is nonparametric. We basically smooth the data in an attempt to get a handle on the probability density. To smooth data, we can use various functions. These functions are called kernel functions in this context. The following equation defines the estimator:

$$(3.10) \quad \hat{f}_h(x) = \frac{1}{n} \sum_{i=1}^n K_h(x - x_i) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - x_i}{h}\right)$$

In the preceding formula, K is the kernel function, a function with properties similar to a PDF. The bandwidth h parameter controls the smoothing process and can be kept fixed or varied. Some libraries use rules of thumb to calculate h , while others let you specify its value. SciPy, statsmodels, scikit-learn, and Seaborn implement kernel density estimation using different algorithms.

How to do it...

In this recipe, we will estimate bivariate kernel density using weather data:

1. The imports are as follows:

```
import seaborn as sns
import matplotlib.pyplot as plt
import dutil as dl
from dutil.stats import zscores
import statsmodels.api as sm
from sklearn.neighbors import KernelDensity
import numpy as np
from scipy import stats
from IPython.html import widgets
from IPython.core.display import display
from IPython.display import HTML
```

2. Define the following function to plot the estimated kernel density:

```
def plot(ax, a, b, c, xlabel, ylabel):
    dl.plotting.scatter_with_bar(ax, 'Kernel Density', a.values,
        b.values, c=c, cmap='Blues')
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
```

3. In the following notebook cell, load the data and define widgets for the selection of weather variables:

```
df = dl.data.Weather.load().resample('M').dropna()
columns = [str(c) for c in df.columns.values]
var1 = widgets.Dropdown(options=columns, selected_label='RAIN')
display(var1)
var2 = widgets.Dropdown(options=columns, selected_label='TEMP')
display(var2)
```

4. In the next notebook cell, define variables using the values of the widgets we created:

```
x = df[var1.value]
xlabel = dl.data.Weather.get_header(var1.value)
y = df[var2.value]
ylabel = dl.data.Weather.get_header(var2.value)
X = [x, y]
```

5. The next notebook cell does the heavy lifting with the most important lines highlighted:

```
# need to use zscores to avoid errors
Z = [zscores(x), zscores(y)]
kde = stats.gaussian_kde(Z)

_, [[sp_ax, sm_ax], [sk_ax, sns_ax]] = plt.subplots(2, 2)
plot(sp_ax, x, y, kde.pdf(Z), xlabel, ylabel)
sp_ax.set_title('SciPy')

sm_kde = sm.nonparametric.KDEMultivariate(data=X, var_type='cc',
                                             bw='normal_reference')
sm_ax.set_title('statsmodels')
plot(sm_ax, x, y, sm_kde.pdf(X), xlabel, ylabel)

XT = np.array(X).T
sk_kde = KernelDensity(kernel='gaussian', bandwidth=0.2).fit(XT)
sk_ax.set_title('Scikit Learn')
plot(sk_ax, x, y, sk_kde.score_samples(XT), xlabel, ylabel)
```

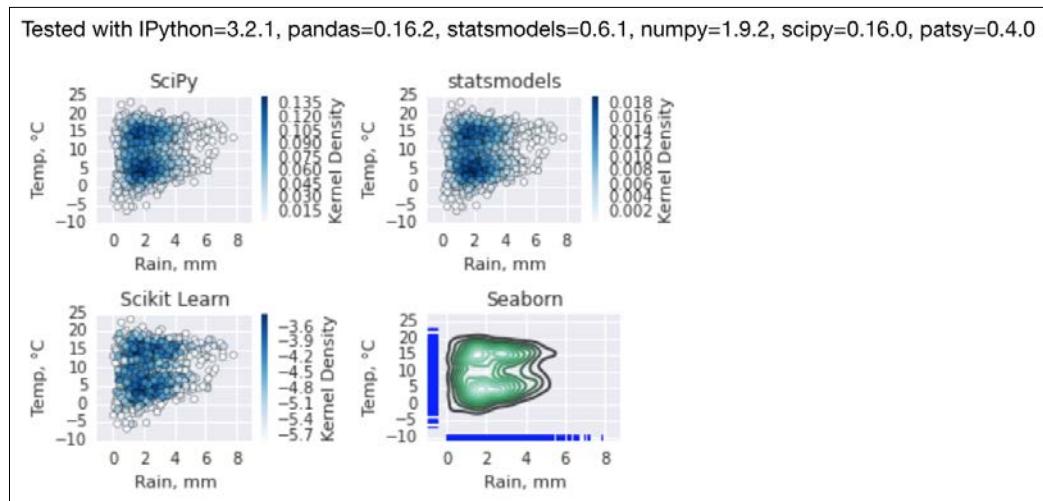
```

sns_ax.set_title('Seaborn')
sns.kdeplot(x, y, ax=sns_ax)
sns.rugplot(x, color="b", ax=sns_ax)
sns.rugplot(y, vertical=True, ax=sns_ax)
sns_ax.set_xlabel(xlabel)
sns_ax.set_ylabel(ylabel)

plt.tight_layout()
HTML(dl.report.HTMLBuilder().watermark())

```

Refer to the following screenshot for the end result (see the `kernel_density_estimation.ipynb` file in this book's code bundle):



See also

- ▶ The kernel density estimation Wikipedia page at https://en.wikipedia.org/wiki/Kernel_density_estimation (retrieved August 2015)
- ▶ The related statsmodels documentation at http://statsmodels.sourceforge.net/devel/generated/statsmodels.nonparametric.kernel_density.KDEMultivariate.html (retrieved August 2015)
- ▶ The related scikit-learn documentation at <http://scikit-learn.org/stable/modules/density.html> (retrieved August 2015)

Determining confidence intervals for mean, variance, and standard deviation

It is sometimes useful to imagine that the data we observe is just the tip of an iceberg. If you get into this mindset, then you probably will want to know how big this iceberg actually is. Obviously, if you can't see the whole thing, you can still try to extrapolate from the data you have. In statistics we try to estimate confidence intervals, which are an estimated range usually associated with a certain confidence level quoted in percentages.

The `scipy.stats.bayes_mvs()` function estimates confidence intervals for mean, variance, and standard deviation. The function uses Bayesian statistics to estimate confidence assuming that the data is independent and normally distributed. **Jackknifing** is an alternative deterministic algorithm to estimate confidence intervals. It falls under the family of resampling algorithms. Usually, we generate new datasets under the jackknifing algorithm by deleting one value (we can also delete two or more values). We generate data N times, where N is the number of values in the dataset. Typically, if we want a 5 percent confidence level, we estimate the means or variances for the new datasets and determine the 2.5 and 97.5 percentile values.

How to do it...

In this recipe, we estimate confidence intervals with the `scipy.stats.bayes_mvs()` function and jackknifing:

1. The imports are as follows:

```
from scipy import stats
import dautil as dl
from dautil.stats import jackknife
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from IPython.html.widgets.interaction import interact
from IPython.display import HTML
```

2. Define the following function to visualize the Scipy result using error bars:

```
def plot_bayes(ax, metric, var, df):
    vals = np.array([[v.statistic, v.minmax[0], v.minmax[1]] for v
in
                    df[metric].values])

    ax.set_title('Bayes {}'.format(metric))
    ax.errorbar(np.arange(len(vals)), vals.T[0], yerr=(vals.T[1],
vals.T[2]))
    set_labels(ax, var)
```

-
3. Define the following function to visualize the jackknifing result using error bars:

```
def plot_jackknife(ax, metric, func, var, df):
    vals = df.apply(lambda x: jackknife(x, func, alpha=0.95))
    vals = np.array([[v[0], v[1], v[2]] for v in vals.values])
```

```
    ax.set_title('Jackknife {}'.format(metric))
    ax.errorbar(np.arange(len(vals)), vals.T[0], yerr=(vals.T[1],
    vals.T[2]))
    set_labels(ax, var)
```

4. Define the following function, which will be called with the help of an IPython interactive widget:

```
def confidence_interval(var='TEMP'):
    df = dl.data.Weather.load().dropna()
    df = dl.ts.groupby_yday(df)
```

```
    def f(x):
        return stats.bayes_mvs(x, alpha=0.95)
```

```
    bayes_df = pd.DataFrame([[v[0], v[1], v[2]] for v in
    df[var].apply(f).values],
    columns=['Mean', 'Var',
    'Std'])
```

```
    fig, axes = plt.subplots(2, 2)
    fig.suptitle('Confidence Intervals')
```

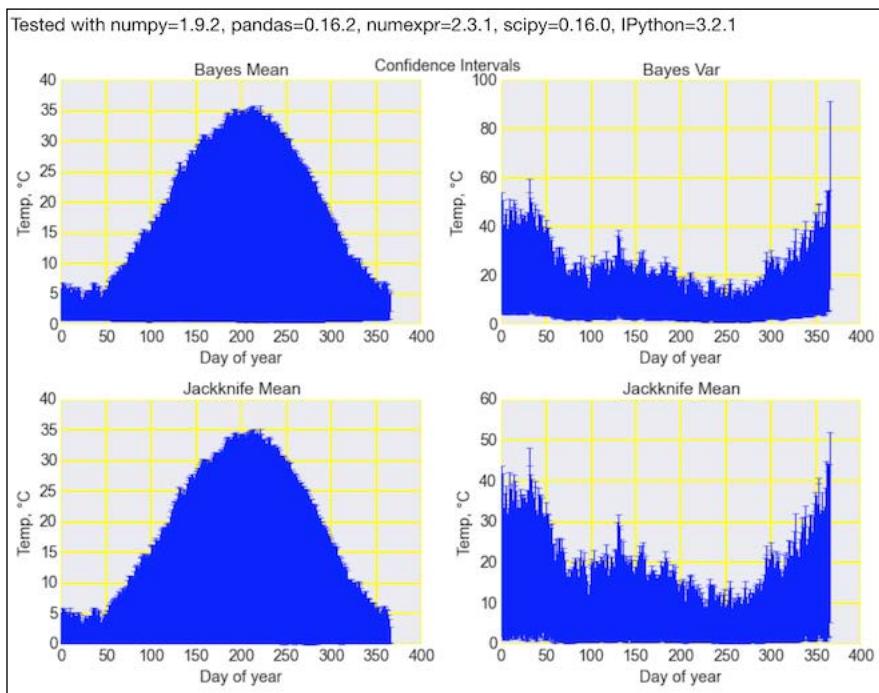
```
    plot_bayes(axes[0][0], 'Mean', var, bayes_df)
    plot_bayes(axes[0][1], 'Var', var, bayes_df)
    plot_jackknife(axes[1][0], 'Mean', np.mean, var, df[var])
    plot_jackknife(axes[1][1], 'Mean', np.var, var, df[var])
```

```
    plt.tight_layout()
```

5. Set up an interactive IPython widget:

```
interact(confidence_interval, var=dl.data.Weather.get_headers())
HTML(dl.report.HTMLBuilder().watermark())
```

Refer to the following screenshot for the end result (see the `bayes_confidence.ipynb` file in this book's code bundle):



See also

- ▶ The Wikipedia page on jackknife resampling at https://en.wikipedia.org/wiki/Jackknife_resampling (retrieved August 2015)
- ▶ T.E. Oliphant, "A Bayesian perspective on estimating mean, variance, and standard-deviation from data" (<http://hdl.handle.net/1877/438>, 2006)

Sampling with probability weights

To create the nuclear bomb during the Second World War, physicists needed to perform pretty complicated calculations. Stanislaw Ulam got the idea to treat this challenge as a game of chance. Later, the method he came up with was given the code name **Monte Carlo**. Games of chance usually have very simple rules, but playing in an optimal way can be difficult. According to quantum mechanics, subatomic particles are also unpredictable. If we simulate many experiments with subatomic particles, we still can get an idea of how they are likely to behave. The Monte Carlo method is not deterministic, but it approaches the correct result for a complex computation for a sufficiently large number of simulations.

The `statsmodels.distributions.empirical_distribution.ECDF` class defines the cumulative distribution function of a data array. We can use its output to simulate a complex process. This simulation is not perfect, because we lose information in the process.

How to do it...

In this recipe, we will simulate weather processes. In particular, I am interested in annual temperature values. I am interested in finding out whether the simulated data sets also show an upward trend:

1. The imports are as follows:

```
from statsmodels.distributions.empirical_distribution import ECDF
import dautil as dl
import numpy as np
import matplotlib.pyplot as plt
from sklearn.utils import check_random_state
from IPython.html.widgets.interaction import interact
from IPython.core.display import HTML
```

2. Define the following function to calculate the slope:

```
def slope(x, y):
    return np.polyfit(x, y, 1)[0]
```

3. Define the following function to generate data for a single year:

```
def simulate(x, years, rs, p):
    N = len(years)
    means = np.zeros(N)

    for i in range(N):
        sample = rs.choice(x, size=365, p=p)
        means[i] = sample.mean()

    return means, np.diff(means).mean(), slope(years, means)
```

4. Define the following function to run multiple simulations:

```
def run_multiple(times, x, years, p):
    sims = []
    rs = check_random_state(20)

    for i in range(times):
        sims.append(simulate(x, years, rs, p))

    return np.array(sims)
```

5. Define the following function, which by default loads temperature values:

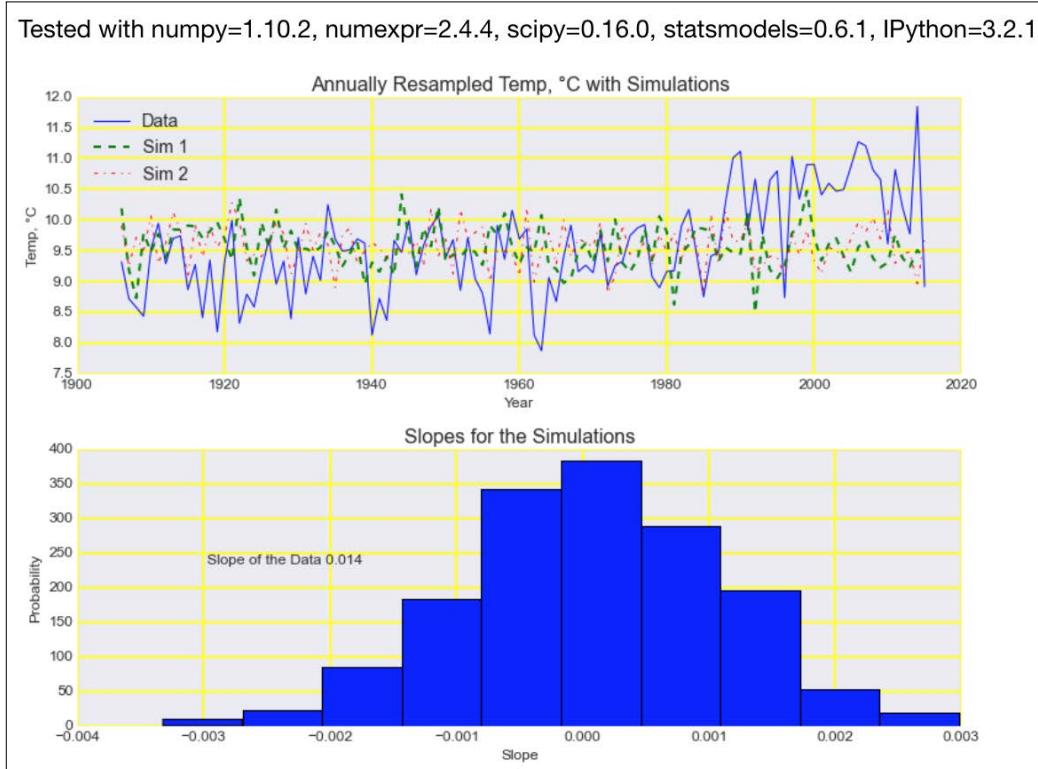
```
def main(var='TEMP'):
    df = dl.data.Weather.load().dropna()[var]
    cdf = ECDF(df)
    x = cdf.x[1:]
    p = np.diff(cdf.y)

    df = df.resample('A')
    years = df.index.year
    sims = run_multiple(500, x, years, p)

    sp = dl.plotting.Subplotter(2, 1, context)
    plotter = dl.plotting.CyclePlotter(sp.ax)
    plotter.plot(years, df.values, label='Data')
    plotter.plot(years, sims[0][0], label='Sim 1')
    plotter.plot(years, sims[1][0], label='Sim 2')
    header = dl.data.Weather.get_header(var)
    sp.label(title_params=header, ylabel_params=header)
    sp.ax.legend(loc='best')

    sp.next_ax()
    sp.label()
    sp.ax.hist(sims.T[2], normed=True)
    plt.figtext(0.2, 0.3, 'Slope of the Data {:.3f}'.format(slope(years, df.values)))
    plt.tight_layout()
```

The notebook stored in the `sampling_weights.ipynb` file in this book's code bundle gives you the option to select other weather variables too. Refer to the following screenshot for the end result:



See also

- ▶ The Wikipedia page for the Monte Carlo method at https://en.wikipedia.org/wiki/Monte_Carlo_method (retrieved August 2015)
- ▶ The documentation for the ECDF class at [http://statsmodels.sourceforge.net/0.6.0/generated/statsmodels.distributions.empirical_distribution\(ECDF\).html](http://statsmodels.sourceforge.net/0.6.0/generated/statsmodels.distributions.empirical_distribution(ECDF).html) (retrieved August 2015)

Exploring extreme values

Worldwide, there are almost a million dams, roughly 5 percent of which are higher than 15 m. A civil engineer designing a dam will have to consider many factors, including rainfall. Let's assume, for the sake of simplicity, that the engineer wants to know the cumulative annual rainfall. We can also take monthly maximums and fit those to a **generalized extreme value (GEV)** distribution. Using this distribution, we can then bootstrap to get our estimate. Instead, I select values that are above the 95th percentile in this recipe.

The GEV distribution is implemented in `scipy.stats` and is a mixture of the Gumbel, Frechet, and Weibull distributions. The following equations describe the cumulative distribution function (3.11) and a related constraint (3.12):

$$(3.11) \quad F(x; \mu, \sigma, \xi) = \exp \left\{ - \left[1 + \xi \left(\frac{x - \mu}{\sigma} \right) \right]^{-1/\xi} \right\}$$

$$(3.12) \quad 1 + \xi(x - \mu) / \sigma > 0$$

In these equations, μ is the location parameter, σ is the scale parameter, and ξ is the shape parameter.

How to do it...

Let's analyze the data using the GEV distribution:

1. The imports are as follows:

```
from scipy.stats.distributions import genextreme
import matplotlib.pyplot as plt
import dautil as dl
import numpy as np
from IPython.display import HTML
```

2. Define the following function to sample the GEV distribution:

```
def run_sims(nsims):
    sums = []

    np.random.seed(19)

    for i in range(nsims):
        for j in range(len(years)):
            sample_sum = dist.rvs(shape, loc, scale, size=365).
sum()
```

```
    sums.append(sample_sum)
```

```
    a = np.array(sums)
    low, high = dl.stats.ci(a)

    return a, low, high
```

3. Load the data and select the extreme values:

```
rain = dl.data.Weather.load()['RAIN'].dropna()
annual_sums = rain.resample('A', how=np.sum)
years = np.unique(rain.index.year)
limit = np.percentile(rain, 95)
rain = rain[rain > limit]
dist = dl.stats.Distribution(rain, genextreme)
```

4. Fit the extreme values to the GEV distribution:

```
shape, loc, scale = dist.fit()
table = dl.report.DFBuilder(['shape', 'loc', 'scale'])
table.row([shape, loc, scale])
dl.options.set_pd_options()
html_builder = dl.report.HTMLBuilder()
html_builder.h1('Exploring Extreme Values')
html_builder.h2('Distribution Parameters')
html_builder.add_df(table.build())
```

5. Get statistics on the fit residuals:

```
pdf = dist.pdf(shape, loc, scale)
html_builder.h2('Residuals of the Fit')
residuals = dist.describe_residuals()
html_builder.add(residuals.to_html())
```

6. Get the fit metrics:

```
table2 = dl.report.DFBuilder(['Mean_AD', 'RMSE'])
table2.row([dist.mean_ad(), dist.rmse()])
html_builder.h2('Fit Metrics')
html_builder.add_df(table2.build())
```

-
7. Plot the data and the result of the bootstrap:

```
sp = dl.plotting.Subplotter(2, 2, context)

sp.ax.hist(annual_sums, normed=True, bins=dl.stats.sqrt_
bins(annual_sums))
sp.label()
set_labels(sp.ax)

sp.next_ax()
sp.label()
sp.ax.set_xlim([5000, 10000])
sims = []
nsims = [25, 50, 100, 200]

for n in nsims:
    sims.append(run_sims(n))

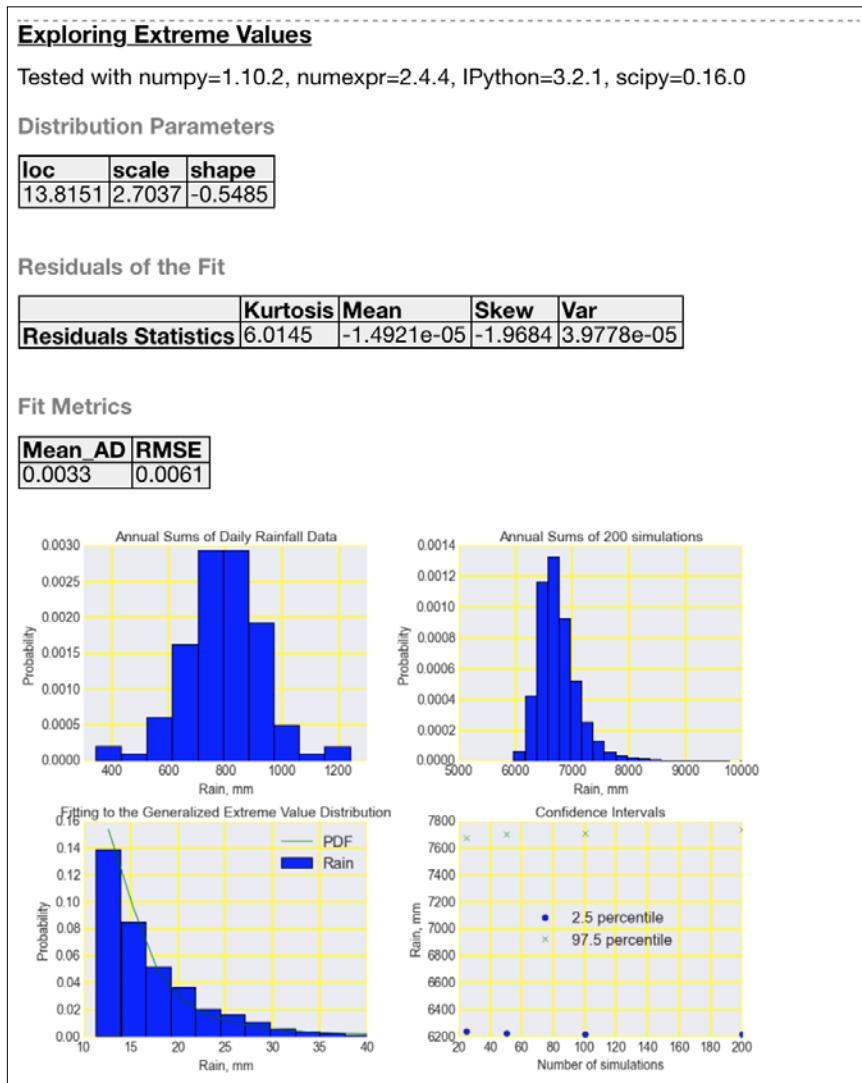
sims = np.array(sims)
sp.ax.hist(sims[2][0], normed=True, bins=dl.stats.sqrt_
bins(sims[2][0]))
set_labels(sp.ax)

sp.next_ax()
sp.label()
sp.ax.set_xlim([10, 40])
sp.ax.hist(rain, bins=dist.nbins, normed=True, label='Rain')
sp.ax.plot(dist.x, pdf, label='PDF')
set_labels(sp.ax)
sp.ax.legend(loc='best')

sp.next_ax()
sp.ax.plot(nsims, sims.T[1], 'o', label='2.5 percentile')
sp.ax.plot(nsims, sims.T[2], 'x', label='97.5 percentile')
sp.ax.legend(loc='center')
sp.label(ylabel_params=dl.data.Weather.get_header('RAIN'))

plt.tight_layout()
HTML(html_builder.html)
```

Refer to the following screenshot for the end result (see the `extreme_values.ipynb` file in this book's code bundle):



See also

- ▶ The Wikipedia page on the GEV distribution at https://en.wikipedia.org/wiki/Generalized_extreme_value_distribution (retrieved August 2015).

Correlating variables with Pearson's correlation

Pearson's r, named after its developer Karl Pearson (1896), measures linear correlation between two variables. Let's look at the following equations:

$$(3.13) \quad r = r_{xy} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}$$

$$(3.14) \quad F(r) = \frac{1}{2} \ln \frac{1+r}{1-r} = \operatorname{arctanh}(r)$$

$$(3.15) \quad SE = \frac{1}{\sqrt{n-3}}$$

$$(3.16) \quad z = \frac{x - \text{mean}}{SE} = [F(r) - F(\rho_0)] \sqrt{n-3}$$

(3.13) defines the coefficient and (3.14) describes the **Fisher transformation** used to compute confidence intervals. (3.15) gives the standard error of the correlation. (3.16) is about the z-score of the Fisher transformed correlation. If we assume a normal distribution, we can use the z-score to compute confidence intervals. Alternatively, we can bootstrap by resampling pairs of values with replacement. Also, the `scipy.stats.pearsonr()` function returns a p-value, which (according to the documentation) is not accurate for samples of less than 500 values. Unfortunately, we are going to use such a small sample in this recipe. We are going to correlate carbon dioxide emission data from the Worldbank with related temperature data for the Netherlands.

How to do it...

In this recipe, we will compute the correlation coefficient and estimate confidence intervals using z-scores and bootstrapping with the following steps:

1. The imports are as follows:

```
import dutil as dl
import pandas as pd
from scipy import stats
import numpy as np
import math
from sklearn.utils import check_random_state
import matplotlib.pyplot as plt
from IPython.display import HTML
from IPython.display import display
```

-
2. Download the data and set up appropriate data structures:

```
wb = dl.data.Worldbank()
indicator = wb.get_name('CO2')
co2 = wb.download(country='NL', indicator=indicator, start=1900,
                  end=2014)
co2.index = [int(year) for year in co2.index.get_level_values(1)]
temp = pd.DataFrame(dl.data.Weather.load()['TEMP'].resample('A'))
temp.index = temp.index.year
temp.index.name = 'year'
df = pd.merge(co2, temp, left_index=True, right_index=True).
dropna()
```

3. Compute the correlation as follows:

```
stats_corr = stats.pearsonr(df[indicator].values, df['TEMP'].
values)
print('Correlation={0:.4g}, p-value={1:.4g}'.format(stats_corr[0],
stats_corr[1]))
```

4. Calculate the confidence interval with the Fisher transform:

```
z = np.arctanh(stats_corr[0])
n = len(df.index)
se = 1/(math.sqrt(n - 3))
ci = z + np.array([-1, 1]) * se * stats.norm.ppf((1 + 0.95)/2)

ci = np.tanh(ci)
dl.options.set_pd_options()
ci_table = dl.report.DFBuilder(['Low', 'High'])
ci_table.row([ci[0], ci[1]])
```

5. Bootstrap by resampling pairs with replacement:

```
rs = check_random_state(34)

ranges = []

for j in range(200):
    corrs = []

    for i in range(100):
        indices = rs.choice(n, size=n)
        pairs = df.values
        gen_pairs = pairs[indices]
        corrs.append(stats.pearsonr(gen_pairs.T[0], gen_
pairs.T[1])[0])

    ranges.append(dl.stats.ci(corrs))
```

```

ranges = np.array(ranges)
bootstrap_ci = dl.stats.ci(corr)
ci_table.row([bootstrap_ci[0], bootstrap_ci[1]])
ci_table = ci_table.build(index=['Formula', 'Bootstrap'])

```

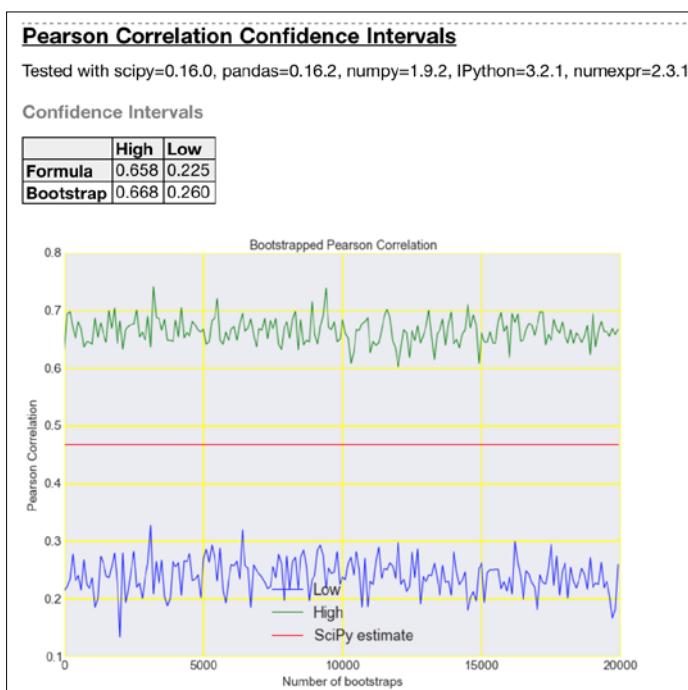
6. Plot the results and produce a report:

```

x = np.arange(len(ranges)) * 100
plt.plot(x, ranges.T[0], label='Low')
plt.plot(x, ranges.T[1], label='High')
plt.plot(x, stats_corr[0] * np.ones_like(x), label='SciPy estimate')
plt.ylabel('Pearson Correlation')
plt.xlabel('Number of bootstraps')
plt.title('Bootstrapped Pearson Correlation')
plt.legend(loc='best')
result = dl.report.HTMLBuilder()
result.h1('Pearson Correlation Confidence intervals')
result.h2('Confidence Intervals')
result.add(ci_table.to_html())
HTML(result.html)

```

Refer to the following screenshot for the end result (see the `correlating_pearson.ipynb` file in this book's code bundle):



See also

- ▶ The related SciPy documentation at <http://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pearsonr.html#scipy.stats.pearsonr> (retrieved August 2015).

Correlating variables with the Spearman rank correlation

The **Spearman rank correlation** uses ranks to correlate two variables with the Pearson Correlation. Ranks are the positions of values in sorted order. Items with equal values get a rank, which is the average of their positions. For instance, if we have two items of equal value assigned position 2 and 3, the rank is 2.5 for both items. Have a look at the following equations:

$$(3.17) \quad \rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}$$

$$(3.18) \quad d_i = x_i - y_i$$

$$(3.19) \quad \sigma = \frac{0.6325}{\sqrt{n-1}}$$

$$(3.20) \quad z = \sqrt{\frac{n-3}{1.06}} F(r)$$

In these equations, n is the sample size. (3.17) shows how the correlation is calculated. (3.19) gives the standard error. (3.20) is about the z-score, which we assume to be normally distributed. $F(r)$ is here the same as in (3.14), since it is the same correlation but applied to ranks.

How to do it...

In this recipe we calculate the Spearman correlation between wind speed and temperature aggregated by the day of the year and the corresponding confidence interval. Then, we display the correlation matrix for all the weather data. The steps are as follows:

1. The imports are as follows:

```
import dautil as dl
from scipy import stats
import numpy as np
import math
```

```

import seaborn as sns
import matplotlib.pyplot as plt
from IPython.html import widgets
from IPython.display import display
from IPython.display import HTML

```

2. Define the following function to compute the confidence interval:

```

def get_ci(n, corr):
    z = math.sqrt((n - 3)/1.06) * np.arctanh(corr)
    se = 0.6325/(math.sqrt(n - 1))
    ci = z + np.array([-1, 1]) * se * stats.norm.ppf((1 + 0.95)/2)

    return np.tanh(ci)

```

3. Load the data and display widgets so that you can correlate a different pair if you want:

```

df = dl.data.Weather.load().dropna()
df = dl.ts.groupby_yday(df).mean()

drop1 = widgets.Dropdown(options=dl.data.Weather.get_headers(),
                        selected_label='TEMP',
                        description='Variable 1')
drop2 = widgets.Dropdown(options=dl.data.Weather.get_headers(),
                        selected_label='WIND_SPEED',
                        description='Variable 2')
display(drop1)
display(drop2)

```

4. Compute the Spearman rank correlation with SciPy:

```

var1 = df[drop1.value].values
var2 = df[drop2.value].values
stats_corr = stats.spearmanr(var1, var2)
dl.options.set_pd_options()
html_builder = dl.report.HTMLBuilder()
html_builder.h1('Spearman Correlation between {0} and {1}'.format(
    dl.data.Weather.get_header(drop1.value), dl.data.Weather.get_
    header(drop2.value)))
html_builder.h2('scipy.stats.spearmanr()')
dfb = dl.report.DFBuilder(['Correlation', 'p-value'])
dfb.row([stats_corr[0], stats_corr[1]])
html_builder.add_df(dfb.build())

```

5. Compute the confidence interval as follows:

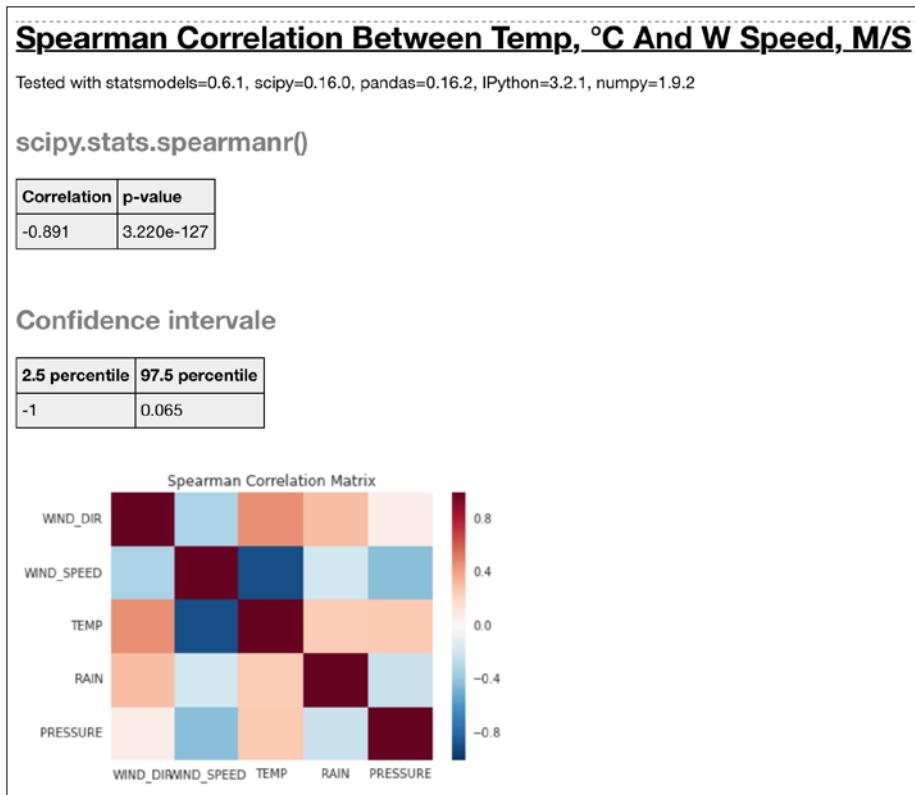
```
n = len(df.index)
ci = get_ci(n, stats_corr)
html_builder.h2('Confidence interval')
dfb = dl.report.DFBuilder(['2.5 percentile', '97.5 percentile'])
dfb.row(ci)
html_builder.add_df(dfb.build())
```

6. Display the correlation matrix as a Seaborn heatmap:

```
corr = df.corr(method='spearman')
```

```
%matplotlib inline
plt.title('Spearman Correlation Matrix')
sns.heatmap(corr)
HTML(html_builder.html)
```

Refer to the following screenshot for the end result (see the `correlating_spearman.ipynb` file in this book's code bundle):



See also

- ▶ The Spearman rank correlation Wikipedia page at https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient (retrieved August 2015)

Correlating a binary and a continuous variable with the point biserial correlation

The **point-biserial correlation** correlates a binary variable Y and a continuous variable X. The coefficient is calculated as follows:

$$(3.21) \quad r_{pb} = \frac{M_1 - M_0}{s_n} \sqrt{\frac{n_1 n_0}{n^2}}$$

$$(3.22) \quad s_n = \sqrt{\frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^2}$$

The subscripts in (3.21) correspond to the two groups of the binary variable. M_1 is the mean of X for values corresponding to group 1 of Y. M_2 is the mean of X for values corresponding to group 0 of Y.

In this recipe, the binary variable we will use is rain or no rain. We will correlate this variable with temperature.

How to do it...

We will calculate the correlation with the `scipy.stats.pointbiserialr()` function. We will also compute the rolling correlation using a 2 year window with the `np.roll()` function. The steps are as follows:

1. The imports are as follows:

```
import dautil as dl
from scipy import stats
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from IPython.display import HTML
```

2. Load the data and correlate the two relevant arrays:

```
df = dl.data.Weather.load().dropna()  
df['RAIN'] = df['RAIN'] > 0  
  
stats_corr = stats.pointbiserialr(df['RAIN'].values, df['TEMP'].values)
```

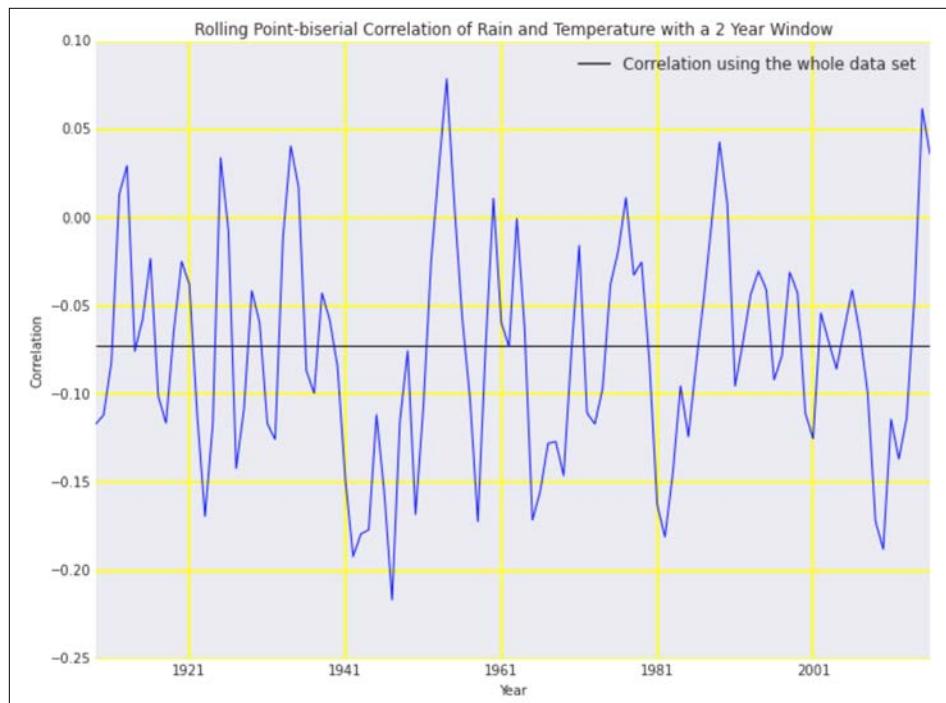
3. Compute the 2 year rolling correlation as follows:

```
N = 2 * 365  
corrs = []  
  
for i in range(len(df.index) - N):  
    x = np.roll(df['RAIN'].values, i)[:N]  
    y = np.roll(df['TEMP'].values, i)[:N]  
    corrs.append(stats.pointbiserialr(x, y)[0])  
  
corrs = pd.DataFrame(corrs,  
                      index=df.index[N:],  
                      columns=['Correlation']).resample('A')
```

4. Plot the results with the following code:

```
plt.plot(corrs.index.values, corrs.values)  
plt.hlines(stats_corr[0], corrs.index.values[0], corrs.index.  
           values[-1],  
           label='Correlation using the whole data set')  
plt.title('Rolling Point-biserial Correlation of Rain and  
Temperature with a 2 Year Window')  
plt.xlabel('Year')  
plt.ylabel('Correlation')  
plt.legend(loc='best')  
HTML(dl.report.HTMLBuilder().watermark())
```

Refer to the following screenshot for the end result (see `correlating_pointbiserial.ipynb` file in this book's code bundle):



See also

- ▶ The relevant SciPy documentation at <http://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pointbiserialr.html#scipy.stats.pointbiserialr> (retrieved August 2015).

Evaluating relations between variables with ANOVA

Analysis of variance (ANOVA) is a statistical data analysis method invented by statistician Ronald Fisher. This method partitions data of a continuous variable using the values of one or more corresponding categorical variables to analyze variance. ANOVA is a form of linear modeling. If we are modeling with one categorical variable, we speak of **one-way ANOVA**. In this recipe, we will use two categorical variables so we have **two-way ANOVA**. In two-way ANOVA, we create a **contingency table**—a table containing counts for all combinations of the two categorical variables (we will see a contingency table example soon). The linear model is then given by the equation:

$$(3.23) \quad \mu_{ij} = \mu + \alpha_i + \beta_j + \gamma_{ij}$$

This is an additive model where μ_{ij} is the mean of the continuous variable corresponding to one cell of the contingency table, μ is the mean for the whole data set, α_i is the contribution of the first categorical variable, β_j is the contribution of the second categorical variable, and y_{ij} is a cross-term. We will apply this model to weather data.

How to do it...

The following steps apply two-way ANOVA to wind speed as continuous variable, rain as a binary variable, and wind direction as categorical variable:

1. The imports are as follows:

```
from statsmodels.formula.api import ols
import dautil as dl
from statsmodels.stats.anova import anova_lm
import seaborn as sns
import matplotlib.pyplot as plt
from IPython.display import HTML
```

2. Load the data and fit the model with statsmodels:

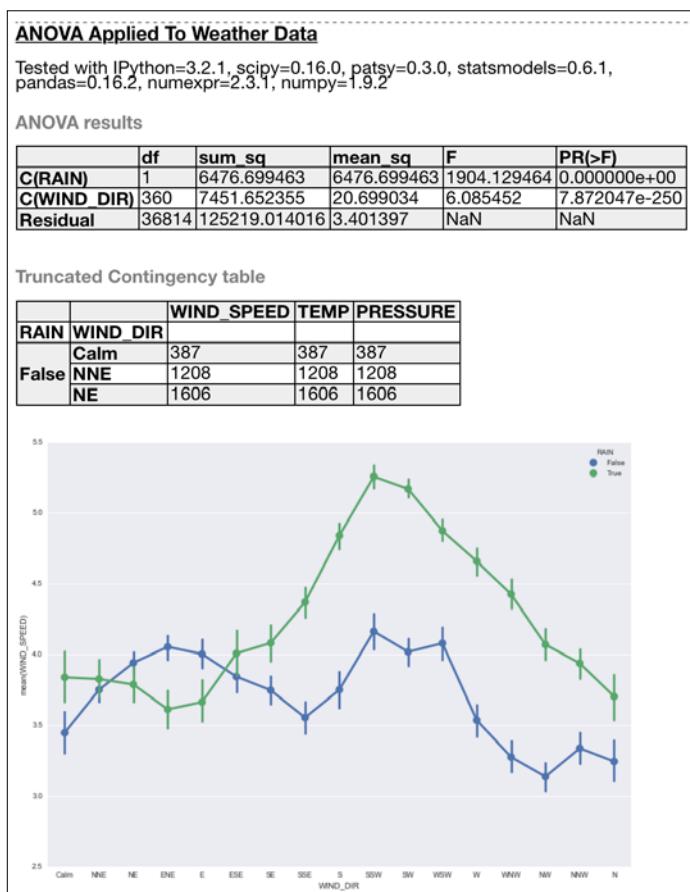
```
df = dl.data.Weather.load().dropna()
df['RAIN'] = df['RAIN'] > 0
formula = 'WIND_SPEED ~ C(RAIN) + C(WIND_DIR)'
lm = ols(formula, df).fit()
hb = dl.HTMLBuilder()
hb.h1('ANOVA Applied to Weather Data')
hb.h2('ANOVA results')
hb.add_df(anova_lm(lm), index=True)
```

3. Display a truncated contingency table and visualize the data with Seaborn:

```
df['WIND_DIR'] = dl.data.Weather.categorize_wind_dir(df)
hb.h2('Truncated Contingency table')
hb.add_df(df.groupby([df['RAIN'], df['WIND_DIR']]).count().
head(3), index=True)

sns.pointplot(y='WIND_SPEED', x='WIND_DIR',
hue='RAIN', data=df[['WIND_SPEED', 'RAIN', 'WIND_
DIR']])
HTML(hb.html)
```

Refer to the following screenshot for the end result (see `anova.ipynb` file in this book's code bundle):



See also

- ▶ The Wikipedia page for two-way ANOVA at https://en.wikipedia.org/wiki/Two-way_analysis_of_variance (retrieved August 2015)
- ▶ The Wikipedia page about the contingency table is https://en.wikipedia.org/wiki/Contingency_table (retrieved August 2015)

4

Dealing with Data and Numerical Issues

The recipes in this chapter are as follows:

- ▶ Clipping and filtering outliers
- ▶ Winsorizing data
- ▶ Measuring central tendency of noisy data
- ▶ Normalizing with the Box-Cox transformation
- ▶ Transforming data with the power ladder
- ▶ Transforming data with logarithms
- ▶ Rebinning data
- ▶ Applying `logit()` to transform proportions
- ▶ Fitting a robust linear model
- ▶ Taking variance into account with weighted least squares
- ▶ Using arbitrary precision for optimization
- ▶ Using arbitrary precision for linear algebra

Introduction

In the real world, data rarely matches textbook definitions and examples. We have to deal with issues such as faulty hardware, uncooperative customers, and disgruntled colleagues. It is difficult to predict what kind of issues you will run into, but it is safe to assume that they will be plentiful and challenging. In this chapter, I will sketch some common approaches to deal with noisy data, which are based more on rules of thumb than strict science. Luckily, the trial and error part of data analysis is limited.

Most of this chapter is about outlier management. Outliers are values that we consider to be abnormal. Of course, this is not the only issue that you will encounter, but it is a sneaky one. A common issue is that of missing or invalid values, so I will briefly mention masked arrays and pandas features such as the `dropna()` function, which I have used throughout this book.

I have also written two recipes about using **mpmath** for arbitrary precision calculations. I don't recommend using mpmath unless you really have to because of the performance penalty you have to pay. Usually we can work around numerical issues, so arbitrary precision libraries are rarely needed.

Clipping and filtering outliers

Outliers are a common issue in data analysis. Although an exact definition of outliers doesn't exist, we know that outliers can influence means and regression results. Outliers are values that are anomalous. Usually, outliers are caused by a measurement error, but the outliers are sometimes real. In the second case, we may be dealing with two or more types of data related to different phenomena.

The data for this recipe is described at <https://vincentarelbundock.github.io/Rdatasets/doc/robustbase/starsCYG.html> (retrieved August 2015). It consists of logarithmic effective temperature and logarithmic light intensity for 47 stars in a certain star cluster. Any astronomers reading this paragraph will know the **Hertzsprung-Russell diagram**. In data analysis terms, the diagram is a scatter plot, but for astronomers, it is of course more than that. The Hertzsprung Russell diagram was defined around 1910 and features a diagonal line (not entirely straight) called the **main sequence**. Most stars in our data set should be on the main sequence with four outliers in the upper-left corner. These outliers are classified as giants.

We have many strategies to deal with outliers. In this recipe, we will use the two simplest strategies: clipping with the NumPy `clip()` function and completely removing the outliers. For this example, I define outliers as values 1.5 interquartile ranges removed from the box defined by the 1st and 3rd quartile.

How to do it...

The following steps show how to clip and filter outliers:

1. The imports are as follows:

```
import statsmodels.api as sm
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import dautil as dl
from IPython.display import HTML
```

-
2. Define the following function to filter outliers:

```
def filter_outliers(a):
    b = a.copy()
    bmin, bmax = dl.stats.outliers(b)
    b[bmin > b] = np.nan
    b[b > bmax] = np.nan

    return b
```

3. Load and clip outliers as follows:

```
starsCYG = sm.datasets.get_rdataset("starsCYG", "robustbase",
cache=True).data

clipped = starsCYG.apply(dl.stats.clip_outliers)
```

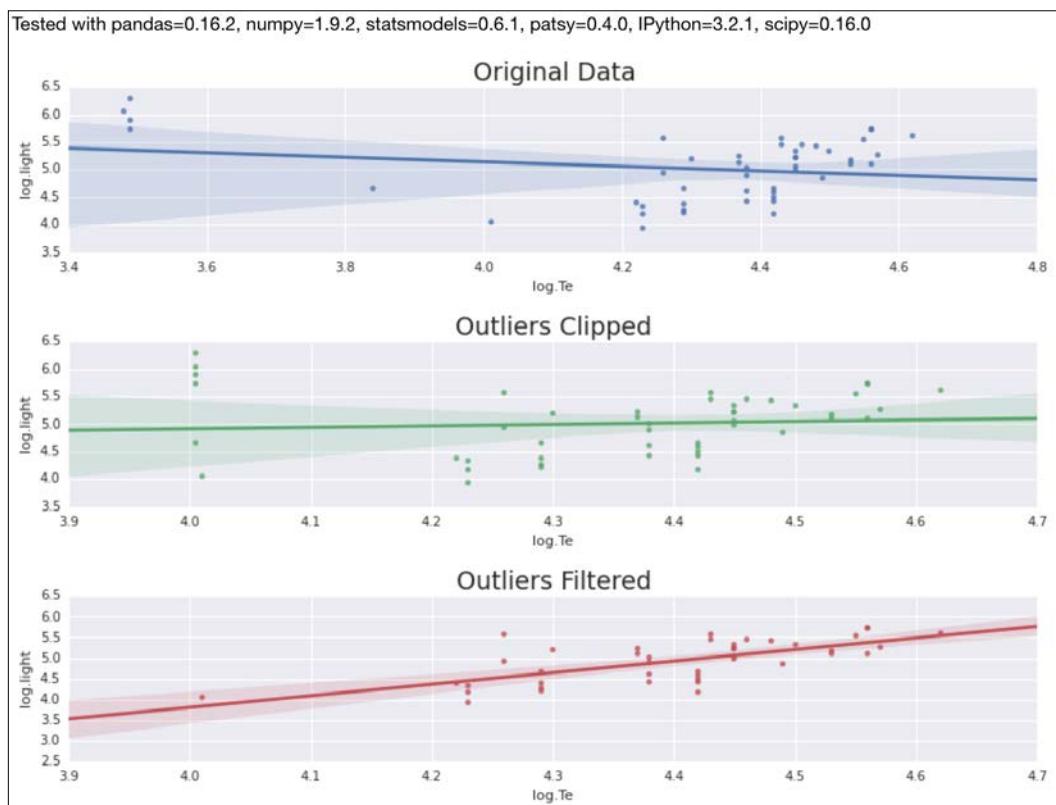
4. Filter outliers as follows:

```
filtered = starsCYG.copy()
filtered['log.Te'] = filter_outliers(filtered['log.Te'].values)
filtered['log.light'] = filter_outliers(filtered['log.light'].values)
filtered.dropna()
```

5. Plot the result with the following code:

```
sp = dl.plotting.Subplotter(3, 1, context)
sp.label()
sns.regplot(x='log.Te', y='log.light', data=starsCYG, ax=sp.ax)
sp.label(advance=True)
sns.regplot(x='log.Te', y='log.light', data=clipped, ax=sp.ax)
sp.label(advance=True)
sns.regplot(x='log.Te', y='log.light', data=filtered, ax=sp.ax)
plt.tight_layout()
HTML(dl.report.HTMLBuilder().watermark())
```

Refer to the following screenshot for the end result (refer to the `outliers.ipynb` file in this book's code bundle):



See also

- ▶ The NumPy `clip()` function documented at <https://docs.scipy.org/doc/numpy/reference/generated/numpy.clip.html#numpy.clip> (retrieved August 2015)
- ▶ You can read more about the Hertzsprung-Russell diagram at https://en.wikipedia.org/wiki/Hertzsprung%E2%80%93Russell_diagram (retrieved August 2015)

Winsorizing data

Winsorizing is another technique to deal with outliers and is named after Charles Winsor. In effect, Winsorization clips outliers to given percentiles in a symmetric fashion. For instance, we can clip to the 5th and 95th percentile. SciPy has a `winsorize()` function, which performs this procedure. The data for this recipe is the same as that for the *Clipping and filtering outliers* recipe.

How to do it...

Winsorize the data with the following procedure:

1. The imports are as follows:

```
rom scipy.stats.mstats import winsorize
import statsmodels.api as sm
import seaborn as sns
import matplotlib.pyplot as plt
import dautil as dl
from IPython.display import HTML
```

2. Load and winsorize the data for the effective temperature (limit is set to 15%):

```
starsCYG = sm.datasets.get_rdataset("starsCYG", "robustbase",
cache=True).data
limit = 0.15
winsorized_x = starsCYG.copy()
winsorized_x['log.Te'] = winsorize(starsCYG['log.Te'],
limits=limit)
```

3. Winsorize the light intensity as follows:

```
winsorized_y = starsCYG.copy()
winsorized_y['log.light'] = winsorize(starsCYG['log.light'],
limits=limit)
winsorized_xy = starsCYG.apply(winsorize, limits=[limit, limit])
```

4. Plot the Hertzsprung-Russell diagram with regression lines (not part of the usual astronomical diagram):

```
sp = dl.plotting.Subplotter(2, 2, context)
sp.label()
sns.regplot(x='log.Te', y='log.light', data=starsCYG, ax=sp.ax)

sp.label(advance=True)
sns.regplot(x='log.Te', y='log.light', data=winsorized_x, ax=sp.
ax)
```

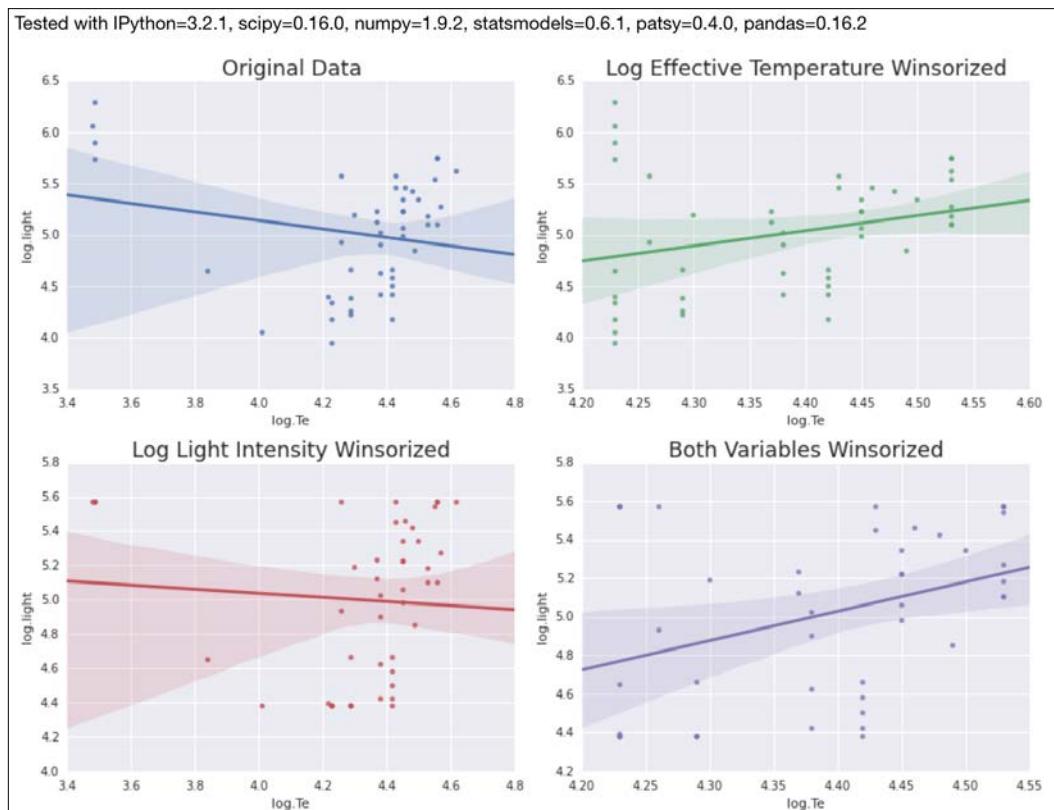
```

sp.label(advance=True)
sns.regplot(x='log.Te', y='log.light', data=winsorized_y, ax=sp.
ax)

sp.label(advance=True)
sns.regplot(x='log.Te', y='log.light', data=winsorized_xy, ax=sp.
ax)
plt.tight_layout()
HTML(dl.report.HTMLBuilder().watermark())

```

Refer to the following screenshot for the end result (refer to the `winsorising_data.ipynb` file in this book's code bundle):



See also

- ▶ The relevant Wikipedia page at <https://en.wikipedia.org/wiki/Winsorising> (retrieved August 2015)

Measuring central tendency of noisy data

We can measure central tendency with the mean and median. These measures use all the data available. It is a generally accepted idea to get rid of outliers by discarding data on the higher and lower end of a data set. The **truncated mean** or **trimmed mean**, and derivatives of it such as the **interquartile mean (IQM)** and **trimean**, use this idea too. Take a look at the following equations:

$$(4.1) \quad TM = \frac{Q_1 + 2Q_2 + Q_3}{4}$$

$$(4.2) \quad x_{IQM} = \frac{2}{n} \sum_{i=\frac{n}{4}+1}^{\frac{3n}{4}} x_i$$

The truncated mean discards the data at given percentiles—for instance, from the lowest value to the 5th percentile and from the 95th percentile to the highest value. The **trimean** (4.1) is a weighted average of the median, first quartile, and third quartile. For the IQM (4.2), we discard the lowest and highest quartile of the data, so it is a special case of the truncated mean. We will calculate these measures with the SciPy `tmean()` and `trima()` functions.

How to do it...

We will take a look at the central tendency for varying levels of truncation with the following steps:

1. The imports are as follows:

```
import matplotlib.pyplot as plt
from scipy.stats import tmean
from scipy.stats.mstats import trima
import numpy as np
import dautil as dl
import seaborn as sns
from IPython.display import HTML

context = dl.nb.Context('central_tendency')
```

2. Define the following function to calculate the interquartile mean:

```
def iqm(a):
    return truncated_mean(a, 25)
```

3. Define the following function to plot distributions:

```
def plotdists(var, ax):
    displot_label = 'From {0} to {1} percentiles'
    cyc = dl.plotting.Cycler()

    for i in range(1, 9, 3):
        limits = dl.stats.outliers(var, method='percentiles',
                                    percentiles=(i, 100 - i))
        truncated = trima(var, limits=limits).compressed()
        sns.distplot(truncated, ax=ax, color=cyc.color(),
                     hist_kws={'histtype': 'stepfilled', 'alpha':
                     1/i,
                     'linewidth': cyc.lw()},
                     label=displot_label.format(i, 100 - i))
```

4. Define the following function to compute the truncated mean:

```
def truncated_mean(a, percentile):
    limits = dl.stats.outliers(a, method='percentiles',
                               percentiles=(percentile, 100 -
                               percentile))

    return tmean(a, limits=limits)
```

5. Load the data and calculate means as follows:

```
df = dl.data.Weather.load().resample('M').dropna()
x = range(9)
temp_means = [truncated_mean(df['TEMP'], i) for i in x]
ws_means = [truncated_mean(df['WIND_SPEED'], i) for i in x]
```

6. Plot the means and distributions with the following code:

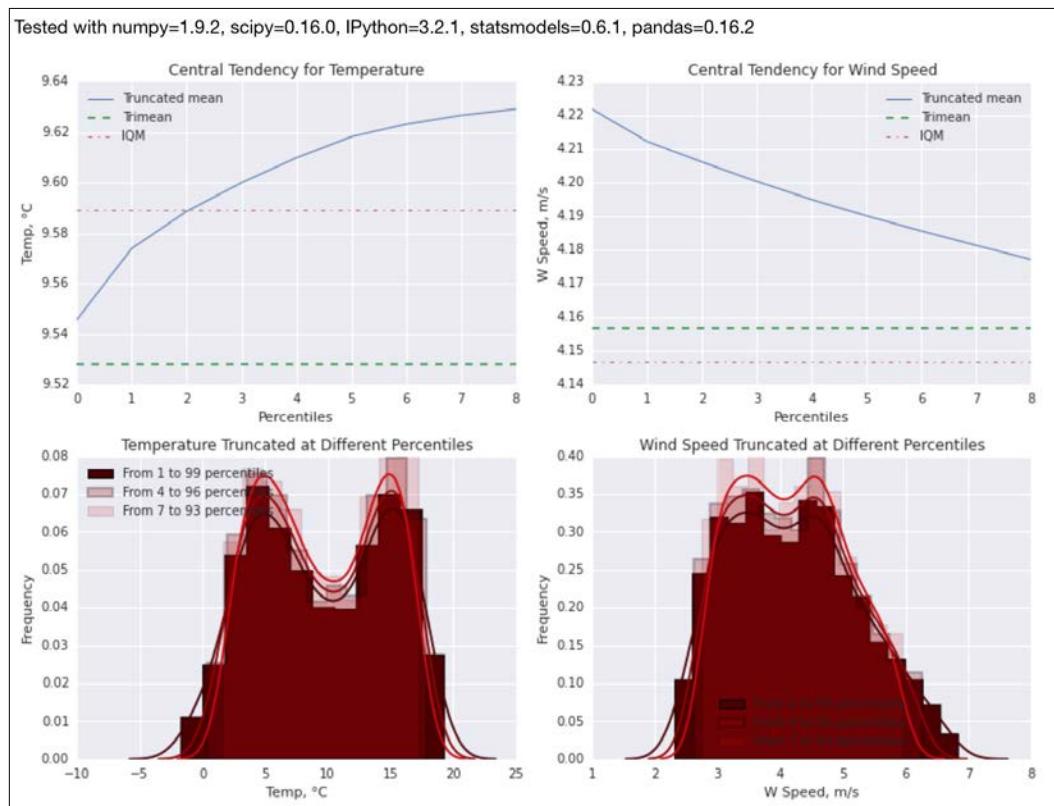
```
sp = dl.plotting.Subplotter(2, 2, context)
cp = dl.plotting.CyclePlotter(sp.ax)
cp.plot(x, temp_means, label='Truncated mean')
cp.plot(x, dl.stats.trimean(df['TEMP']) * np.ones_like(x),
label='Trimean')
cp.plot(x, iqm(df['TEMP']) * np.ones_like(x), label='IQM')
sp.label(ylabel_params=dl.data.Weather.get_header('TEMP'))

cp = dl.plotting.CyclePlotter(sp.next_ax())
cp.plot(x, ws_means, label='Truncated mean')
cp.plot(x, dl.stats.trimean(df['WIND_SPEED']) * np.ones_like(x),
label='Trimean')
cp.plot(x, iqm(df['WIND_SPEED']) * np.ones_like(x), label='IQM')
sp.label(ylabel_params=dl.data.Weather.get_header('WIND_SPEED'))

plotdists(df['TEMP'], sp.next_ax())
```

```
sp.xlabel(xlabel_params=dl.data.Weather.get_header('TEMP'))\n\nplotdists(df['WIND_SPEED'], sp.next_ax())\nsp.xlabel(xlabel_params=dl.data.Weather.get_header('WIND_SPEED'))\nplt.tight_layout()\nHTML(dl.report.HTMLBuilder().watermark())
```

Refer to the following screenshot for the end result (refer to the `central_tendency.ipynb` file in this book's code bundle):



See also

- ▶ The SciPy documentation for `trima()` at <https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mstats.trima.html> (retrieved August 2015)
- ▶ The SciPy documentation for `tmean()` at <https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.tmean.html#scipy.stats.tmean> (retrieved August 2015)

Normalizing with the Box-Cox transformation

Data that doesn't follow a known distribution, such as the normal distribution, is often difficult to manage. A popular strategy to get control of the data is to apply the Box-Cox transformation. It is given by the following equation:

$$(4.3) \quad y_i^{(\lambda)} = \begin{cases} \frac{y_i^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0, \\ \ln(y_i) & \text{if } \lambda = 0, \end{cases}$$

The `scipy.stats.boxcox()` function can apply the transformation for positive data. We will use the same data as in the *Clipping and filtering outliers* recipe. With Q-Q plots, we will show that the Box-Cox transformation does indeed make the data appear more normal.

How to do it...

The following steps show how to normalize data with the Box-Cox transformation:

1. The imports are as follows:

```
import statsmodels.api as sm
import matplotlib.pyplot as plt
from scipy.stats import boxcox
import seaborn as sns
import dutil as dl
from IPython.display import HTML
```

2. Load the data and transform it as follows:

```
context = dl.nb.Context('normalizing_boxcox')

starsCYG = sm.datasets.get_rdataset("starsCYG", "robustbase",
cache=True).data

var = 'log.Te'

# Data must be positive
transformed, _ = boxcox(starsCYG[var])
```

3. Display the Q-Q plots and the distribution plots as follows:

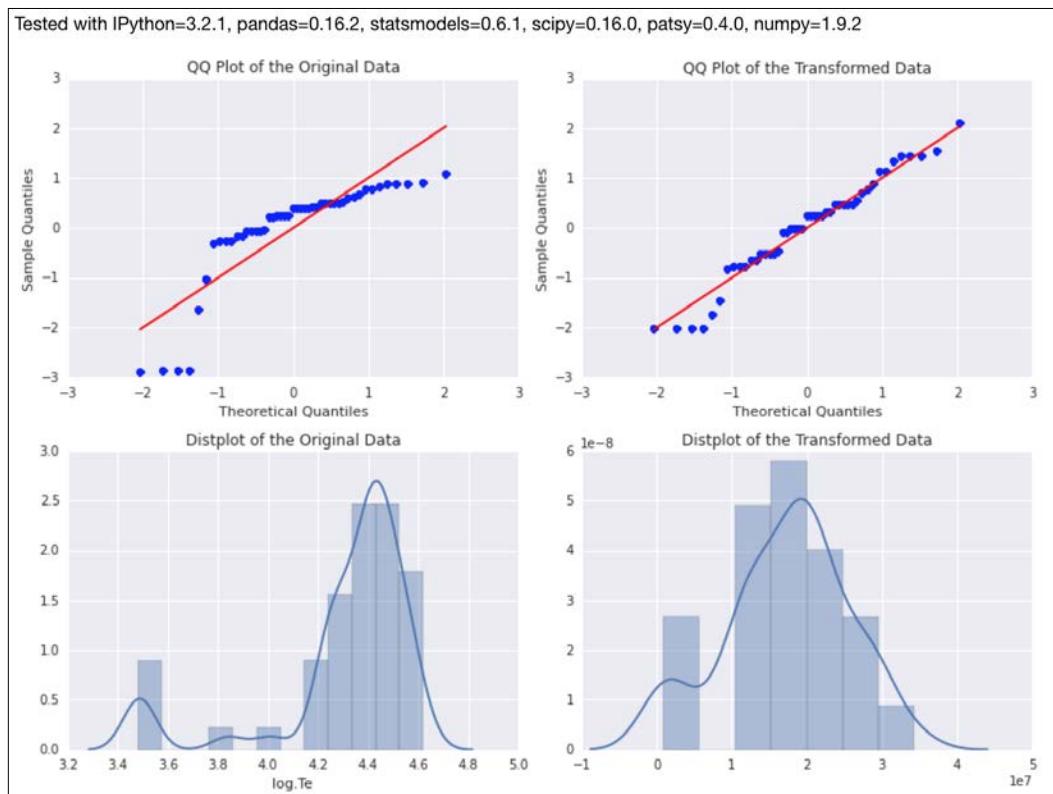
```
sp = dl.plotting.Subplotter(2, 2, context)
sp.label()
sm.qqplot(starsCYG[var], fit=True, line='s', ax=sp.ax)

sp.label(advance=True)
sm.qqplot(transformed, fit=True, line='s', ax=sp.ax)

sp.label(advance=True)
sns.distplot(starsCYG[var], ax=sp.ax)

sp.label(advance=True)
sns.distplot(transformed, ax=sp.ax)
plt.tight_layout()
HTML(dl.report.HTMLBuilder().watermark())
```

Refer to the following screenshot for the end result (refer to the `normalizing_boxcox.ipynb` file in this book's code bundle):



How it works

The Q-Q plots, in the previous screenshot, graph theoretical quantiles for the normal distribution against the quantiles of the actual data. To help evaluate conformance to the normal distribution, I displayed a line that should correspond with perfectly normal data. The more the data fits the line, the more normal it is. As you can see, the transformed data fits the line better and is, therefore, more normal. The distribution plots should help you to confirm this.

See also

- ▶ The relevant Wikipedia page at https://en.wikipedia.org/wiki/Power_transform (retrieved August 2015)
- ▶ G.E.P. Box and D.R. Cox, *An Analysis of Transformations*, Journal of the Royal Statistical Society B, 26, 211-252 (1964).

Transforming data with the power ladder

Linear relations are commonplace in science and data analysis. Obviously, linear models are easier to understand than non-linear models. So historically, tools for linear models were developed first. In certain cases, it pays to linearize (make linear) data to make analysis simpler. A simple strategy that sometimes works is to square or cube one or more variables. Similarly, we can transform the data down an imaginary power ladder by taking the square or cube root.

In this recipe, we will use data from the Duncan dataset as described in <https://vincentarelbundock.github.io/Rdatasets/doc/car/Duncan.html> (retrieved August 2015). The data was gathered around 1961 and is about 45 occupations with four columns—type, income, education, and prestige. We will take a look at income and prestige. These variables seem to be linked by a cubic polynomial, so we can take the cube root of income or the cube of prestige. To check the result, we will visualize the residuals of regression. The expectation is that the residuals are randomly distributed, which means that we don't expect them to follow a recognizable pattern.

How to do it...

In the following steps, I will demonstrate the basic data transformation:

1. The imports are as follows:

```
import matplotlib.pyplot as plt
import numpy as np
import dautil as dl
import seaborn as sns
import statsmodels.api as sm
from IPython.display import HTML
```

2. Load and transform the data as follows:

```
df = sm.datasets.get_rdataset("Duncan", "car", cache=True).data
transformed = df.copy()
transformed['income'] = np.power(transformed['income'], 1.0/3)
```

3. Plot the original data with a Seaborn regression plot (cubic polynomial) as follows:

```
sp = dl.plotting.Subplotter(2, 2, context)
sp.label()
sns.regplot(x='income', y='prestige', data=df, order=3, ax=sp.ax)
```

4. Plot the transformed data with the following lines:

```
sp.label(advance=True)
sns.regplot(x='income', y='prestige', data=transformed, ax=sp.ax)
```

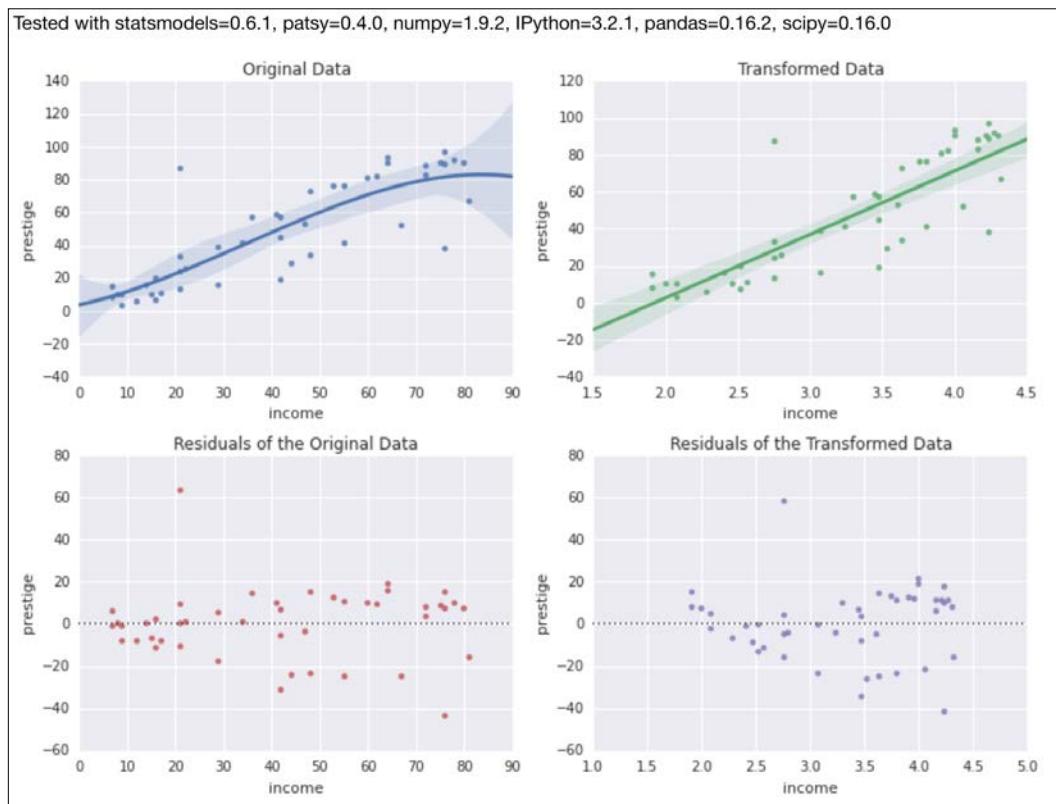
5. Plot the residuals plot for the cubic polynomial:

```
sp.label(advance=True)
sns.residplot(x='income', y='prestige', data=df, order=3, ax=sp.
ax)
```

6. Plot the residuals plot for the transformed data as follows:

```
sp.label(advance=True)
sp.ax.set_xlim([1, 5])
sns.residplot(x='income', y='prestige', data=transformed, ax=sp.
ax)
plt.tight_layout()
HTML(dl.report.HTMLBuilder().watermark())
```

Refer to the following screenshot for the end result (refer to the `transforming_up.ipynb` file in this book's code bundle):



Transforming data with logarithms

When data varies by orders of magnitude, transforming the data with logarithms is an obvious strategy. In my experience, it is less common to do the opposite transformation using an exponential function. Usually when exploring, we visualize a log-log or semi-log scatter plot of paired variables.

To demonstrate this transformation, we will use the Worldbank data for infant mortality rate per 1000 livebirths and **Gross Domestic Product (GDP)** per capita for the available countries. If we apply the logarithm of base 10 to both variables, the slope of the line we get by fitting the data has a useful property. A one percent increase in one variable corresponds to a percentage change given by the slope of the other variable.

How to do it...

Transform the data using logarithms with the following procedure:

1. The imports are as follows:

```
import dautil as dl
import matplotlib.pyplot as plt
import numpy as np
from IPython.display import HTML
```

2. Download the data for 2010 with the following code:

```
wb = dl.data.Worldbank()
countries = wb.get_countries()[['name', 'iso2c']]
inf_mort = wb.get_name('inf_mort')
gdp_pcap = wb.get_name('gdp_pcap')
df = wb.download(country=countries['iso2c'],
                  indicator=[inf_mort, gdp_pcap],
                  start=2010, end=2010).dropna()
```

3. Apply the log transform with the following snippet:

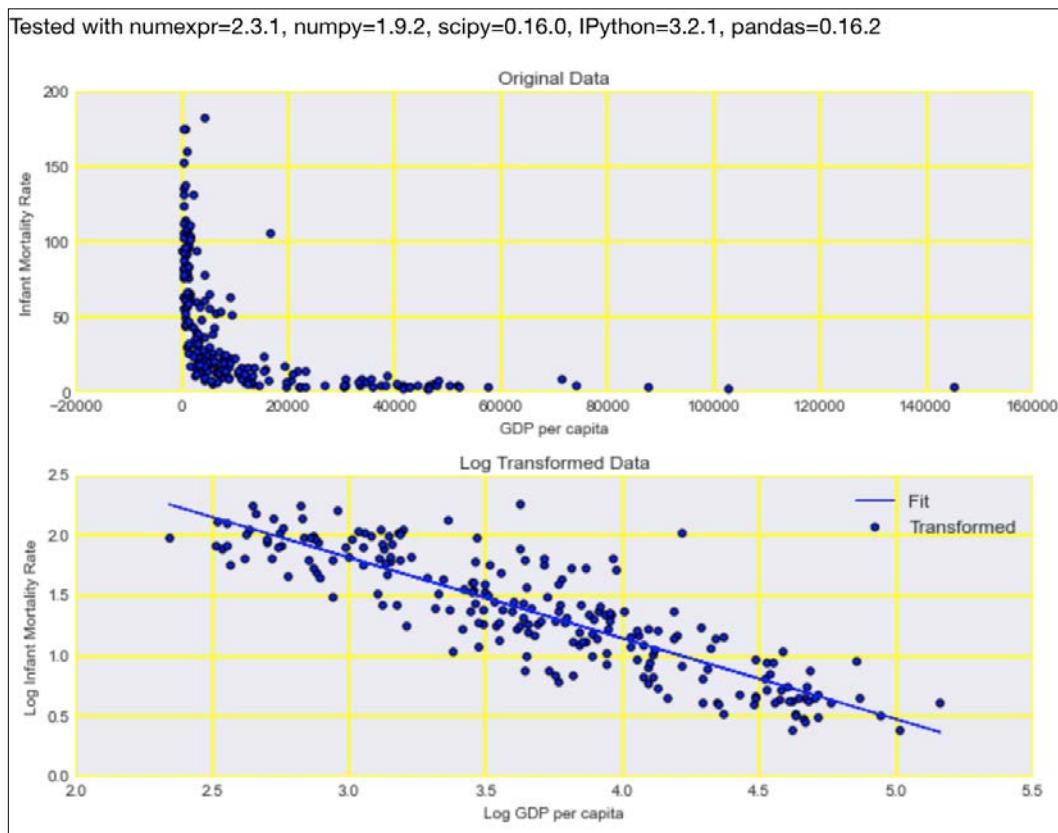
```
loglog = df.applymap(np.log10)
x = loglog[gdp_pcap]
y = loglog[inf_mort]
```

4. Plot the data before and after the transformation:

```
sp = dl.plotting.Subplotter(2, 1, context)
xvar = 'GDP per capita'
sp.label(xlabel_params=xvar)
sp.ax.set_xlim([0, 200])
sp.ax.scatter(df[gdp_pcap], df[inf_mort])

sp.next_ax()
sp.ax.scatter(x, y, label='Transformed')
dl.plotting.plot_polyfit(sp.ax, x, y)
sp.label(xlabel_params=xvar)
plt.tight_layout()
HTML(dl.report.HTMLBuilder().watermark())
```

Refer to the following screenshot for the end result (refer to the `transforming_down.ipynb` file in this book's code bundle):



Rebinning data

Often, the data we have is not structured the way we want to use it. A structuring technique we can use is called (statistical) **data binning** or **bucketing**. This strategy replaces values within an interval (a bin) with one representative value. In the process, we may lose information; however, we gain better control over the data and efficiency.

In the weather dataset, we have wind direction in degrees and wind speed in m/s, which can be represented in a different way. In this recipe, I chose to present wind direction with cardinal directions (north, south, and so on). For the wind speed, I used the Beaufort scale (visit https://en.wikipedia.org/wiki/Beaufort_scale).

How to do it...

Follow these instructions to rebin the data:

1. The imports are as follows:

```
import dautil as dl
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from IPython.display import HTML
```

2. Load and rebin the data as follows (wind direction is in degree 0-360; we rebin to cardinal directions such as north, southwest, and so on):

```
df = dl.data.Weather.load()[['WIND_SPEED', 'WIND_DIR']].dropna()
categorized = df.copy()
categorized['WIND_DIR'] = dl.data.Weather.categorize_wind_dir(df)
categorized['WIND_SPEED'] = dl.data.Weather.beaufort_scale(df)
```

3. Show distributions and countplots with the following code:

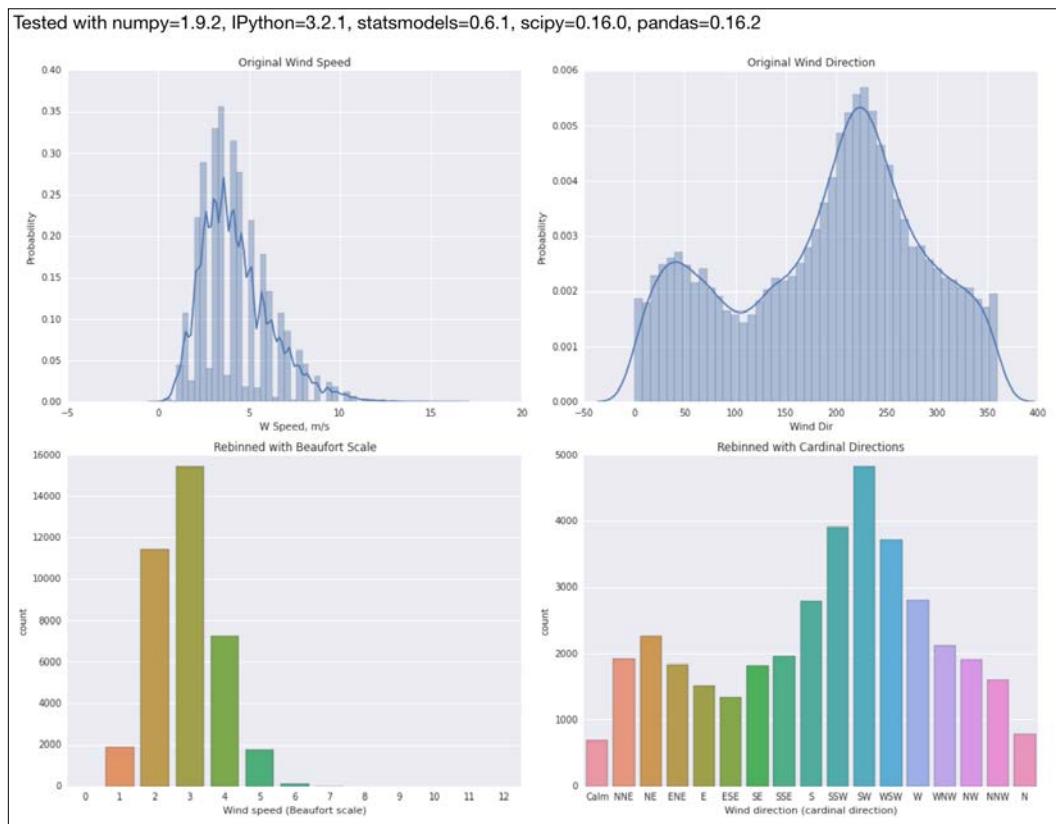
```
sp = dl.plotting.Subplotter(2, 2, context)
sns.distplot(df['WIND_SPEED'], ax=sp.ax)
sp.label(xlabel_params=dl.data.Weather.get_header('WIND_SPEED'))

sns.distplot(df['WIND_DIR'], ax=sp.next_ax())
sp.label(xlabel_params=dl.data.Weather.get_header('WIND_DIR'))

sns.countplot(x='WIND_SPEED', data=categorized, ax=sp.next_ax())
sp.label()

sns.countplot(x='WIND_DIR', data=categorized, ax=sp.next_ax())
sp.label()
plt.tight_layout()
HTML(dl.report.HTMLBuilder().watermark())
```

Refer to the following screenshot for the end result (refer to the `rebinning_data.ipynb` file in this book's code bundle):



Applying logit() to transform proportions

We can transform proportions or ratios with the SciPy `logit()` function. The result should be a more Gaussian distribution. This function is defined by the following equation:

$$(4.4) \quad \text{logit}(p) = \log\left(\frac{p}{1-p}\right) = \log(p) - \log(1-p) = -\log\left(\frac{1}{p} - 1\right)$$

As you can see in equation (4.4), the logit is the logarithm of the odds. What we want to achieve with this transformation is getting a more symmetric distribution—a skew close to zero. As the proportions approach zero and one, the logit asymptotically approaches minus infinity and infinity, so we have to be careful in those cases.

As an example of a proportion, we will take the monthly proportions of rainy days. We get these proportions by turning rain amounts into a binary variable and then averaging over each month.

How to do it...

Transform the ratios by following this guide:

1. The imports are as follows:

```
import dautil as dl
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import math
import statsmodels.api as sm
from scipy.special import logit
from IPython.display import HTML
```

2. Load the data and transform it with the following code:

```
rain = dl.data.Weather.load()['RAIN'].dropna()
rain = rain > 0
rain = rain.resample('M').dropna()
transformed = rain.apply(logit)
transformed = dl.data.dropinf(transformed.values)
```

3. Plot the result of the transformation with distribution plots and Q-Q plots:

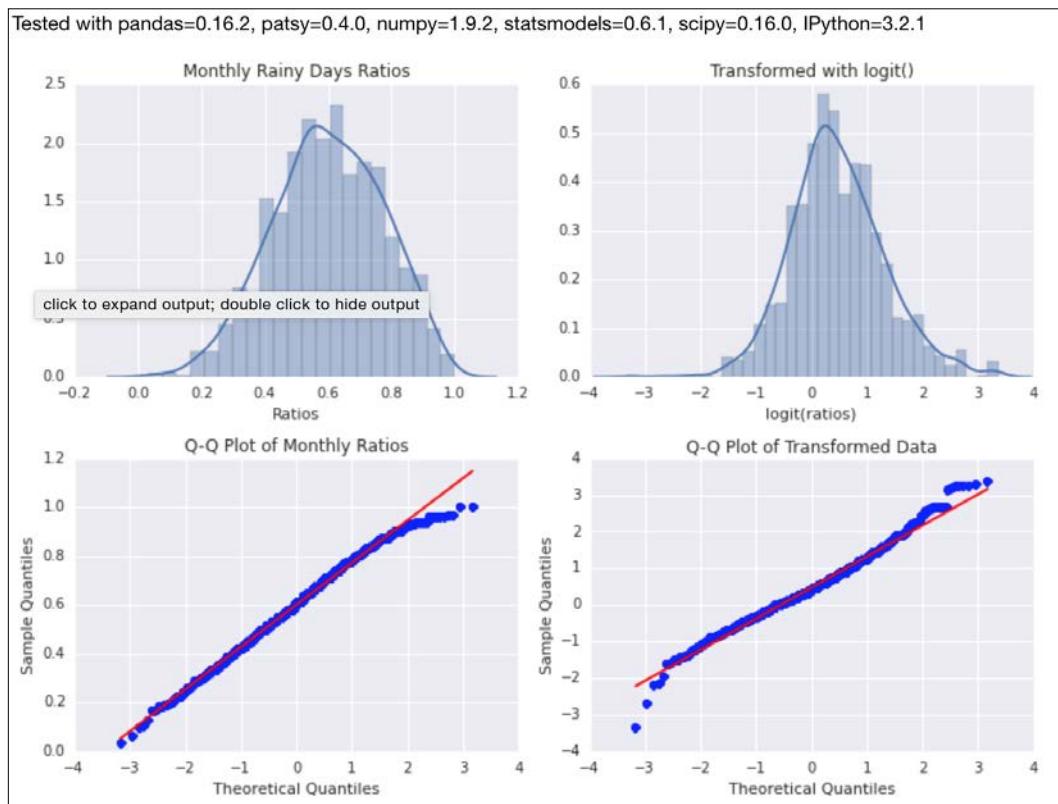
```
sp = dl.plotting.Subplotter(2, 2, context)
sns.distplot(rain, ax=sp.ax)
sp.label()

sp.label(advance=True)
sns.distplot(transformed, ax=sp.ax)

sp.label(advance=True)
sm.qqplot(rain, line='s', ax=sp.ax)

sp.label(advance=True)
sm.qqplot(transformed, line='s', ax=sp.ax)
plt.tight_layout()
HTML(dl.report.HTMLBuilder().watermark())
```

Refer to the following screenshot for the end result (refer to the `transforming_ratios.ipynb` file in this book's code bundle):



Fitting a robust linear model

Robust regression is designed to deal better with outliers in data than ordinary regression. This type of regression uses special robust estimators, which are also supported by statsmodels. Obviously, there is no best estimator, so the choice of estimator depends on the data and the model.

In this recipe, we will fit data about annual sunspot counts available in statsmodels. We will define a simple model where the current count depends linearly on the previous value. To demonstrate the effect of outliers, I added a pretty big value and we will compare the robust regression model and an ordinary least squares model.

How to do it...

The following steps describe how to apply the robust linear model:

1. The imports are as follows:

```
import statsmodels.api as sm
import matplotlib.pyplot as plt
import dautil as dl
from IPython.display import HTML
```

2. Define the following function to set the labels of the plots:

```
def set_labels(ax):
    ax.set_xlabel('Year')
    ax.set_ylabel('Sunactivity')
```

3. Define the following function to plot the model fits:

```
def plot_fit(df, ax, results):
    x = df['YEAR']
    cp = dl.plotting.CyclePlotter(ax)
    cp.plot(x[1:], df['SUNACTIVITY'][1:], label='Data')
    cp.plot(x[2:], results.predict()[1:], label='Fit')
    ax.legend(loc='best')
```

4. Load the data and add an outlier for demonstration purposes:

```
df = sm.datasets.sunspots.load_pandas().data
vals = df['SUNACTIVITY'].values

# Outlier added by malicious person, because noone
# laughs at his jokes.
vals[0] = 100
```

5. Fit the robust model as follows:

```
rlm_model = sm.RLM(vals[1:], sm.add_constant(vals[:-1]),
                    M=sm.robust.norms.TrimmedMean())

rlm_results = rlm_model.fit()
hb = dl.report.HTMLBuilder()
hb.h1('Fitting a robust linear model')
hb.h2('Robust Linear Model')
hb.add(rlm_results.summary().tables[1].as_html())
```

6. Fit an ordinary least squares model:

```
hb.h2('Ordinary Linear Model')
ols_model = sm.OLS(vals[1:], sm.add_constant(vals[:-1]))
ols_results = ols_model.fit()
hb.add(ols_results.summary().tables[1].as_html())
```

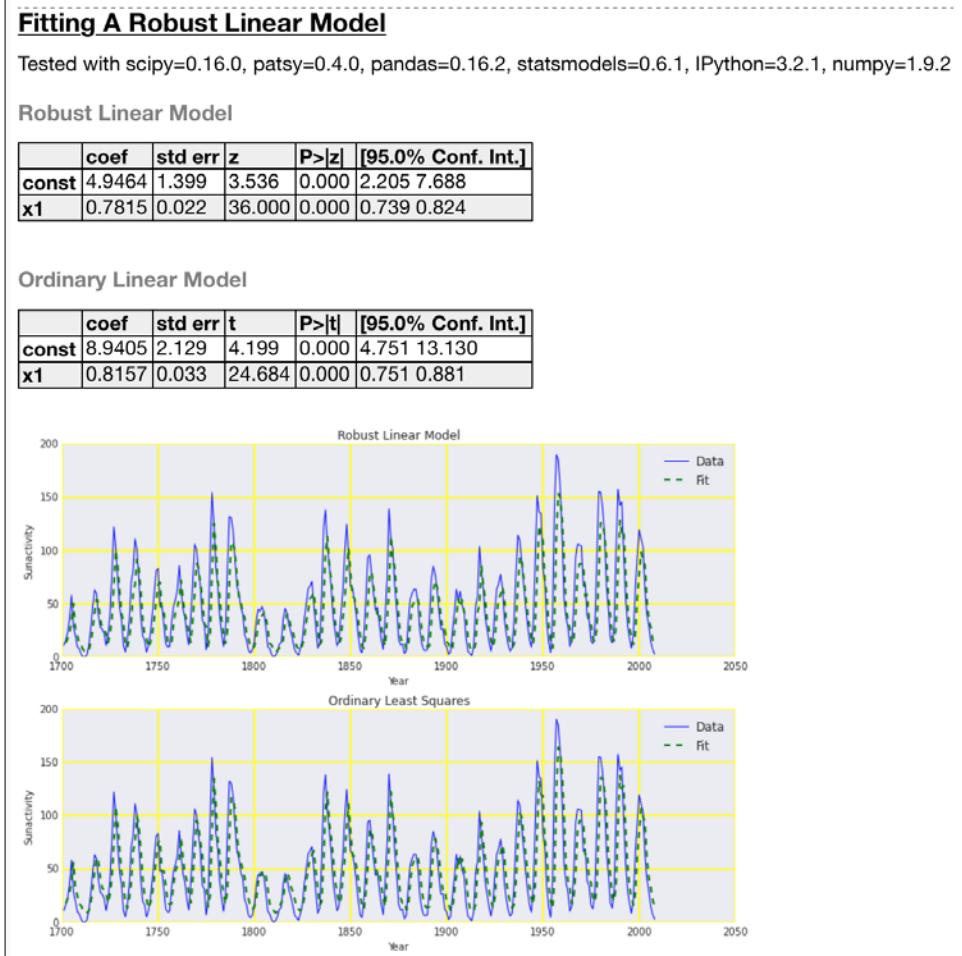
7. Plot the data and the model results with the following code:

```
fig, [ax, ax2] = plt.subplots(2, 1)

plot_fit(df, ax, rlm_results)
ax.set_title('Robust Linear Model')
set_labels(ax)

ax2.set_title('Ordinary Least Squares')
plot_fit(df, ax2, ols_results)
set_labels(ax2)
plt.tight_layout()
HTML(hb.html)
```

Refer to the following screenshot for the end result (refer to the `rlm_demo.ipynb` file in this book's code bundle):



See also

- ▶ The relevant statsmodels documentation at http://statsmodels.sourceforge.net/0.6.0/generated/statsmodels.robust.robust_linear_model.RLM.html (retrieved August 2015)

Taking variance into account with weighted least squares

The statsmodels library allows us to define arbitrary weights per data point for regression. Outliers are sometimes easy to spot with simple rules of thumbs. One of these rules of thumb is based on the interquartile range, which is the difference between the first and third quartile of data. With the interquartile ranges, we can define weights for the **weighted least squares** regression.

We will use the data and model from *Fitting a robust linear mode*, but with arbitrary weights. The points we suspect are outliers will get a lower weight, which is the inverse of the interquartile range values just mentioned.

How to do it...

Fit the data with weighted least squares using the following method:

1. The imports are as follows:

```
import dutil as dl
import matplotlib.pyplot as plt
import statsmodels.api as sm
import numpy as np
from IPython.display import HTML
```

2. Load the data and add an outlier:

```
temp = dl.data.Weather.load() ['TEMP'].dropna()
temp = dl.ts.groupby_yday(temp).mean()

# Outlier added by malicious person, because noone
# laughs at his jokes.
temp.values[0] = 100
```

3. Fit using an ordinary least squares model:

```
ntemp = len(temp)
x = np.arange(1, ntemp + 1)
factor = 2 * np.pi/365.25
```

```
cos_x = sm.add_constant(np.cos(-factor * x - factor * 337))
ols_model = sm.OLS(temp, cos_x)
ols_results = ols_model.fit()
hb = dl.report.HTMLBuilder()
hb.h1('Taking variance into account with weighted least squares')
hb.h2('Ordinary least squares')
hb.add(ols_results.summary().tables[1].as_html())
ols_preds = ols_results.predict()
```

4. Compute weights using interquartile ranges and fit the weighted least squares model:

```
box = dl.stats.Box(temp)
iqrs = box.iqr_from_box()
# Adding 1 to avoid div by 0
weights = 1. / (iqrs + 1)
wls_model = sm.WLS(temp, cos_x, weights=weights)
wls_results = wls_model.fit()

hb.h2('Weighted least squares')
hb.add(wls_results.summary().tables[1].as_html())
```

5. Plot the model results and weights:

```
sp = dl.plotting.Subplotter(2, 2, context)

sp.ax.plot(x[1:], temp[1:], 'o', label='Data')
sp.ax.plot(x[1:], ols_preds[1:], label='Fit')
sp.label(ylabel_params=dl.data.Weather.get_header('TEMP'))

sp.label(advance=True)
sp.ax.plot(x, iqrs, 'o')

sp.next_ax().plot(x[1:], temp[1:], 'o', label='Data')
sp.ax.plot(x[1:], wls_results.predict()[1:], label='Fit')
sp.label(ylabel_params=dl.data.Weather.get_header('TEMP'))

sp.label(advance=True)
sp.ax.plot(x, weights, 'o')
plt.tight_layout()
HTML(hb.html)
```

Refer to the following screenshot for the end result (refer to the `weighted_ls.ipynb` file in this book's code bundle):

Taking Variance Into Account With Weighted Least Squares

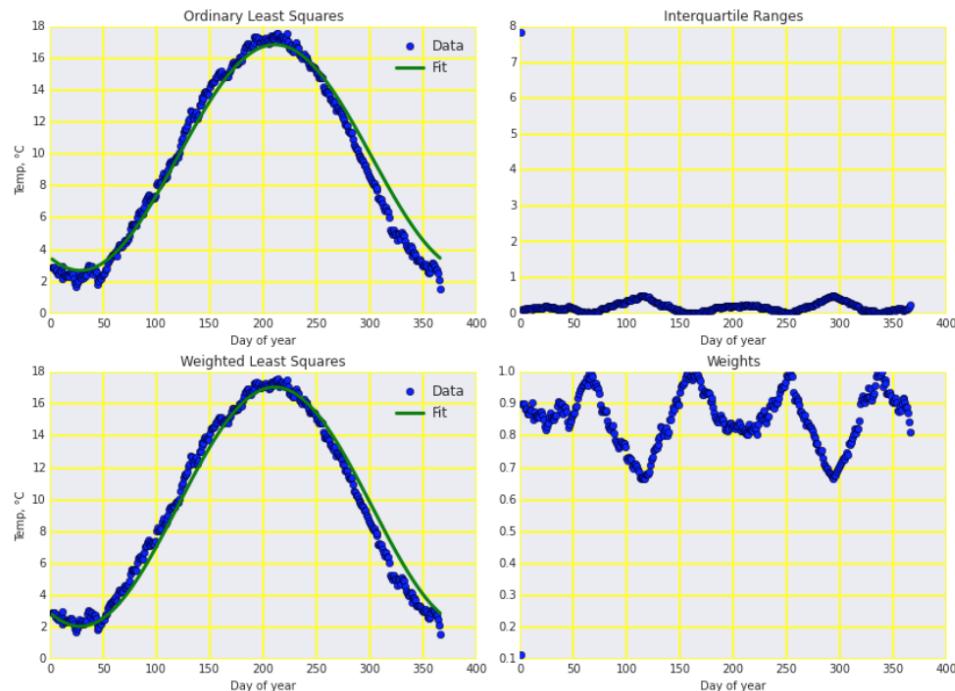
Tested with statsmodels=0.6.1, patsy=0.4.0, numpy=1.9.2, IPython=3.2.1, scipy=0.16.0, pandas=0.16.2

Ordinary least squares

	coef	std err	t	P> t	[95.0% Conf. Int.]
const	9.7902	0.269	36.433	0.000	9.262 10.319
x1	-7.0908	0.380	-18.670	0.000	-7.838 -6.344

Weighted least squares

	coef	std err	t	P> t	[95.0% Conf. Int.]
const	9.5604	0.105	90.637	0.000	9.353 9.768
x1	-7.5123	0.147	-51.103	0.000	-7.801 -7.223



See also

- ▶ The relevant statsmodels documentation at http://statsmodels.sourceforge.net/0.5.0/examples/generated/example_wls.html (retrieved August 2015).

Using arbitrary precision for optimization

The intended readers of this book should be aware of floating point number issues. I will remind you that we are not able to represent floating point numbers exactly. Even integer representation is limited. For certain applications, for instance financial calculations or work involving known analytic expressions, we need a higher precision than available with numerical software such as NumPy. The Python standard library provides the `Decimal` class, which we can use to achieve arbitrary precision. However, the specialized `mpmath` library is a better fit for more advanced use.

Temperature follows a seasonal pattern, so a model involving the cosine seems natural. We will apply such a model. The nice thing about using arbitrary precision is that you can easily do analysis, differentiate, find roots, and approximate polynomials.

Getting ready

Install the specialized `mpmath` library with either of the following commands:

```
$ conda install mpmath  
$ pip install mpmath
```

I tested the code with `mpmath` 0.19 via Anaconda.

How to do it...

The following instructions describe how to use arbitrary precision for optimization:

1. The imports are as follows:

```
import numpy as np  
import matplotlib.pyplot as plt  
import mpmath  
import dautil as dl  
from IPython.display import HTML
```

2. Define the following functions for the model and the first derivative:

```
def model(t):
    mu, C, w, phi = (9.6848106, -7.59870042, -0.01766333,
-5.83349705)

    return mu + C * mpmath.cos(w * t + phi)

def diff_model(t):
    return mpmath.diff(model, t)
```

3. Load the data and find the root of the first derivative:

```
vals = dl.data.Weather.load()['TEMP'].dropna()
vals = dl.ts.groupby_yday(vals).mean()
diff_root = mpmath.findroot(diff_model, (1, 366),
solver='anderson')
```

4. Get a polynomial approximation for the model as follows:

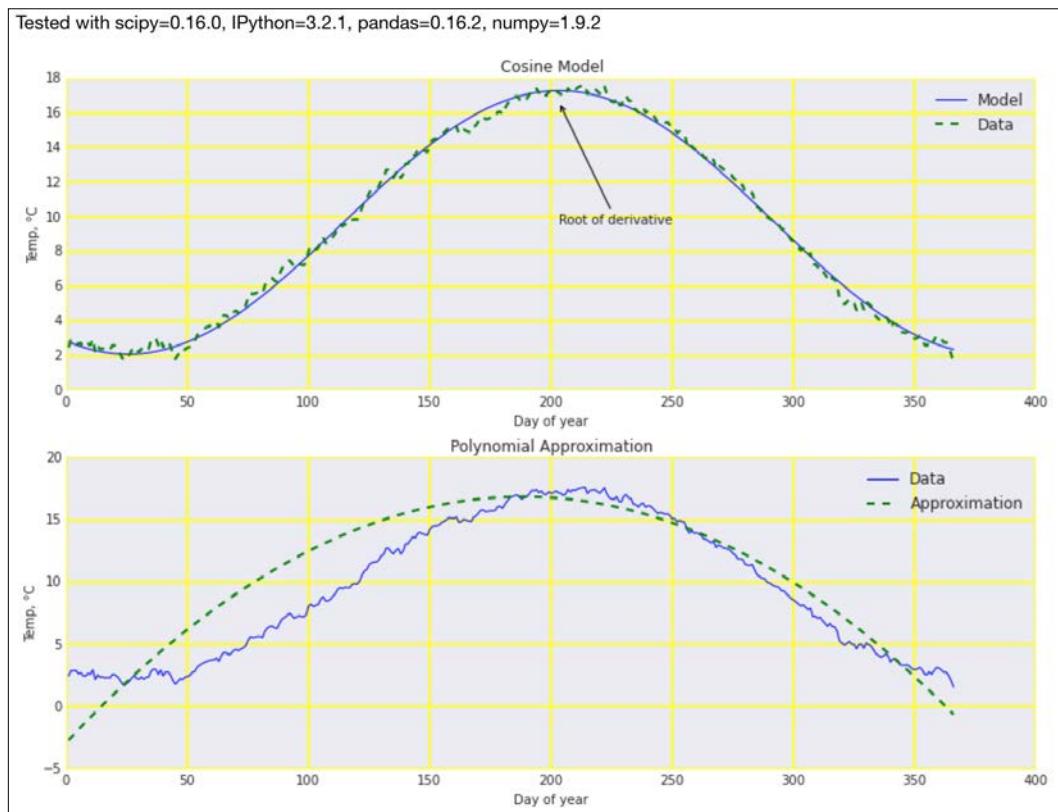
```
days = range(1, 367)
poly = mpmath.chebyfit(model, (1, 366), 3)
poly = np.array([float(c) for c in poly])
```

5. Plot the data, model results, and approximation with the following code:

```
sp = dl.plotting.Subplotter(2, 1, context)
cp = dl.plotting.CyclePlotter(sp.ax)
cp.plot(days, [model(i) for i in days], label='Model')
cp.plot(days, vals, label='Data')
sp.ax.annotate(s='Root of derivative', xy=(diff_root, vals.max() -
1),
xytext=(diff_root, vals.max() - 8),
arrowprops=dict(arrowstyle='->'))
yvar = dl.data.Weather.get_header('TEMP')
sp.label(ylabel_params=yvar)

cp = dl.plotting.CyclePlotter(sp.next_ax())
cp.plot(days, vals, label='Data')
cp.plot(days, np.polyval(poly, days), label='Approximation')
sp.label(ylabel_params=yvar)
plt.tight_layout()
HTML(dl.report.HTMLBuilder().watermark())
```

Refer to the following screenshot for the end result (refer to the `mpmath_fit.ipynb` file in this book's code bundle):



See also

- ▶ The documentation for the `chebyfit()` function at <https://mpmath.readthedocs.org/en/latest/calculus/approximation.html#mpmath.chebyfit> (retrieved August 2015)
- ▶ The documentation for the `findroot()` function at <https://mpmath.readthedocs.org/en/latest/calculus/optimization.html> (retrieved August 2015)

Using arbitrary precision for linear algebra

A lot of models can be reduced to systems of linear equations, which are the domain of linear algebra. The mpmath library mentioned in the *Using arbitrary precision for optimization* recipe can do arbitrary precision linear algebra too.

Theoretically, we can approximate any differentiable function as a polynomial series. To find the coefficients of the polynomial, we can define a system of linear equations, basically taking powers of a data vector (vector as mathematical term) and using a vector of ones to represent the constant in the polynomial. We will solve such a system with the mpmath `lu_solve()` function. As example data, we will use wind speed data grouped by the day of year.

Getting ready

For the relevant instructions, refer to the *Using arbitrary precision for optimization* recipe.

How to do it...

Follow these steps to use arbitrary precision for linear algebra:

1. The imports are as follows:

```
import mpmath
import dautil as dl
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import HTML
```

2. Define the following function to compute the arithmetic mean with mpmath:

```
def mpmean(arr):
    mpfs = [mpmath.mpf(a) for a in arr]

    return sum(mpfs)/len(arr)
```

3. Load the data and solve the system with `lu_solve()`:

```
vals = dl.data.Weather.load() ['WIND_SPEED'].dropna()
vals = dl.ts.groupby_yday(vals).apply(mpmean)

days = np.arange(1, 367, dtype=int)
A = [[], [], []]
A[0] = np.ones_like(days, dtype=int).tolist()
A[1] = days.tolist()
```

```
A[2] = (days ** 2).tolist()
A = mpmath.matrix(A).transpose()

params = mpmath.lu_solve(A, vals)

result = dl.report.HTMLBuilder()
result.h1('Arbitrary Precision Linear Algebra')
result.h2('Polynomial fit')
dfb = dl.report.DFBuilder(['Coefficient 0', 'Coefficient 1',
                           'Coefficient 2'])
dfb.row(params)
result.add_df(dfb.build())
```

4. Define the following function to evaluate the polynomial we got:

```
def poly(x):
    return mpmath.polyval(params[::-1], x)
```

5. Use the fourier() function to get a trigonometric approximation:

```
cs = mpmath.fourier(poly, days.tolist(), 1)
result.h2('Cosine and sine terms')
dfb = dl.report.DFBuilder(['Coefficient 1', 'Coefficient 2'])
dfb.row(cs[0])
dfb.row(cs[1])
result.add_df(dfb.build(index=['Cosine', 'Sine']), index=True)
```

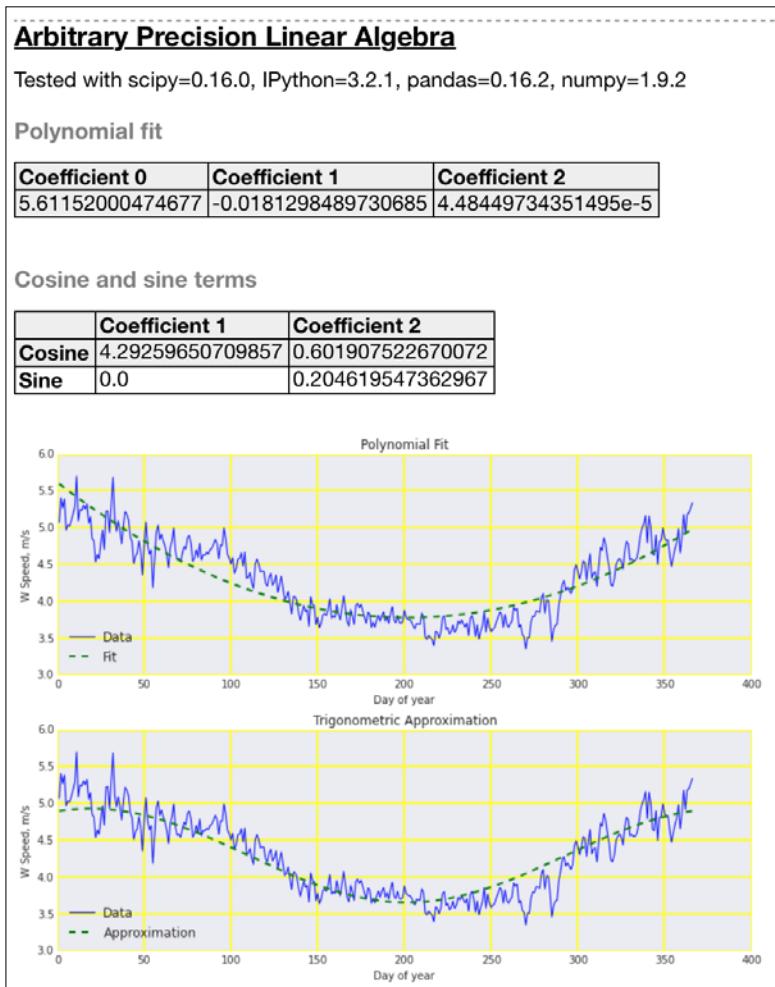
6. Plot the data, model results, and approximation as follows:

```
sp = dl.plotting.Subplotter(2, 1, context)

cp = dl.plotting.CyclePlotter(sp.ax)
cp.plot(days, vals, label='Data')
cp.plot(days, poly(days), label='Fit')
yvar = dl.data.Weather.get_header('WIND_SPEED')
sp.label(ylabel_params=yvar)

cp = dl.plotting.CyclePlotter(sp.next_ax())
cp.plot(days, vals, label='Data')
cp.plot(days, [mpmath.fourierval(cs, days, d) for d in days],
        label='Approximation')
sp.label(ylabel_params=yvar)
plt.tight_layout()
HTML(result.html)
```

Refer to the following screenshot for the end result (refer to the `mpmath_linalg.ipynb` file in this book's code bundle):



See also

- ▶ The `fourier()` function documented at <https://mpmath.readthedocs.org/en/latest/calculus/approximation.html?highlight=fourier#mpmath.fourier> (retrieved August 2015)
- ▶ The `lu_solve()` function documented at https://mpmath.readthedocs.org/en/latest/matrices.html?highlight=lu_solve (retrieved August 2015)

5

Web Mining, Databases, and Big Data

On the menu for this chapter are the following recipes:

- ▶ Simulating web browsing
- ▶ Scraping the Web
- ▶ Dealing with non-ASCII text and HTML entities
- ▶ Implementing association tables
- ▶ Setting up database migration scripts
- ▶ Adding a table column to an existing table
- ▶ Adding indices after table creation
- ▶ Setting up a test web server
- ▶ Implementing a star schema with fact and dimension tables
- ▶ Using HDFS
- ▶ Setting up Spark
- ▶ Clustering data with Spark

Introduction

This chapter is light on math, but it is more focused on technical topics. Technology has a lot to offer for data analysts. Databases have been around for a while, but the relational databases that most people are familiar with can be traced back to the 1970s. Edgar Codd came up with a number of ideas that later led to the creation of the relational model and SQL. Relational databases have been a dominant technology since then. In the 1980s, object-oriented programming languages caused a paradigm shift and an unfortunate mismatch with relational databases.

Object-oriented programming languages support concepts such as inheritance, which relational databases and SQL do not support (of course with some exceptions). The Python ecosystem has several **object-relational mapping (ORM)** frameworks that try to solve this mismatch issue. It is not possible and is unnecessary to cover them all, so I chose SQLAlchemy for the recipes here. We will also have a look at database schema migration as a common hot topic, especially for production systems.

Big data is one of the buzzwords that you may have heard of. Hadoop and Spark may probably also sound familiar. We will look at these frameworks in this chapter. If you use my Docker image, you will unfortunately not find Selenium, Hadoop, and Spark in there because I decided not to include them to save space.

Another important technological development is the World Wide Web, also known as the Internet. The Internet is the ultimate data source; however, getting this data in an easy-to-analyze form is sometimes quite a challenge. As a last resource, we may have to crawl and scrape web pages. Success is not guaranteed because the website owner can change the content without warning us. It is up to you to keep the code of the web scraping recipes up to date.

Simulating web browsing

Corporate websites are usually made by teams or departments using specialized tools and templates. A lot of the content is generated on the fly and consists of a large part of JavaScript and CSS. This means that even if we download the content, we still have to, at least, evaluate the JavaScript code. One way that we can do this from a Python program is using the **Selenium** API. Selenium's main purpose is actually testing websites, but nothing stops us from using it to scrape websites.

Instead of scraping a website, we will scrape an IPython Notebook—the `test_widget.ipynb` file in this book's code bundle. To simulate browsing this web page, we provided a unit test class in `test_simulating_browsing.py`. In case you wondered, this is not the recommended way to test IPython Notebooks.

For historic reasons, I prefer using XPath to find HTML elements. XPath is a query language, which also works with HTML. This is not the only method, you can also use CSS selectors, tag names, or IDs. To find the right XPath expression, you can either install a relevant plugin for your favorite browser, or for instance in Google Chrome, you can inspect an element's XPath.

Getting ready

Install Selenium with the following command:

```
$ pip install selenium
```

I tested the code with Selenium 2.47.1.

How to do it...

The following steps show you how to simulate web browsing using an IPython widget that I made. The code for this recipe is in the `test_simulating_browsing.py` file in this book's code bundle:

1. The first step is to run the following:

```
$ ipython notebook
```

2. The imports are as follows:

```
from selenium import webdriver
import time
import unittest
import dautil as dl

NAP_SECS = 10
```

3. Define the following function, which creates a Firefox browser instance:

```
class SeleniumTest(unittest.TestCase):
    def setUp(self):
        self.logger = dl.log_api.conf_logger(__name__)
        self.browser = webdriver.Firefox()
```

4. Define the following function to clean up when the test is done:

```
def tearDown(self):
    self.browser.quit()
```

-
5. The following function clicks on the widget tabs (we have to wait for the user interface to respond):

```
def wait_and_click(self, toggle, text):
    xpath = "//a[@data-toggle='{0}' and contains(text(), '{1}')]"
    xpath = xpath.format(toggle, text)
    elem = dl.web.wait_browser(self.browser, xpath)
    elem.click()
```

6. Define the following function, which performs the test that consists of evaluating the notebook cells and clicking on a couple of tabs in the IPython widget (we use port 8888):

```
def test_widget(self):
    self.browser.implicitly_wait(NAP_SECS)
    self.browser.get('http://localhost:8888/notebooks/test_
widget.ipynb')

    try:
        # Cell menu
        xpath = '//*[@id="menus"]/div/div/ul/li[5]/a'
        link = dl.web.wait_browser(self.browser, xpath)
        link.click()
        time.sleep(1)

        # Run all
        xpath = '//*[@id="run_all_cells"]/a'
        link = dl.web.wait_browser(self.browser, xpath)
        link.click()
        time.sleep(1)

        self.wait_and_click('tab', 'Figure')
        self.wait_and_click('collapse', 'figure.figsize')
    except Exception:
        self.logger.warning('Error while waiting to click',
exc_info=True)
        self.browser.quit()

    time.sleep(NAP_SECS)
    self.browser.save_screenshot('widgets_screenshot.png')

if __name__ == "__main__":
    unittest.main()
```

The following screenshot is created by the code:

The screenshot shows a Jupyter Notebook interface. At the top, there's a menu bar with File, Edit, View, Insert, Cell, Kernel, Help, and a toolbar with various icons. The status bar indicates "Python 3". Below the toolbar, a code cell contains the following Python code:

```
In [1]: import dutil as dl
context = dl.nb.Context('test')
dl.nb.RcWidget(context)
```

Below the code cell is a configuration panel titled "RcParams". It has tabs for Axes, Figure, Font, Grid, and Lines. The "Axes" tab is selected, showing the "edgecolor" parameter. A preview window displays three horizontal bars labeled "Red", "Green", and "Blue" with their respective colors and a value of 0 at the end. Other tabs show "facecolor", "labelcolor", "linewidth", and "titlesize" with their current values set to black.

See also

- ▶ The Selenium documentation is at <https://selenium-python.readthedocs.org/en/latest/installation.html> (retrieved September 2015)
- ▶ The Wikipedia page about XPath is at <https://en.wikipedia.org/wiki/XPath> (retrieved September 2015)

Scraping the Web

We know that search engines send out autonomous programs called bots to find information on the Internet. Usually, this leads to the creation of giant indices similar to a phonebook or a dictionary. The current situation (September 2015) for Python 3 users is not ideal when it comes to scraping the Web. Most frameworks only support Python 2. However, Guido van Rossum, **Benevolent Dictator for Life (BDFL)** has just contributed a crawler on GitHub that uses the `asyncio` API. All hail the BDFL!

I forked the repository and made small changes in order to save crawled URLs. I also made the crawler exit early. These changes are not very elegant, but this was all I could do in a limited time frame. Anyway, I can't hope to do better than the BDfL himself.

Once we have a list of web links, we will load these webpages from Selenium (refer to the *Simulating web browsing* recipe). I chose PhantomJS, a headless browser, which should have a lighter footprint than Firefox. Although this is not strictly necessary, I think that it makes sense to sometimes download the web pages you are scraping, because you then can test scraping locally. You can also change the links in the downloaded HTML to point to local files. This is related to the *Setting up a test web server* recipe. A common use case of scraping is to create a text corpus for linguistic analysis. This is our goal in this recipe.

Getting ready

Install Selenium as described in the *Simulating web browsing* recipe. I use PhantomJS in this recipe, but this is not a hard requirement. You can use any other browser supported by Selenium. My modifications are under the 0.0.1 tag at <https://github.com/ivanidris/500lines/releases> (retrieved September 2015). Download one of the source archives and unpack it. Navigate to the crawler directory and its code subdirectory.

Start (optional step) the crawler with the following command (I used CNN as an example):

```
$ python crawl.py edition.cnn.com
```

How to do it...

You can use the CSV file with links in this book's code bundle or make your own as I explained in the previous section. The following procedure describes how to create a text corpus of news articles (refer to the `download_html.py` file in this book's code bundle):

1. The imports are as follows:

```
import dautil as dl
import csv
import os
from selenium import webdriver
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.by import By
import urllib.parse as urlparse
import urllib.request as urlrequest
```

2. Define the following global constants:

```
LOGGER = dl.log_api.conf_logger('download_html')
DRIVER = webdriver.PhantomJS()
NAP_SECONDS = 10
```

3. Define the following function to extract text from a HTML page and save it:

```

def write_text(fname):
    elems = []

    try:
        DRIVER.get(dl.web.path2url(fname))

        elems = WebDriverWait(DRIVER, NAP_SECONDS).until(
            EC.presence_of_all_elements_located((By.XPATH, '//p'))
        )

        LOGGER.info('Elems', elems)

        with open(fname.replace('.html', '_phantomjs.html'), 'w') as pjs_file:
            LOGGER.warning('Writing to %s', pjs_file.name)
            pjs_file.write(DRIVER.page_source)

    except Exception:
        LOGGER.error("Error processing HTML", exc_info=True)

    new_name = fname.replace('html', 'txt')

    if not os.path.exists(new_name):
        with open(new_name, 'w') as txt_file:
            LOGGER.warning('Writing to %s', txt_file.name)

            lines = [e.text for e in elems]
            LOGGER.info('lines', lines)
            txt_file.write('\n'.join(lines))

```

4. Define the following main() function, which reads the CSV file with links and calls the functions in the previous steps:

```

def main():
    filedir = os.path.join(dl.data.get_data_dir(), 'edition.cnn.com')

    with open('saved_urls.csv') as csvfile:
        reader = csv.reader(csvfile)

        for line in reader:
            timestamp, count, basename, url = line

```

```
fname = '_'.join([count, basename])
fname = os.path.join(filedir, fname)

if not os.path.exists(fname):
    dl.data.download(url, fname)

write_text(fname)

if __name__ == '__main__':
    DRIVER.implicitly_wait(NAP_SECONDS)
    main()
    DRIVER.quit()
```

Dealing with non-ASCII text and HTML entities

HTML is not as structured as data from a database query or a pandas DataFrame. You may be tempted to manipulate HTML with regular expressions or string functions. However, this approach works only in a limited number of cases. You are better off using specialized Python libraries to process HTML. In this recipe, we will use the `clean_html()` function of the `lxml` library. This function strips all JavaScript and CSS from a HTML page.

American Standard Code for Information Interchange (ASCII) was the dominant encoding standard on the Internet until the end of 2007 with UTF-8 (8-bit Unicode) taking over first place. ASCII is limited to the English alphabet and has no support for alphabets of different languages. Unicode has a much broader support for alphabets. However, we sometimes need to limit ourselves to ASCII, so this recipe gives you an example of how to ignore non-ASCII characters.

Getting ready

Install `lxml` with pip or conda, as follows:

```
$ pip install lxml
$ conda install lxml
```

I tested the code with `lxml 3.4.2` from Anaconda.

How to do it...

The code is in the `processing_html.py` file in this book's code bundle and is broken up in the following steps:

1. The imports are as follows:

```
from lxml.html.clean import clean_html
from difflib import Differ
import unicodedata
import dautil as dl

PRINT = dl.log_api.Printer()
```

2. Define the following function to diff two files:

```
def diff_files(text, cleaned):
    d = Differ()
    diff = list(d.compare(text.splitlines(keepends=True),
                          cleaned.splitlines(keepends=True)))
    PRINT.print(diff)
```

3. The following code block opens a HTML file, cleans it, and compares the cleaned file with the original:

```
with open('460_cc_phantomjs.html') as html_file:
    text = html_file.read()
    cleaned = clean_html(text)
    diff_files(text, cleaned)
    PRINT.print(dl.web.find_hrefs(cleaned))
```

4. The following snippet demonstrates handling of non-ASCII text:

```
bulgarian = 'ПИТОН is Bulgarian for Python'
PRINT.print('Bulgarian', bulgarian)
PRINT.print('Bulgarian ignored', unicodedata.normalize('NFKD',
bulgarian).encode('ascii', 'ignore'))
```

Refer to the following screenshot for the end result (I omitted some of the output for brevity):

```
'- </body></html>',
'+ \n',
'+ \n',
'+ </body></div>']

['',
'http://hraunfoss.fcc.gov/edocs_public/attachmatch/FCC-12-9A1.doc',
'mailto:IPClosedCaptioning@turner.com']

'Bulgarian'
'ПИТОН is Bulgarian for Python'

'Bulgarian ignored'
'b' is Bulgarian for Python'
```

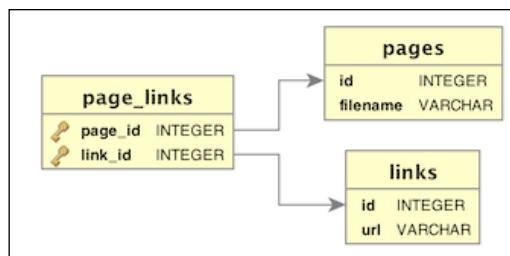
See also

- ▶ The lxml documentation is at <http://lxml.de/index.html> (retrieved September 2015)

Implementing association tables

The association table acts as a bridge between database tables, which have a many-to-many relationship. The table contains foreign keys that are linked to the primary keys of the tables it connects.

In this recipe, we will associate web pages with links within the page. A page has many links, and links can be in many pages. We will concern ourselves only with links to other websites, but this is not a requirement. If you are trying to reproduce a website on your local machine for testing or analysis, you will want to store image and JavaScript links as well. Have a look at the following relational schema diagram:



Getting ready

I installed SQLAlchemy 0.9.9 with Anaconda, as follows:

```
$ conda install sqlalchemy
```

If you prefer, you can also install SQLAlchemy with the following command:

```
$ pip install sqlalchemy
```

How to do it...

The following code from the `impl_association.py` file in this book's code bundle implements the association table pattern:

1. The imports are as follows:

```
from sqlalchemy import create_engine
from sqlalchemy import Column
```

```

from sqlalchemy import ForeignKey
from sqlalchemy import Integer
from sqlalchemy import String
from sqlalchemy import Table
from sqlalchemy.orm import backref
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from sqlalchemy.exc import IntegrityError
import dautil as dl
import os

Base = declarative_base()

```

2. Define the following class to represent a web page:

```

class Page(Base):
    __tablename__ = 'pages'
    id = Column(Integer, primary_key=True)
    filename = Column(String, nullable=False, unique=True)
    links = relationship('Link', secondary='page_links')

    def __repr__(self):
        return "Id=%d filename=%s" %(self.id, self.filename)

```

3. Define the following class to represent a web link:

```

class Link(Base):
    __tablename__ = 'links'
    id = Column(Integer, primary_key=True)
    url = Column(String, nullable=False, unique=True)

    def __repr__(self):
        return "Id=%d url=%s" %(self.id, self.url)

```

4. Define the following class to represent the association between pages and links:

```

class PageLink(Base):
    __tablename__ = 'page_links'
    page_id = Column(Integer, ForeignKey('pages.id'), primary_
key=True)
    link_id = Column(Integer, ForeignKey('links.id'), primary_
key=True)
    page = relationship('Page', backref=backref('link_assoc'))
    link = relationship('Link', backref=backref('page_assoc'))

    def __repr__(self):
        return "page_id=%s link_id=%s" %(self.page_id, self.link_
id)

```

-
5. Define the following function to go through HTML files and update the related tables:

```
def process_file(fname, session):
    with open(fname) as html_file:
        text = html_file.read()

    if dl.db.count_where(session, Page.filename, fname):
        # Cowardly refusing to continue
        return

    page = Page(filename=fname)
    hrefs = dl.web.find_hrefs(text)

    for href in set(hrefs):
        # Only saving http links
        if href.startswith('http'):
            if dl.db.count_where(session, Link.url, href):
                continue

            link = Link(url=href)
            session.add(PageLink(page=page, link=link))

    session.commit()
```

6. Define the following function to populate the database:

```
def populate():
    dir = dl.data.get_data_dir()
    path = os.path.join(dir, 'crawled_pages.db')
    engine = create_engine('sqlite:///{} + path')
    DBSession = sessionmaker(bind=engine)
    Base.metadata.create_all(engine)
    session = DBSession()

    files = ['460_cc_phantomjs.html', '468_live_phantomjs.html']

    for file in files:
        process_file(file, session)

    return session
```

7. The following code snippet uses the functions and classes that we defined:

```
if __name__ == "__main__":
    session = populate()
    printer = dl.log_api.Printer(nelems=3)
```

```
pages = session.query(Page).all()
printer.print('Pages', pages)

links = session.query(Link).all()
printer.print('Links', links)

page_links = session.query(PageLink).all()
printer.print('PageLinks', page_links)
```

Refer to the following screenshot for the end result:

```
'Pages'
[Id=1 filename=460_cc_phantomjs.html, Id=2 filename=468_live_phantomjs.html]

'Links'
[Id=1 url=http://hraunfoss.fcc.gov/edocs_public/attachmatch/FCC-12-9A1.doc,
 '',
 '...',
 Id=4 url=http://www.cnn.com/feedback/forms/form1.html?43,
 '',
 Id=7 url=http://www.macromedia.com/support/documentation/en/fla-
shplayer/help/settings_manager03.html]

'PageLinks'
[page_id=1 link_id=1, '...', page_id=2 link_id=4, '...', page_id=2 link_id=7]
```

Setting up database migration scripts

One of the first things that you learn in programming classes is that nobody can get a complex program right the very first time. Software evolves over time, and we hope for the best. Automation in automated testing helps ensure that our programs improve over time. However, when it comes to evolving database schemas, automation doesn't seem to be so obvious. Especially in large enterprises, database schemas are the domain of database administrators and specialists. Of course, there are security and operational issues related to changing schemas, even more so in production databases. In any case, you can always implement database migration in your local test environment and document proposed changes for the production team.

We will use Alembic to demonstrate how you can go about setting up migration scripts. In my opinion, Alembic is the right tool for the job, although it is in beta as of September 2015.

Getting ready

Install Alembic with the following command:

```
$ pip install alembic
```

Alembic depends on SQLAlchemy, and it will automatically install SQLAlchemy if needed. You should now have the `alembic` command in your path. I used Alembic 0.8.2 for this chapter.

How to do it...

The following steps describe how to set up Alembic migration steps. When we run the Alembic initialization script in a directory, it creates an `alembic` directory and a configuration file named `alembic.ini`:

1. Navigate to the appropriate directory and initialize the migration project, as follows:

```
$ alembic init alembic
```
2. Edit the `alembic.ini` file as required. For instance, change the `sqlalchemy.url` property to point to the correct database.

See also

- ▶ The relevant Alembic documentation is at <https://alembic.readthedocs.org/en/latest/tutorial.html> (retrieved September 2015)

Adding a table column to an existing table

If we use an object-relational mapper (ORM), such as SQLAlchemy, we map classes to tables and class attributes to table columns. Often, due to new business requirements, we need to add a table column and corresponding class attribute. We will probably need to populate the column immediately after adding it.

If we deal with a production database, then probably you do not have direct access. Luckily, we can generate SQL with Alembic, which a database administrator can review.

Getting ready

Refer to the *Setting up database migration scripts* recipe.

How to do it...

Alembic has its own versioning system, which requires extra tables. It also creates a `versions` directory under the `alembic` directory with generated Python code files. We need to specify the types of change necessary for migration in these files:

1. Create a new revision, as follows:

```
$ alembic revision -m "Add a column"
```

2. Open the generated Python file (for instance, `27218d73000_add_a_column.py`). Replace the two functions in there with the following code, which adds the `link_type` String column:

```
def upgrade():
    # MODIFIED Ivan Idris
    op.add_column('links', sa.Column('link_type', sa.String(20)))

def downgrade():
    # MODIFIED Ivan Idris
    op.drop_column('links', 'link_type')
```

3. Generate SQL, as follows:

```
$ alembic upgrade head --sql
```

Refer to the following screenshot for the end result:

```
INFO [alembic.runtime.migration] Context impl SQLiteImpl.
INFO [alembic.runtime.migration] Generating static SQL
INFO [alembic.runtime.migration] Will assume non-transactional
DDL.
CREATE TABLE alembic_version (
    version_num VARCHAR(32) NOT NULL
);

INFO [alembic.runtime.migration] Running upgrade  -> 27218d7300
0, Add a column
-- Running upgrade  -> 27218d73000

ALTER TABLE links ADD COLUMN link_type VARCHAR(20);

INSERT INTO alembic_version (version_num) VALUES ('27218d73000')
;
```

Adding indices after table creation

Indices are a general concept in computing. This book also has an index for faster lookup, which matches concepts to page numbers. An index takes up space; in the case of this book, a couple of pages. Database indices have the added disadvantage that they make inserts and updates slower because of the extra overhead of updating the index. Usually, primary and foreign keys automatically get an index, but this depends on the database implementation.

Adding indices should not be taken lightly, and this is best done after consulting database administrators. Alembic has features for index addition similar to the features that we saw in the *Adding a table column to an existing table* recipe.

Getting ready

Refer to the *Setting up database migration scripts* recipe.

How to do it...

This recipe has some overlap with the *Adding a table column to an existing table* recipe, so I will not repeat all the details:

1. Create a new revision, as follows:

```
$ alembic revision -m "Add indices"
```

2. Open the generated Python file (for instance, 21579ecccd8_add_indices.py) and modify the code to have the following functions, which take care of adding indices:

```
def upgrade():
    # MODIFIED Ivan Idris
    op.create_index('idx_links_url', 'links', ['url'])
    op.create_index('idx_pages_filename', 'pages', ['filename'])

def downgrade():
    # MODIFIED Ivan Idris
    op.drop_index('idx_links_url')
    op.drop_index('idx_pages_filename')
```

3. Generate SQL, as follows:

```
$ alembic upgrade head --sql
```

Refer to the following screenshot for the end result:

```

INFO [alembic.runtime.migration] Context impl SQLiteImpl.
INFO [alembic.runtime.migration] Generating static SQL
INFO [alembic.runtime.migration] Will assume non-transactional
DDL.
CREATE TABLE alembic_version (
    version_num VARCHAR(32) NOT NULL
);

INFO [alembic.runtime.migration] Running upgrade  -> 21579ecccd8, Add indices
-- Running upgrade  -> 21579ecccd8

CREATE INDEX idx_links_url ON links (url);

CREATE INDEX idx_pages_filename ON pages (filename);

INSERT INTO alembic_version (version_num) VALUES ('21579ecccd8')
;

```

How it works...

The `create_index()` function adds indices given an index name, a table, and a list of table columns. The `drop_index()` function does the opposite, removing indices given an index name.

See also

- ▶ The Wikipedia page about database indices is at https://en.wikipedia.org/wiki/Database_index (retrieved September 2015)

Setting up a test web server

In *Chapter 1, Laying the Foundation for Reproducible Data Analysis*, we discussed why unit testing is a good idea. Purists will tell you that you only need unit tests. However, the general consensus is that higher-level testing can also be useful.

Obviously, this book is about data analysis and not about web development. Still, sharing your results or data via a website or web service is a common requirement. When you mine the Web or do something else related to the Web, it often becomes necessary to reproduce certain use cases, such as login forms. As you expect of a mature language, Python has many great web frameworks. I chose Flask, a simple Pythonic web framework for this recipe because it seemed easy to set up, but you should use your own judgment because I have no idea what your requirements are.

Getting ready

I tested the code with Flask 0.10.1 from Anaconda. Install Flask with `conda` or `pip`, as follows:

```
$ conda install flask  
$ pip install flask
```

How to do it...

In this recipe, we will set up a secure page with a login form, which you can use for testing. The code consists of a `app.py` Python file and a HTML file in the `templates` directory (I will not discuss the HTML in detail):

1. The imports are as follows:

```
from flask import Flask  
from flask import render_template  
from flask import request  
from flask import redirect  
from flask import url_for  
  
app = Flask(__name__)
```

2. Define the following function to handle requests for the home page:

```
@app.route('/')  
def home():  
    return "Test Site"
```

3. Define the following function to process login attempts:

```
@app.route('/secure', methods=['GET', 'POST'])  
def login():  
    error = None  
    if request.method == 'POST':  
        if request.form['username'] != 'admin' or\  
            request.form['password'] != 'admin':  
            error = 'Invalid password or user name.'  
        else:  
            return redirect(url_for('home'))  
    return render_template('admin.html', error=error)
```

4. The following block runs the server (don't use debug=True for public-facing websites):

```
if __name__ == '__main__':
    app.run(debug=True)
```

5. Run \$ python app.py and open a web browser at http://127.0.0.1:5000/ and http://127.0.0.1:5000/secure.

Refer to the following screenshot for the end result:

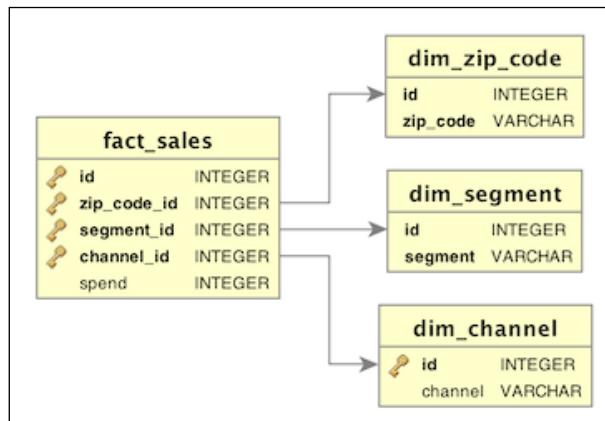


Implementing a star schema with fact and dimension tables

The **star schema** is a database pattern that facilitates reporting. Star schemas are appropriate for the processing of events, such as website visits, ad clicks, or financial transactions. Event information (metrics, such as temperature or purchase amount) is stored in fact tables linked to much smaller dimension tables. Star schemas are denormalized, which places the responsibility of integrity checks to the application code. For this reason, we should only write to the database in a controlled manner. If you use SQLAlchemy for bulk inserts, you should choose the Core API over the ORM API or use straight SQL. You can read more about the reasons at http://docs.sqlalchemy.org/en/rel_1_0/faq/performance.html (retrieved September 2015).

Time is a common dimension in reporting. For instance, we can store dates of daily weather measurements in a dimension table. For each date in our data, we can save the date, year, month, and day of year. We can prepopulate this table before processing events and then add new dates as needed. We don't even have to add new records to the time dimension table if we assume that we only need to maintain the database for a century. In such a case, we will just prepopulate the time dimension table with all the possible dates in the range that we want to support. If we are dealing with binary or categorical variables, pre-populating the dimension tables should be possible too.

In this recipe, we will implement a star schema for direct marketing data described in <http://blog.minethatdata.com/2008/03/minethatdata-e-mail-analytics-and-data.html> (retrieved September 2015). The data is in a CSV file from a direct marketing campaign. For the sake of simplicity, we will ignore some of the columns. As a metric, we will take the `spend` column with purchase amounts. For the dimensions, I chose the channel (Phone, Web, or Multichannel), the zip code (Rural, Suburban, or Urban) and segment (Mens, Womens, or no e-mail). Refer to the following entity-relationship diagram:



How to do it...

The following code downloads the data, loads it in a database, and then queries the database (refer to the `star_schema.py` file in this book's code bundle):

1. The imports are as follows:

```

from sqlalchemy import Column
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import ForeignKey
from sqlalchemy import Integer
from sqlalchemy import String
from sqlalchemy.orm import sessionmaker
from sqlalchemy import func
import dautil as dl
from tabulate import tabulate
import sqlite3
import os
from joblib import Memory

Base = declarative_base()
memory = Memory(cachedir='.')
  
```

2. Define the following class to represent the ZIP code dimension:

```
class DimZipCode(Base):
    __tablename__ = 'dim_zip_code'
    id = Column(Integer, primary_key=True)
    # Urban, Suburban, or Rural.
    zip_code = Column(String(8), nullable=False, unique=True)
```

3. Define the following class to represent the segment dimension:

```
class DimSegment(Base):
    __tablename__ = 'dim_segment'
    id = Column(Integer, primary_key=True)
    # Mens E-Mail, Womens E-Mail or No E-Mail
    segment = Column(String(14), nullable=False, unique=True)
```

4. Define the following class to represent the channel dimension:

```
class DimChannel(Base):
    __tablename__ = 'dim_channel'
    id = Column(Integer, primary_key=True)
    channel = Column(String)
```

5. Define the following class to represent the fact table:

```
class FactSales(Base):
    __tablename__ = 'fact_sales'
    id = Column(Integer, primary_key=True)
    zip_code_id = Column(Integer, ForeignKey('dim_zip_code.id'),
                          primary_key=True)
    segment_id = Column(Integer, ForeignKey('dim_segment.id'),
                         primary_key=True)
    channel_id = Column(Integer, ForeignKey('dim_channel.id'),
                         primary_key=True)

    # Storing amount as cents
    spend = Column(Integer)

    def __repr__(self):
        return "zip_code_id={0} channel_id={1} segment_id={2}".
format(
            self.zip_code_id, self.channel_id, self.segment_id)
```

6. Define the following function to create a SQLAlchemy session:

```
def create_session(dbname):
    engine = create_engine('sqlite:///{}'.format(dbname))
    DBSession = sessionmaker(bind=engine)
    Base.metadata.create_all(engine)

    return DBSession()
```

-
7. Define the following function to populate the segment dimension table:

```
def populate_dim_segment(session):
    options = ['Mens E-Mail', 'Womens E-Mail', 'No E-Mail']

    for option in options:
        if not dl.db.count_where(session, DimSegment.segment,
option):
            session.add(DimSegment(segment=option))

    session.commit()
```

8. Define the following function to populate the ZIP code dimension table:

```
def populate_dim_zip_code(session):
    # Note the interesting spelling
    options = ['Urban', 'Surburban', 'Rural']

    for option in options:
        if not dl.db.count_where(session, DimZipCode.zip_code,
option):
            session.add(DimZipCode(zip_code=option))

    session.commit()
```

9. Define the following function to populate the channel dimension table:

```
def populate_dim_channels(session):
    options = ['Phone', 'Web', 'Multichannel']

    for option in options:
        if not dl.db.count_where(session, DimChannel.channel,
option):
            session.add(DimChannel(channel=option))

    session.commit()
```

10. Define the following function to populate the fact table (it uses straight SQL for performance reasons):

```
def load(csv_rows, session, dbname):
    channels = dl.db.map_to_id(session, DimChannel.channel)
    segments = dl.db.map_to_id(session, DimSegment.segment)
    zip_codes = dl.db.map_to_id(session, DimZipCode.zip_code)
    conn = sqlite3.connect(dbname)
    c = conn.cursor()
```

```

logger = dl.log_api.conf_logger(__name__)

for i, row in enumerate(csv_rows):
    channel_id = channels[row['channel']]
    segment_id = segments[row['segment']]
    zip_code_id = zip_codes[row['zip_code']]
    spend = dl.data.centify(row['spend'])

    insert = "INSERT INTO fact_sales (id, segment_id,\n        zip_code_id, channel_id, spend) VALUES({id}, \\\n            {sid}, {zid}, {cid}, {spend})"
    c.execute(insert.format(id=i, sid=segment_id,
                           zid=zip_code_id, cid=channel_id,
                           spend=spend))

    if i % 1000 == 0:
        logger.info("Progress %s/64000", i)
        conn.commit()

    conn.commit()
    c.close()
    conn.close()

```

11. Define the following function to download and parse the data:

```

@memory.cache
def get_and_parse():
    out = dl.data.get_direct_marketing_csv()
    return dl.data.read_csv(out)

```

12. The following block uses the functions and classes we defined:

```

if __name__ == "__main__":
    dbname = os.path.join(dl.data.get_data_dir(), 'marketing.db')
    session = create_session(dbname)
    populate_dim_segment(session)
    populate_dim_zip_code(session)
    populate_dim_channels(session)

    if session.query(FactSales).count() < 64000:
        load(get_and_parse(), session, dbname)

    fsum = func.sum(FactSales.spend)
    query = session.query(DimSegment.segment, DimChannel.channel,
                          DimZipCode.zip_code, fsum)

```

```

dim_cols = (DimSegment.segment, DimChannel.channel,
DimZipCode.zip_code)
dim_entities = [dl.db.entity_from_column(col) for col in dim_
cols]
spend_totals = query.join(FactSales,
                           *dim_entities)\n
                           .group_by(*dim_cols).order_by(fsum.
desc()).all()
print(tabulate(spend_totals, tablefmt='psql',
               headers=['Segment', 'Channel', 'Zip Code',
'Spend']))

```

Refer to the following screenshot for the end result (spending amounts in cents):

Segment	Channel	Zip Code	Spend
Mens E-Mail	Web	Surburban	753965
Mens E-Mail	Phone	Urban	475161
Womens E-Mail	Web	Urban	465327
Mens E-Mail	Phone	Surburban	447702
Mens E-Mail	Web	Urban	437970
Womens E-Mail	Phone	Urban	397354
Womens E-Mail	Web	Surburban	333875
No E-Mail	Phone	Surburban	292136
No E-Mail	Web	Surburban	289213
Womens E-Mail	Phone	Surburban	273985
Mens E-Mail	Web	Rural	252286
No E-Mail	Web	Urban	244141
Mens E-Mail	Multichannel	Urban	224474
Womens E-Mail	Web	Rural	218235
Womens E-Mail	Multichannel	Urban	216351
No E-Mail	Phone	Urban	211589
Womens E-Mail	Multichannel	Surburban	209122
Mens E-Mail	Multichannel	Surburban	198106
Mens E-Mail	Phone	Rural	193816
Womens E-Mail	Phone	Rural	157707
No E-Mail	Phone	Rural	97356
No E-Mail	Web	Rural	95936
No E-Mail	Multichannel	Surburban	62989
No E-Mail	Multichannel	Urban	49961
Mens E-Mail	Multichannel	Rural	47689
No E-Mail	Multichannel	Rural	47512
Womens E-Mail	Multichannel	Rural	31855

See also

- The Star schema Wikipedia page at https://en.wikipedia.org/wiki/Star_schema (retrieved September 2015)

Using HDFS

Hadoop Distributed File System (HDFS) is the storage component of the Hadoop framework for Big Data. HDFS is a distributed filesystem, which spreads data on multiple systems, and is inspired by the Google File System used by Google for its search engine. HDFS requires a **Java Runtime Environment (JRE)**, and it uses a NameNode server to keep track of the files. The system also replicates the data so that losing a few nodes doesn't lead to data loss. The typical use case for HDFS is processing large read-only files. Apache Spark, also covered in this chapter, can use HDFS too.

Getting ready

Install Hadoop and a JRE. As these are not Python frameworks, you will have to check what the appropriate procedure is for your operating system. I used Hadoop 2.7.1 with Java 1.7.0_60 for this recipe. This can be a complicated process, but there are many resources online that can help you troubleshoot for your specific system.

How to do it...

We can configure HDFS with several XML files found in your Hadoop install. Some of the steps in this section serve only as example and you should implement them as appropriate for your operating system, environment, and personal preferences:

1. Edit the `core-site.xml` file so that it has the following content (comments omitted):

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <property>
        <name>fs.default.name</name>
        <value>hdfs://localhost:8020</value>
    </property>
</configuration>
```

2. Edit the `hdfs-site.xml` file so that it has the following content (comments omitted), setting the replication of each file to just 1, to run HDFS locally:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

3. If necessary, enable Remote login on your system to SSH into localhost and generate keys (Windows users can use putty):

```
$ ssh-keygen -t dsa -f ~/.ssh/id_dsa
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```
4. Format the filesystem from the root of the Hadoop directory:

```
$ bin/hdfs namenode -format
```
5. Start the NameNode server, as follows (the opposite command is \$ sbin/stop-dfs.sh):

```
$ sbin/start-dfs.sh
```
6. Create a directory in HDFS with the following command:

```
$ hadoop fs -mkdir direct_marketing
```
7. Optionally, if you want to use the direct_marketing.csv file in the Spark recipe, you need to copy it into HDFS, as follows:

```
$ hadoop fs -copyFromLocal <path to file>/direct_marketing.csv
direct_marketing
```

See also

- ▶ The HDFS user guide at <https://hadoop.apache.org/docs/r2.6.0/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html> (retrieved September 2015)

Setting up Spark

Apache Spark is a project in the Hadoop ecosystem (refer to the *Using HDFS* recipe), which purportedly performs better than Hadoop's MapReduce. Spark loads data into memory as much as possible, and it has good support for machine learning. In the *Clustering data with Spark* recipe, we will apply a machine learning algorithm via Spark.

Spark can work standalone, but it is designed to work with Hadoop using HDFS. **Resilient Distributed Datasets (RDDs)** are the central structure in Spark, and they represent distributed data. Spark has good support for Scala, which is a JVM language, and a somewhat lagging support for Python. For instance, the support to stream in the pyspark API lags a bit. Spark also has the concept of DataFrames, but it is not implemented through pandas, but through a Spark implementation.

Getting ready

Download Spark from the downloads page at <https://spark.apache.org/downloads.html> (retrieved September 2015). I downloaded the spark-1.5.0-bin-hadoop2.6.tgz archive for Spark 1.5.0.

Unpack the archive in an appropriate directory.

How to do it...

The following steps illustrate a basic setup for Spark with a few optional steps:

1. If you want to use a different Python version than the system Python, set the PYSPARK PYTHON environment variable via the GUI of your operating system or the CLI, as follows:

```
$ export PYSPARK_PYTHON=/path/to/anaconda/bin/python
```
2. Set the SPARK_HOME environment variable, as follows:

```
$ export SPARK_HOME=<path/to/spark/>spark-1.5.0-bin-hadoop2.6
```
3. Add the python directory to your PYTHONPATH environment variable, as follows:

```
$ export PYTHONPATH=$SPARK_HOME/python:$PYTHONPATH
```
4. Add the ZIP of py4j to your PYTHONPATH environment variable, as follows:

```
$ export PYTHONPATH=$SPARK_HOME/python/lib/py4j-0.8.2.1-src.zip:$PYTHONPATH
```
5. If the logging of Spark is too verbose, copy the log4j.properties.template file in the \$SPARK_HOME/conf directory to log4j.properties and change the INFO levels to WARN.

See also

The official Spark website is at <http://spark.apache.org/> (retrieved September 2015)

Clustering data with Spark

In the previous recipe, *Setting up Spark*, we covered a basic setup of Spark. If you followed the *Using HDFS* recipe, you can optionally serve the data from Hadoop. In this case, you need to specify the URL of the file in this manner, `hdfs://hdfs-host:port/path/direct_marketing.csv`.

We will use the same data as we did in the *Implementing a star schema with fact and dimension tables* recipe. However, this time we will use the spend, history, and recency columns. The first column corresponds to recent purchase amounts after a direct marketing campaign, the second to historical purchase amounts, and the third column to the recency of purchase in months. The data is described in <http://blog.minethatdata.com/2008/03/minethatdata-e-mail-analytics-and-data.html> (retrieved September 2015). We will apply the popular K-means machine-learning algorithm to cluster the data. *Chapter 9, Ensemble Learning and Dimensionality Reduction*, pays more attention to machine learning algorithms. The K-means algorithm attempts to find the best clusters for a dataset given a number of clusters. We are supposed to either know this number or find it through trial and error. In this recipe, I evaluate the clusters through the **Within Set Sum Squared Error (WSSSE)**, also known as **Within Cluster Sum of Squares (WCSS)**. This metric calculates the sum of the squared error of the distance between each point and its assigned cluster. You can read more about evaluation metrics in *Chapter 10, Evaluating Classifiers, Regressors, and Clusters*.

Getting ready

Follow the instructions in the *Setting up Spark* recipe.

How to do it...

The code for this recipe is in the `clustering_spark.py` file in this book's code bundle:

1. The imports are as follows:

```
from pyspark.mllib.clustering import KMeans
from pyspark import SparkContext
import dautil as dl
import csv
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib.colors import Normalize
```

2. Define the following function to compute the error:

```
def error(point, clusters):
    center = clusters.centers[clusters.predict(point)]

    return dl.stats.wssse(point, center)
```

3. Read and parse the data, as follows:

```
sc = SparkContext()
csv_file = dl.data.get_direct_marketing_csv()
lines = sc.textFile(csv_file)
header = lines.first().split(',')

---


```

```
cols_set = set(['recency', 'history', 'spend'])
select_cols = [i for i, col in enumerate(header) if col in cols_set]
```

4. Set up the following RDDs:

```
header_rdd = lines.filter(lambda l: 'recency' in l)
noheader_rdd = lines.subtract(header_rdd)
temp = noheader_rdd.map(lambda v: list(csv.reader([v]))[0]) \
    .map(lambda p: (int(p[select_cols[0]]),
                    dl.data.centify(p[select_cols[1]]),
                    dl.data.centify(p[select_cols[2]])))
```

spend > 0

```
temp = temp.filter(lambda x: x[2] > 0)
```

5. Cluster the data with the k-means algorithm:

```
points = []
clusters = None

for i in range(2, 28):
    clusters = KMeans.train(temp, i, maxIterations=10,
                            runs=10, initializationMode="random")

    val = temp.map(lambda point: error(point, clusters)) \
        .reduce(lambda x, y: x + y)
    points.append((i, val))
```

6. Plot the clusters, as follows:

```
dl.options.mimic_seaborn()
fig, [ax, ax2] = plt.subplots(2, 1)
ax.set_title('k-means Clusters')
ax.set_xlabel('Number of clusters')
ax.set_ylabel('WSSSE')
dl.plotting.plot_points(ax, points)

collected = temp.collect()
recency, history, spend = zip(*collected)
indices = [clusters.predict(c) for c in collected]
ax2.set_title('Clusters for spend, history and recency')
ax2.set_xlabel('history (cents)')
ax2.set_ylabel('spend (cents)')
markers = dl.plotting.map_markers(indices)
```

```

colors = dl.plotting.sample_hex_cmap(name='hot',
ncolors=len(set(recency)))

for h, s, r, m in zip(history, spend, recency, markers):
    ax2.scatter(h, s, s=20 + r, marker=m, c=colors[r-1])

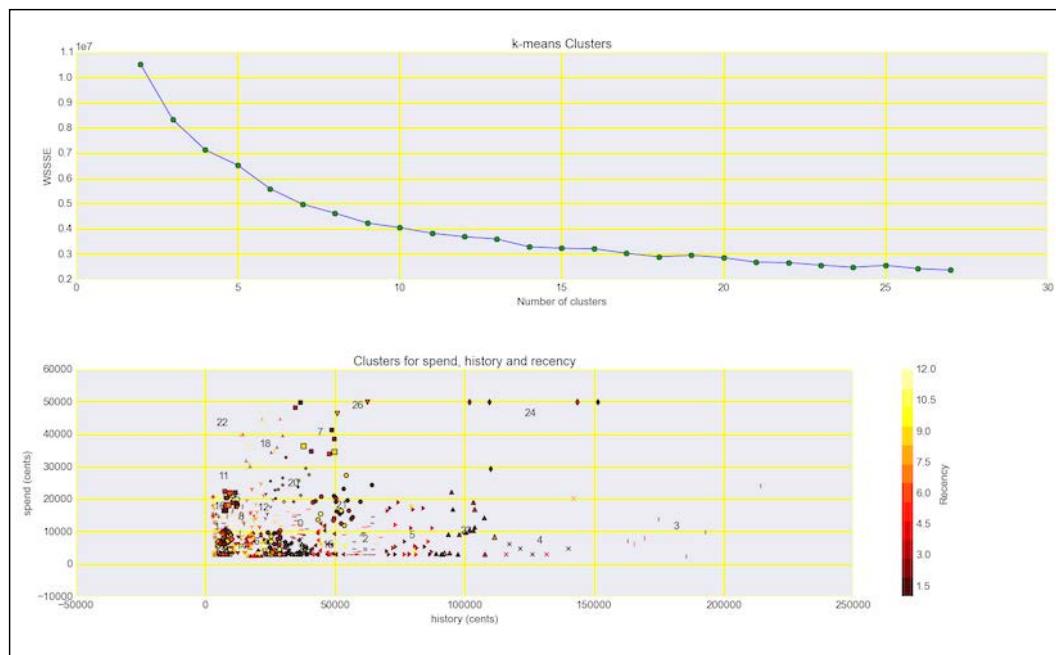
cma = mpl.colors.ListedColormap(colors, name='from_list', N=None)
norm = Normalize(min(recency), max(recency))
msm = mpl.cm.ScalarMappable(cmap=cma, norm=norm)
msm.set_array([])
fig.colorbar(msm, label='Recency')

for i, center in enumerate(clusters.clusterCenters):
    recency, history, spend = center
    ax2.text(history, spend, str(i))

plt.tight_layout()
plt.show()

```

Refer to the following screenshot for the end result (the numbers in the plot correspond to cluster centers):



How it works...

K-means clustering assigns data points to k clusters. The problem of clustering is not solvable directly, but we can apply heuristics, which achieve an acceptable result. The algorithm for k-means iterates between two steps not including the (usually random) initialization of k-means:

- ▶ Assign each data point a cluster with the lowest WCSS mean
- ▶ Recalculate the center of the cluster as the mean of the cluster points coordinates

The algorithm stops when the cluster assignments become stable.

There's more...

Spark 1.5.0 added experimental support to stream K-means. Due to the experimental nature of these new features, I decided to not discuss them in detail. I have added the following example code in the `streaming_clustering.py` file in this book's code bundle:

```
import dautil as dl
from pyspark.mllib.clustering import StreamingKMeansModel
from pyspark import SparkContext

csv_file = dl.data.get_direct_marketing_csv()
csv_rows = dl.data.read_csv(csv_file)

stkm = StreamingKMeansModel(28 * [[0., 0., 0.]], 28 * [1.])
sc = SparkContext()

for row in csv_rows:
    spend = dl.data.centify(row['spend'])

    if spend > 0:
        history = dl.data.centify(row['history'])
        data = sc.parallelize([[int(row['recency']),
                               history, spend]])
        stkm = stkm.update(data, 0., 'points')

print(stkm.centers)
```

See also

- ▶ *Building Machine Learning Systems with Python*, Willi Richert, and Luis Pedro Coelho (2013)
- ▶ The Wikipedia page about K-means clustering is at https://en.wikipedia.org/wiki/K-means_clustering (retrieved September 2015)

6

Signal Processing and Timeseries

In this chapter, we will cover the following recipes:

- ▶ Spectral analysis with periodograms
- ▶ Estimating power spectral density with the Welch method
- ▶ Analyzing peaks
- ▶ Measuring phase synchronization
- ▶ Exponential smoothing
- ▶ Evaluating smoothing
- ▶ Using the Lomb-Scargle periodogram
- ▶ Analyzing the frequency spectrum of audio
- ▶ Analyzing signals with the discrete cosine transform
- ▶ Block bootstrapping time series data
- ▶ Moving block bootstrapping time series data
- ▶ Applying the discrete wavelet transform

Introduction

Time is an important dimension in science and daily life. Time series data is abundant and requires special techniques. Usually, we are interested in trends and seasonality or periodicity. In mathematical terms, this means that we try to represent the data by (usually linear) polynomial or trigonometric functions, or a combination of both.

When we investigate seasonality, we generally distinguish between time domain and frequency domain analysis. In the time domain, we can use a dozen pandas functions for rolling windows. We can also smooth data to remove noise while hopefully keeping enough of the signal. Smoothing is in many respects similar to fitting, which is convenient because we can reuse some of the regression tools we know.

To get in the frequency domain, we apply transforms such as the **fast Fourier Transform** and **discrete cosine transform**. We can then further analyze signals with periodograms.

Spectral analysis with periodograms

We can think of periodic signals as being composed of multiple frequencies. For instance, sound is composed of multiple tones and light is composed of multiple colors. The range of frequencies is called the **frequency spectrum**. When we analyze the frequency spectrum of a signal, it's natural to take a look at the result of the Fourier Transform of the signal. The periodogram extends this and is equal to the squared magnitude of the Fourier Transform, as follows:

$$(6.1) \quad S(f) = \frac{\Delta t}{N} \left| \sum_{n=0}^{N-1} x_n e^{-i2\pi n f} \right|^2, \quad -\frac{1}{2\Delta t} < f \leq \frac{1}{2\Delta t}$$

We will look at the periodograms of the following variables:

- ▶ Rain values from the KNMI De Bilt weather data
- ▶ The second difference (comparable to second derivative in calculus) of the rain values
- ▶ The rolling sum of the rain values using a window of 365 days
- ▶ The rolling mean of the rain values using a window of 365 days

How to do it...

1. The imports are as follows:

```
from scipy import signal
import matplotlib.pyplot as plt
import dautil as dl
import numpy as np
import pandas as pd
from IPython.display import HTML
```

2. Load the data as follows:

```
fs = 365
rain = dl.data.Weather.load()['RAIN'].dropna()
```

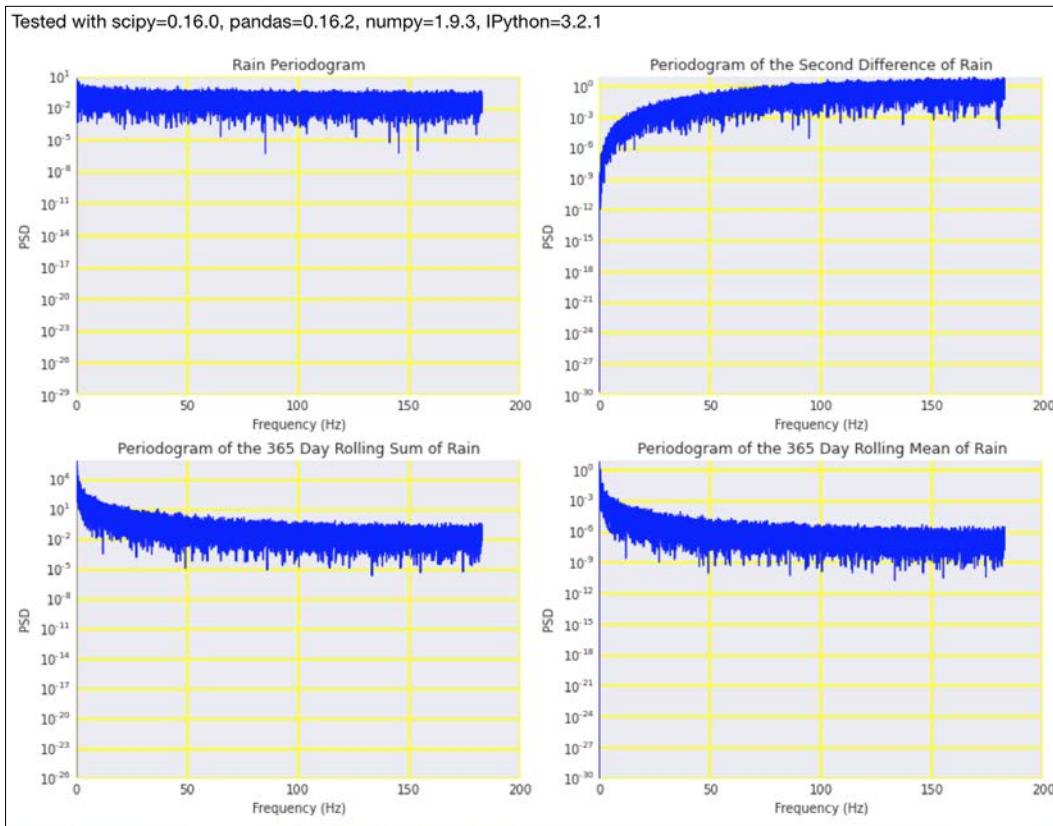
3. Define the following function to plot periodograms:

```
def plot_periodogram(arr, ax):
    f, Pxx_den = signal.periodogram(arr, fs)
    ax.set_xlabel('Frequency (Hz)')
    ax.set_ylabel('PSD')
    ax.semilogy(f, Pxx_den)
```

4. Plot the periodograms with the following code:

```
sp = dl.plotting.Subplotter(2, 2, context)
sp.label()
plot_periodogram(rain, sp.ax)
sp.label(advance=True)
plot_periodogram(np.diff(rain, 2), sp.ax)
sp.label(advance=True)
plot_periodogram(pd.rolling_sum(rain, fs).dropna(), sp.ax)
sp.label(advance=True)
plot_periodogram(pd.rolling_mean(rain, fs).dropna(), sp.ax)
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code is from the `periodograms.ipynb` file in this book's code bundle demonstrates periodograms.

See also

- ▶ The documentation for the `periodogram()` function at <https://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.signal.periodogram.html#scipy.signal.periodogram> (retrieved September 2015)
- ▶ The relevant Wikipedia page at https://en.wikipedia.org/wiki/Spectral_density_estimation (retrieved September 2015)

Estimating power spectral density with the Welch method

The **Welch method** is an improvement (it reduces noise) of the periodogram technique and is named after P.D. Welch. The noise of the power spectrum is reduced with the following steps:

1. We split the signal with a fixed number of overlapping points. If the overlap is 0, then we have **Bartlett's method**.
2. In the time domain, we apply window functions to each of the segments of step 1.
3. We compute the periodogram for each segment as explained in the *Spectral analysis with periodograms* recipe.
4. We average the periodograms, thus reducing noise. Averaging effectively smoothens the signal. However, we are now dealing with frequency bins (like in a histogram).

We will also explore the **Fano factor**, which is given as follows:

$$(6.2) \quad F = \frac{\sigma_w^2}{\mu_w}$$

It is a windowed variance-to-mean ratio. Dividing by the mean basically normalizes the values, and we get a normalized measure of dispersion. As input data we will use temperature data.

How to do it...

1. The imports are as follows:

```
from scipy import signal
import matplotlib.pyplot as plt
import dautil as dl
from IPython.display import HTML
```

2. Load the data and compute the Fano factor:

```
fs = 365
temp = dl.data.Weather.load() ['TEMP'].dropna()
fano_factor = dl.ts.fano_factor(temp, fs)
```

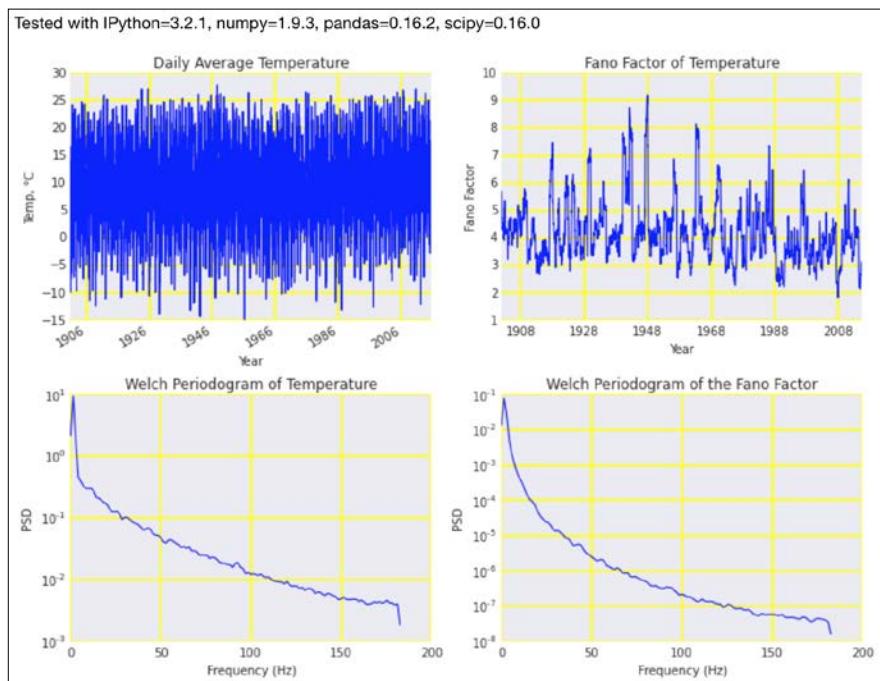
3. Define the following function to plot the periodograms:

```
def plot_welch(arr, ax):
    f, Pxx_den = signal.welch(arr, fs)
    ax.semilogy(f, Pxx_den)
```

4. Plot the input data and corresponding periodograms:

```
sp = dl.plotting.Subplotter(2, 2, context)
temp.plot(ax=sp.ax)
sp.label(ylabel_params=dl.data.Weather.get_header('TEMP'))
sp.label(advance=True)
sp.ax.plot(temp.index, fano_factor)
sp.label(advance=True)
plot_welch(temp, sp.ax)
sp.label(advance=True)
plot_welch(fano_factor.dropna(), sp.ax)
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code is in the `estimating_welch.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about the Fano factor at https://en.wikipedia.org/wiki/Fano_factor (retrieved September 2015)
- ▶ The Wikipedia page about the Welch method at https://en.wikipedia.org/wiki/Welch's_method (retrieved September 2015)
- ▶ The `welch()` function documented at <https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.welch.html#scipy.signal.welch> (retrieved September 2015)

Analyzing peaks

The analysis of peaks is similar to that of valleys, since both are extreme values. SciPy has the `argrelmax()` function that finds the relative maxima. When we apply this function to daily temperature values, it not only finds hot days in summer but also hot days in winter unless we make the function consider a larger time frame. Of course, we can also check whether values are above a threshold or only select summer data using prior knowledge.

When we analyze peaks in time series data, we can apply two approaches. The first approach is to consider the highest peaks in a year, a month, or another fixed time interval and build a series with those values. The second approach is to define any value above a threshold as a peak. In this recipe, we will use the 95th percentile as the threshold. In the context of this approach, we can have multiple peaks in a sequence. Long streaks can have a negative impact, for instance, in the case of heat waves.

How to do it...

1. The imports are as follows:

```
import dautil as dl
from scipy import signal
import matplotlib.pyplot as plt
import seaborn as sns
from IPython.display import HTML
```

2. Load and resample the data:

```
temp = dl.data.Weather.load() ['TEMP'].dropna()
monthly = temp.resample('M')
```

3. Plot peaks and note that hot days in winter are also considered:

```
sp = dl.plotting.Subplotter(2, 2, context)
max_locs = signal.argrelextrema(monthly.values)
sp.ax.plot(monthly.index, monthly, label='Monthly means')
sp.ax.plot(monthly.index[max_locs], monthly.values[max_locs],
           'o', label='Tops')
sp.label(ylabel_params=dl.data.Weather.get_header('TEMP'))
```

4. Plot the annual maximum series:

```
annual_max = dl.ts.groupby_year(temp).max()
sp.next_ax().plot(annual_max.index, annual_max, label='Annual
Maximum Series')
dl.plotting.plot_polyfit(sp.ax, annual_max.index, annual_max.
values)
sp.label(ylabel_params=dl.data.Weather.get_header('TEMP'))
```

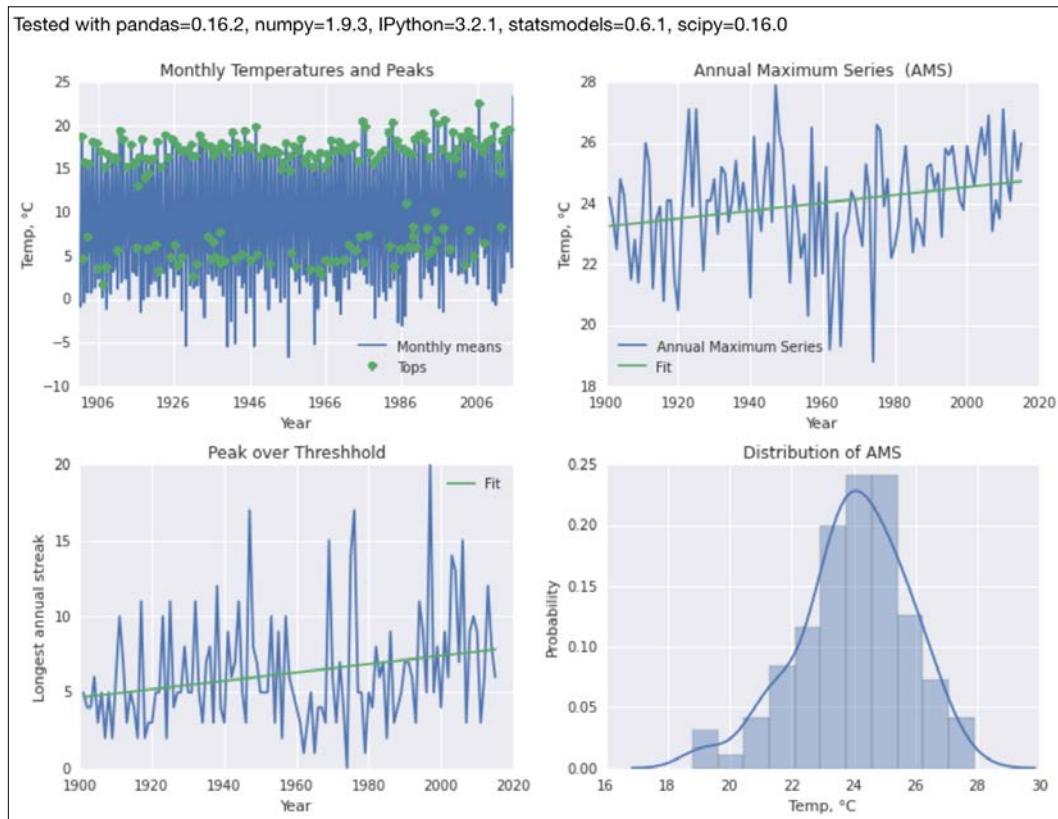
5. Plot the longest annual streaks of hot days over the 95th percentile threshold:

```
_, threshhold = dl.stats.outliers(temp, method='percentiles')
over_threshhold = temp > threshhold
streaks = dl.ts.groupby_year(over_threshhold).apply(
    lambda x: dl.collect.longest_streak(x, 1))
sp.next_ax().plot(streaks.index, streaks)
dl.plotting.plot_polyfit(sp.ax, streaks.index, streaks.values)
over_threshhold = dl.ts.groupby_year(over_threshhold).mean()
sp.label()
```

6. Plot the annual maximum series distribution:

```
sp.label(advance=True)
sns.distplot(annual_max, ax=sp.ax)
sp.label(xlabel_params=dl.data.Weather.get_header('TEMP'))
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code is in the `analyzing_peaks.ipynb` file in this book's code bundle.

See also

- ▶ The `argrelmax()` function documented at <https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.argrelmax.html> (retrieved September 2015)

Measuring phase synchronization

Two signals can be fully synchronized, not synchronized, or somewhere in between. We usually measure **phase synchronization** in radians. The related quantity of **instantaneous phase** can be measured with the NumPy `angle()` function. For real-valued data, we need to obtain the analytic representation of the signal, which is given by the Hilbert transform. The Hilbert transform is also available in SciPy and NumPy.

Cross-correlation measures the correlation between two signals using a sliding inner product. We can use cross-correlation to measure the time delay between two signals. NumPy offers the `correlate()` function, which calculates the cross-correlation between two arrays.

How to do it...

1. The imports are as follows:

```
import dautil as dl
import matplotlib.pyplot as plt
import numpy as np
from IPython.display import HTML
```

2. Load the data and calculate the instantaneous phase:

```
df = dl.data.Weather.load().dropna()
df = dl.ts.groupby_yday(df).mean().dropna()
ws_phase = dl.ts.instant_phase(df['WIND_SPEED'])
wd_phase = dl.ts.instant_phase(df['WIND_DIR'])
```

3. Plot the wind direction and speed z-scores:

```
sp = dl.plotting.Subplotter(2, 2, context)
cp = dl.plotting.CyclePlotter(sp.ax)
cp.plot(df.index, dl.stats.zscores(df['WIND_DIR'].values),
        label='Wind direction')
cp.plot(df.index, dl.stats.zscores(df['WIND_SPEED'].values),
        label='Wind speed')
sp.label()
```

4. Plot the instantaneous phase as follows:

```
cp = dl.plotting.CyclePlotter(sp.next_ax())
cp.plot(df.index, ws_phase, label='Wind speed')
cp.plot(df.index, wd_phase, label='Wind direction')
sp.label()
```

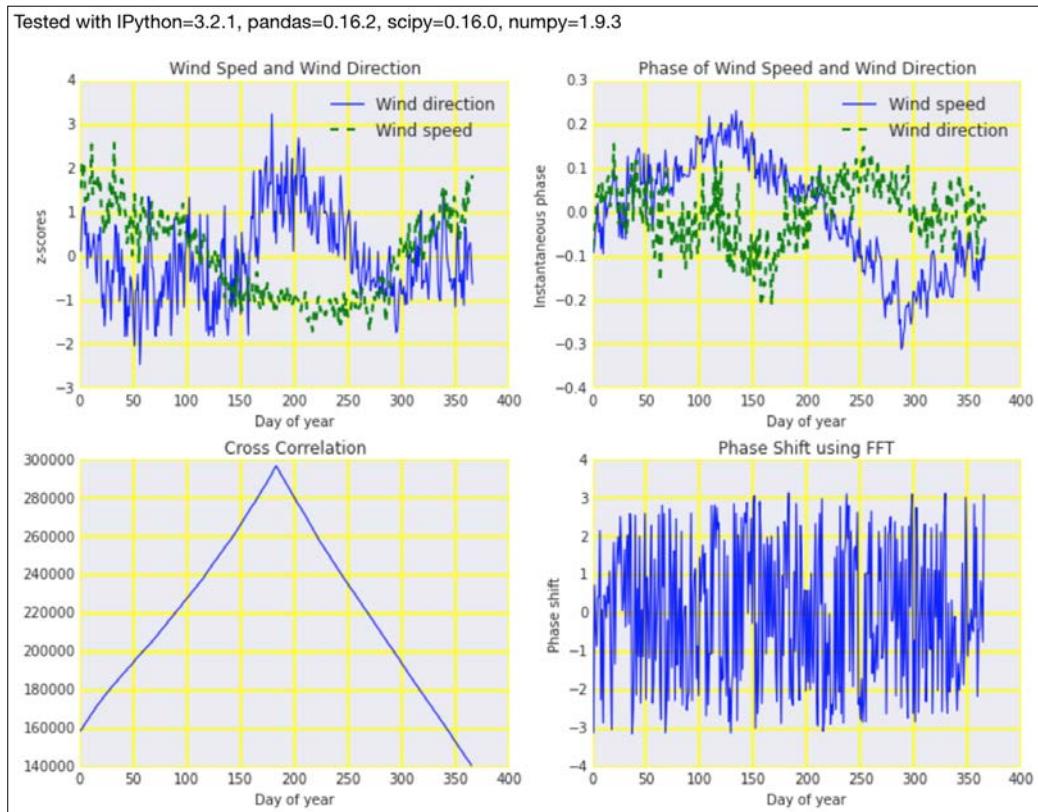
5. Plot the correlation of wind speed and direction:

```
sp.label(advance=True)
sp.ax.plot(np.correlate(df['WIND_SPEED'], df['WIND_DIR'], 'same'))
```

6. Plot the phase shift with the fast Fourier Transform:

```
sp.label(advance=True)
sp.ax.plot(np.angle(np.fft.fft(df['WIND_SPEED'])/np.fft.
fft(df['WIND_DIR'])))
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code for this recipe is in the `phase_synchrony.ipynb` file in this book's code bundle.

See also

- ▶ The instantaneous phase Wikipedia page at https://en.wikipedia.org/wiki/Instantaneous_phase (retrieved September 2015)
- ▶ The analytic signal Wikipedia page at https://en.wikipedia.org/wiki/Analytic_signal (retrieved September 2015)
- ▶ The Wikipedia page about cross-correlation at <https://en.wikipedia.org/wiki/Cross-correlation> (retrieved September 2015)
- ▶ The documentation for the `angle()` function at <https://docs.scipy.org/doc/numpy/reference/generated/numpy.angle.html> (retrieved September 2015)
- ▶ The documentation for the `correlate()` function at <https://docs.scipy.org/doc/numpy/reference/generated/numpy.correlate.html> (retrieved September 2015)

Exponential smoothing

Exponential smoothing is a low-pass filter that aims to remove noise. In this recipe, we will apply single and double exponential smoothing, as shown by the following equations:

$$(6.3) \quad s_t = \alpha x_t + (1 - \alpha)s_{t-1}, \quad t > 0$$

$$(6.4) \quad s_t = \alpha x_t + (1 - \alpha)(s_{t-1} + b_{t-1})$$

$$(6.5) \quad b_t = \beta(s_t - s_{t-1}) + (1 - \beta)b_{t-1}$$

Single exponential smoothing (6.3) requires the **smoothing factor** α , where $0 < \alpha < 1$. Double exponential smoothing (6.4 and 6.5) attempts to handle trends in data via the **trend smoothing factor** β , where $0 < \beta < 1$.

We will also take a look at rolling deviations of wind speed, which are similar to z-scores, but they are applied to a rolling window. Smoothing is associated with regression, although the goal of smoothing is to get rid of noise. Nevertheless, metrics related to regression, such as the **Mean Squared Error (MSE)**, are also appropriate for smoothing.

How to do it...

1. The imports are as follows:

```
import dautil as dl
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from IPython.display import HTML
```

2. Define the following function to help visualize the result of double exponential smoothing:

```
def grid_mse(i, j, devs):
    alpha = 0.1 * i
    beta = 0.1 * j
    cell = dl.ts.double_exp_smoothing(devs.values, alpha, beta)

    return dl.stats.mse(devs, cell)
```

-
3. Load the wind speed data and calculate annual means and rolling deviations:

```
wind = dl.data.Weather.load()['WIND_SPEED'].dropna()  
wind = dl.ts.groupby_year(wind).mean()  
devs = dl.ts.rolling_deviations(wind, 12).dropna()
```

4. Plot the annual means of the wind speed data:

```
sp = dl.plotting.Subplotter(2, 2, context)  
sp.label(ylabel_params=dl.data.Weather.get_header('WIND_SPEED'))  
sp.ax.plot(wind.index, wind)
```

5. Plot the rolling deviations with an α of 0.7:

```
cp = dl.plotting.CyclePlotter(sp.next_ax())  
cp.plot(devs.index, devs, label='Rolling Deviations')  
cp.plot(devs.index, dl.ts.exp_smoothing(devs.values, 0.7),  
label='Smoothing')  
sp.label()
```

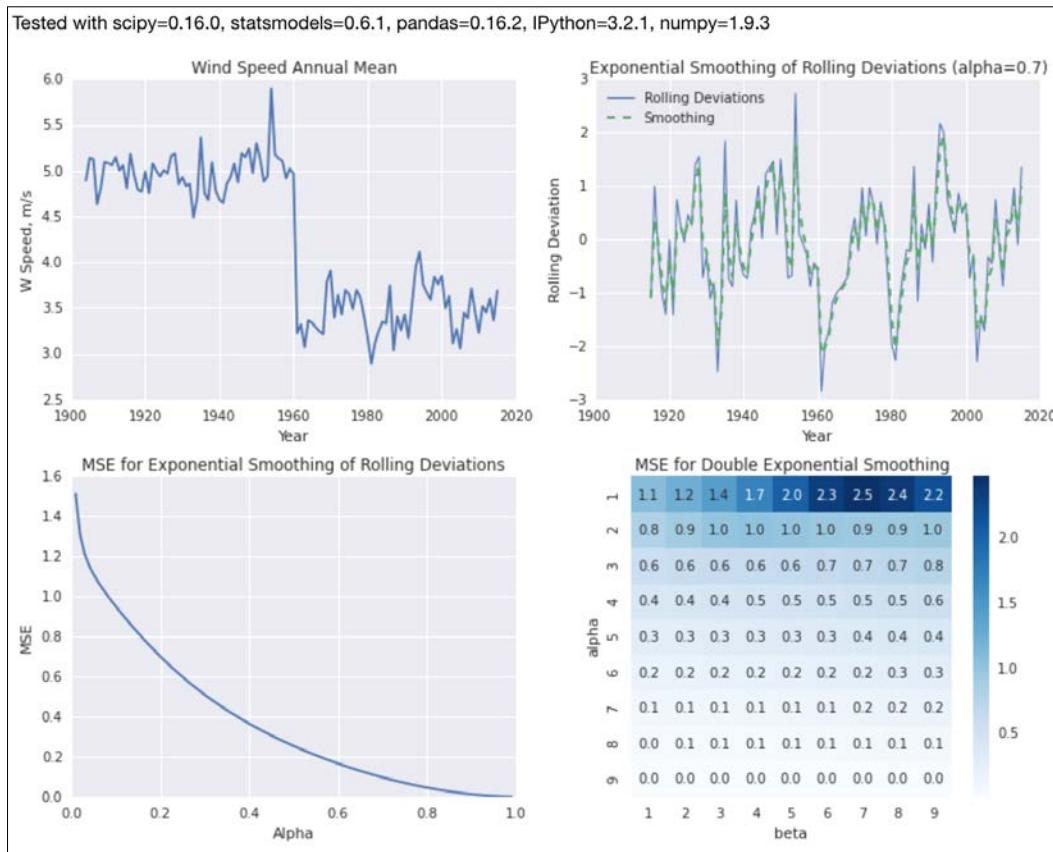
6. Plot the MSE for varying smoothing factors:

```
alphas = 0.01 * np.arange(1, 100)  
errors = [dl.stats.mse(devs, dl.ts.exp_smoothing(devs.values,  
alpha)  
          for alpha in alphas]  
sp.label(advance=True)  
sp.ax.plot(alphas, errors)
```

7. Plot the MSE for a grid of α and β values:

```
sp.label(advance=True)  
rng = range(1, 10)  
df = dl.report.map_grid(rng, rng, ["alpha", "beta", "mse"], grid_  
mse, devs)  
sns.heatmap(df, cmap='Blues', square=True, annot=True, fmt='.1f',  
            ax=sp.ax)  
  
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code is in the `exp_smoothing.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about exponential smoothing at https://en.wikipedia.org/wiki/Exponential_smoothing (retrieved September 2015)

Evaluating smoothing

Many aspects of smoothing are comparable to regression; therefore, you can apply some of the techniques in *Chapter 10, Evaluating Classifiers, Regressors, and Clusters*, to smoothing too. In this recipe, we will smooth with the **Savitzky-Golay filter**, which conforms to the following equation:

$$(6.6) \quad Y_j = \sum_{i=-(m-1)/2}^{i=(m-1)/2} C_i y_j + i \quad \frac{m+1}{2} \leq j \leq n - \frac{m-1}{2}$$

The filter fits points within a rolling window of size n to a polynomial of order m . Abraham Savitzky and Marcel J. E. Golay created the algorithm around 1964 and first applied it to chemistry problems. The filter has two parameters that naturally form a grid. As in regression problems, we will take a look at a difference, in this case, the difference between the original signal and the smoothed signal. We assume, just like when we fit data, that the residuals are random and follow a Gaussian distribution.

How to do it...

The following steps are from the `eval_smooth.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import dautil as dl
import matplotlib.pyplot as plt
from scipy.signal import savgol_filter
import pandas as pd
import numpy as np
import seaborn as sns
from IPython.display import HTML
```

2. Define the following helper functions:

```
def error(data, fit):
    return data - fit

def win_rng():
    return range(3, 25, 2)

def calc_mape(i, j, pres):
    return dl.stats.mape(pres, savgol_filter(pres, i, j))
```

3. Load the atmospheric pressure data as follows:

```
pres = dl.data.Weather.load()['PRESSURE'].dropna()
pres = pres.resample('A')
```

4. Plot the original data and the filter with window size 11 and various polynomial orders:

```
sp = dl.plotting.Subplotter(2, 2, context)
cp = dl.plotting.CyclePlotter(sp.ax)
cp.plot(pres.index, pres, label='Pressure')
cp.plot(pres.index, savgol_filter(pres, 11, 2), label='Poly order 2')
cp.plot(pres.index, savgol_filter(pres, 11, 3), label='Poly order 3')
cp.plot(pres.index, savgol_filter(pres, 11, 4), label='Poly order 4')
sp.label(ylabel_params=dl.data.Weather.get_header('PRESSURE'))
```

5. Plot the standard deviations of the filter residuals for varying window sizes:

```
cp = dl.plotting.CyclePlotter(sp.next_ax())
stds = [error(pres, savgol_filter(pres, i, 2)).std()
        for i in win_rng()]
cp.plot(win_rng(), stds, label='Filtered')
stds = [error(pres, pd.rolling_mean(pres, i)).std()
        for i in win_rng()]
cp.plot(win_rng(), stds, label='Rolling mean')
sp.label()
```

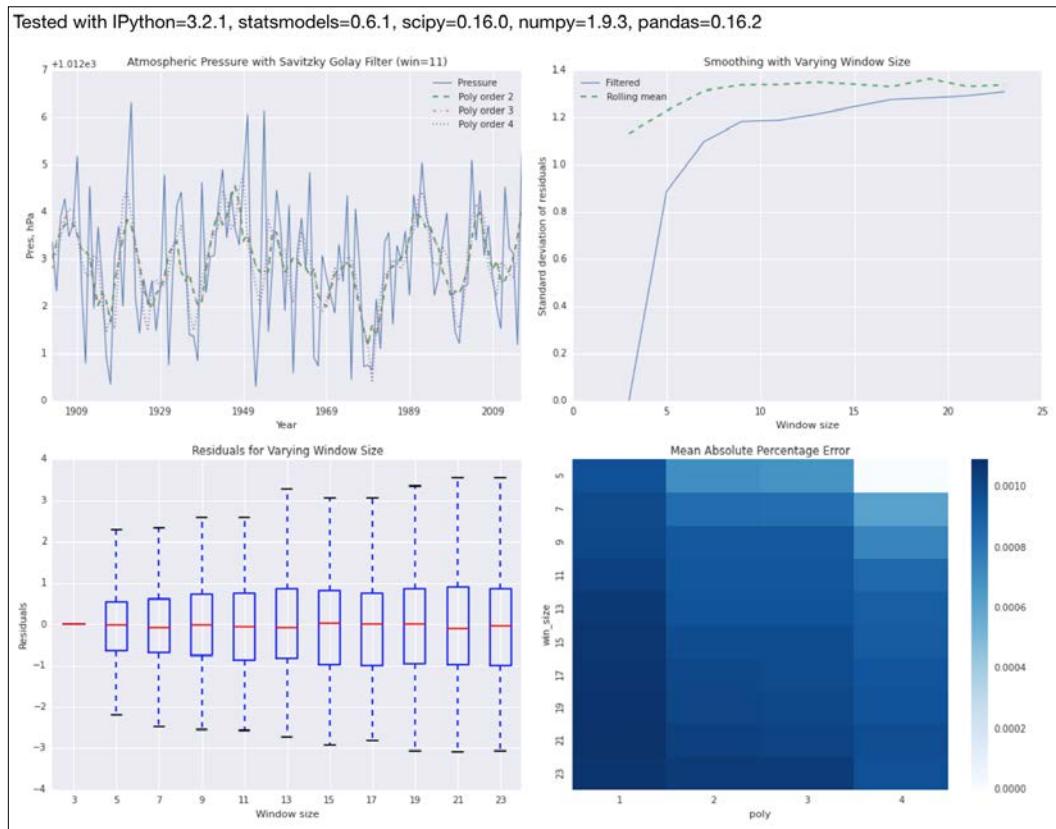
6. Plot the box plots of the filter residuals:

```
sp.label(advance=True)
sp.ax.boxplot([error(pres, savgol_filter(pres, i, 2))
               for i in win_rng()])
sp.ax.set_xticklabels(win_rng())
```

7. Plot the MAPE for a grid of window sizes and polynomial orders:

```
sp.label(advance=True)
df = dl.report.map_grid(win_rng()[1:], range(1, 5),
                        ['win_size', 'poly', 'mape'], calc_mape, pres)
sns.heatmap(df, cmap='Blues', ax=sp.ax)
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



See also

- ▶ The Wikipedia page about the Savitzky-Golay filter at https://en.wikipedia.org/wiki/Savitzky%E2%80%93Golay_filter (retrieved September 2015)
- ▶ The `savgol_filter()` function documented at https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.savgol_filter.html (retrieved September 2015)

Using the Lomb-Scargle periodogram

The Lomb-Scargle periodogram is a frequency spectrum estimation method that fits sines to data, and it is frequently used with unevenly sampled data. The method is named after Nicholas R. Lomb and Jeffrey D. Scargle. The algorithm was published around 1976 and has been improved since then. Scargle introduced a time delay parameter, which separates the sine and cosine waveforms. The following equations define the time delay (6.7) and periodogram (6.8).

$$(6.7) \quad \tan 2\omega\tau = \frac{\sum_j \sin 2\omega t_j}{\sum_j \cos 2\omega t_j}$$

$$(6.8) \quad P_x(\omega) = \frac{1}{2} \left(\frac{\left[\sum_j X_j \cos \omega(t_j - \tau) \right]^2}{\sum_j \cos^2 \omega(t_j - \tau)} + \frac{\left[\sum_j X_j \sin \omega(t_j - \tau) \right]^2}{\sum_j \sin^2 \omega(t_j - \tau)} \right)$$

How to do it...

1. The imports are as follows:

```
from scipy import signal
import numpy as np
import matplotlib.pyplot as plt
import dautil as dl
import statsmodels.api as sm
from IPython.display import HTML
```

2. Load the sunspots data as follows:

```
df = sm.datasets.sunspots.load_pandas().data
sunspots = df['SUNACTIVITY'].values
size = len(sunspots)
t = np.linspace(-2 * np.pi, 2 * np.pi, size)
sine = dl.ts.sine_like(sunspots)
f = np.linspace(0.01, 2, 10 * size)
```

3. Plot a sine waveform as follows:

```
sp = dl.plotting.Subplotter(2, 2, context)
sp.ax.plot(t, sine)
sp.label()

sp.next_ax().plot(df['YEAR'], df['SUNACTIVITY'])
sp.label()
```

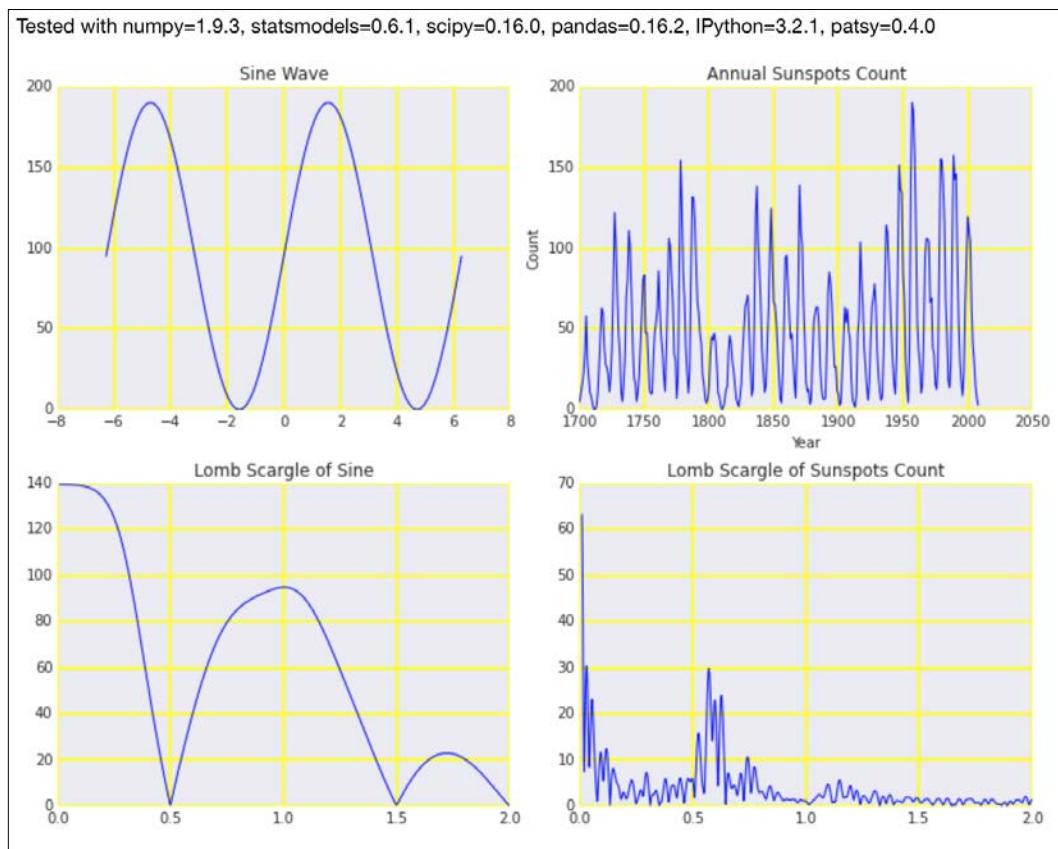
4. Apply the periodogram to the sine:

```
pgram = signal.lombscargle(t, sine, f)
sp.next_ax().plot(f, 2 * np.sqrt(pgram/size))
sp.label()
```

5. Apply the periodogram to the sunspots data:

```
pgram = signal.lombscargle(np.arange(size, dtype=float), sunspots,
f)
sp.next_ax().plot(f, 2 * np.sqrt(pgram/size))
sp.label()
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The preceding code is a breakdown of the `lomb_scargle.ipynb` file in this book's code bundle.

See also

- ▶ The relevant Wikipedia page at https://en.wikipedia.org/wiki/Least-squares_spectral_analysis (retrieved September 2015)
- ▶ The `lombscargle()` function documented at <https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.lombscargle.html> (retrieved September 2015)

Analyzing the frequency spectrum of audio

We can apply many techniques to analyze audio, and, therefore, we can debate at length about which techniques are most appropriate. The most obvious method is purportedly the FFT. As a variation, we can use the **short-time Fourier transform (STFT)**. The STFT splits the signal in the time domain into equal parts, and it then applies the FFT to each segment. Another algorithm we will use is the **cepstrum**, which was originally used to analyze earthquakes but was later successfully applied to speech analysis. The power cepstrum is given by the following equation:

$$(6.9) \quad \left| F^{-1} \left\{ \log \left(|F\{f(t)\}|^2 \right) \right\} \right|^2$$

The algorithm is as follows:

1. Calculate the Fourier transform.
2. Compute the squared magnitude of the transform.
3. Take the logarithm of the previous result.
4. Apply the inverse Fourier transform.
5. Calculate the squared magnitude again.

The cepstrum is, in general, useful when we have large changes in the frequency domain. An important use case of the cepstrum is to form feature vectors for audio classification. This requires a mapping from frequency to the **mel scale** (refer to the Wikipedia page mentioned in the See also section).

How to do it...

1. The imports are as follows:

```
import daultil as dl
import matplotlib.pyplot as plt
import numpy as np
from ch6util import read_wav
from IPython.display import HTML
```

2. Define the following function to calculate the magnitude of the signal with FFT:

```
def amplitude(arr):
    return np.abs(np.fft.fft(arr))
```

3. Load the data as follows:

```
rate, audio = read_wav()
```

4. Plot the audio waveform:

```
sp = dl.plotting.Subplotter(2, 2, context)
t = np.arange(0, len(audio)/float(rate), 1./rate)
sp.ax.plot(t, audio)
freqs = np.fft.fftfreq(audio.size, 1./rate)
indices = np.where(freqs > 0) [0]
sp.label()
```

5. Plot the amplitude spectrum:

```
magnitude = amplitude(audio)
sp.next_ax().semilogy(freqs[indices], magnitude[indices])
sp.label()
```

6. Plot the cepstrum as follows:

```
cepstrum = dl.ts.power(np.fft.ifft(np.log(magnitude ** 2)))
sp.next_ax().semilogy(cepstrum)
sp.label()
```

7. Plot the STFT as a contour diagram:

```
npieces = 200
stft_amps = []

for i, c in enumerate(dl.collect.chunk(audio[: npieces ** 2],
len(audio)/npieces)):
    amps = amplitude(c)
    stft_amps.extend(amps)
```

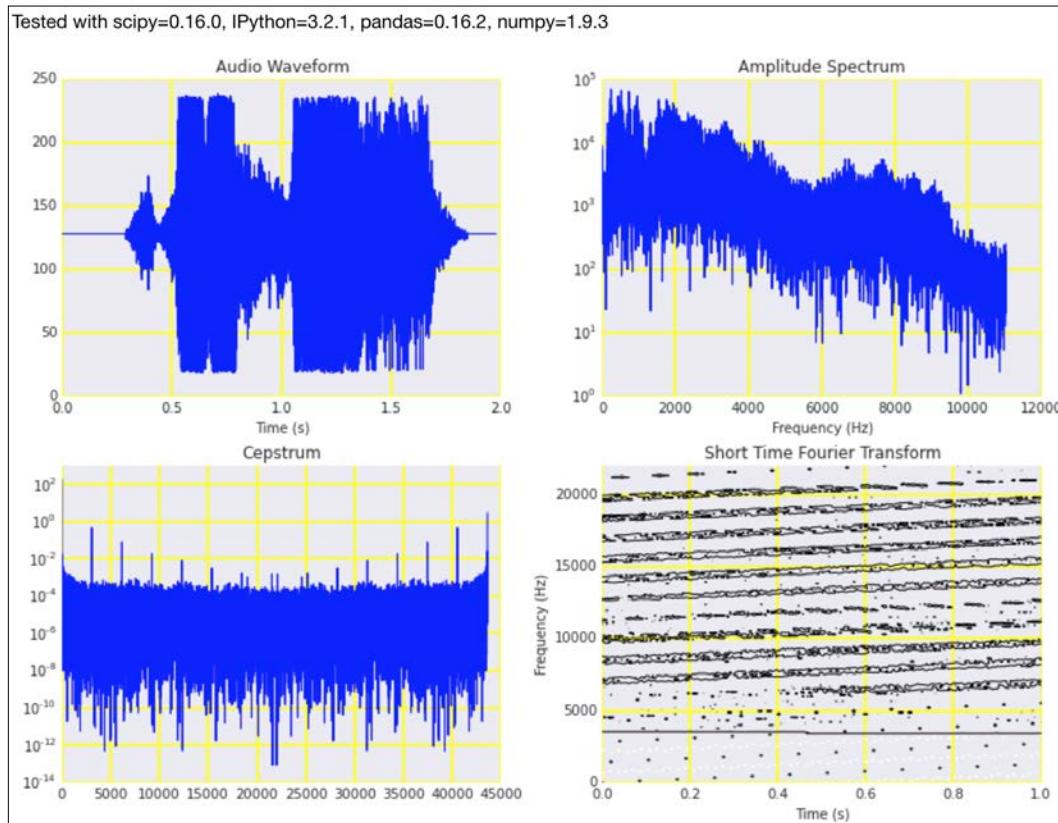
```

stft_freqs = np.linspace(0, rate, npieces)
stft_times = np.linspace(0, len(stftamps)/float(rate), npieces)
sp.next_ax().contour(stft_freqs/rate, stft_freqs,
                     np.log(stftamps).reshape(npieces, npieces))
sp.label()

HTML(sp.exit())

```

Refer to the following screenshot for the end result:



The example code is in the `analyzing_audio.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about the STFT at https://en.wikipedia.org/wiki/Short-time_Fourier_transform (retrieved September 2015)
- ▶ The Wikipedia page about the cepstrum at <https://en.wikipedia.org/wiki/Cepstrum> (retrieved September 2015)
- ▶ The Wikipedia page about the mel scale at https://en.wikipedia.org/wiki/Mel_scale (retrieved September 2015)

Analyzing signals with the discrete cosine transform

The **discrete cosine transform (DCT)** is a transform similar to the Fourier transform, but it tries to represent a signal by a sum of cosine terms only (refer to equation 6.11). The DCT is used for signal compression and in the calculation of the **mel frequency** spectrum, which I mentioned in the *Analyzing the frequency spectrum of audio* recipe. We can convert normal frequencies to the mel frequency (a frequency more appropriate for the analysis of speech and music) with the following equation:

$$(6.10) \quad m = 1127 \log\left(1 + \frac{f}{700}\right)$$

$$(6.11) \quad X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)k\right] \quad k = 0, \dots, N-1$$

The steps to create the mel frequency spectrum are not complicated, but there are quite a few of them. The relevant Wikipedia page is available at https://en.wikipedia.org/wiki/Mel-frequency_cepstrum (retrieved September 2015). If you do a quick web search, you can find a couple of Python libraries that implement the algorithm. I implemented a very simple version of the computation in this recipe.

How to do it...

1. The imports are as follows:

```
import dautil as dl
from scipy.fftpack import dct
import matplotlib.pyplot as plt
import ch6util
import seaborn as sns
import numpy as np
from IPython.display import HTML
```

2. Load the data and transform it as follows:

```
rate, audio = ch6util.read_wav()
transformed = dct(audio)
```

3. Plot the amplitude spectrum using DCT:

```
sp = dl.plotting.Subplotter(2, 2, context)
freqs = np.fft.fftfreq(audio.size, 1./rate)
indices = np.where(freqs > 0) [0]
sp.ax.semilogy(np.abs(transformed) [indices])
sp.label()
```

4. Plot the distribution of the amplitude:

```
sns.distplot(np.log(np.abs(transformed)), ax=sp.next_ax())
sp.label()
```

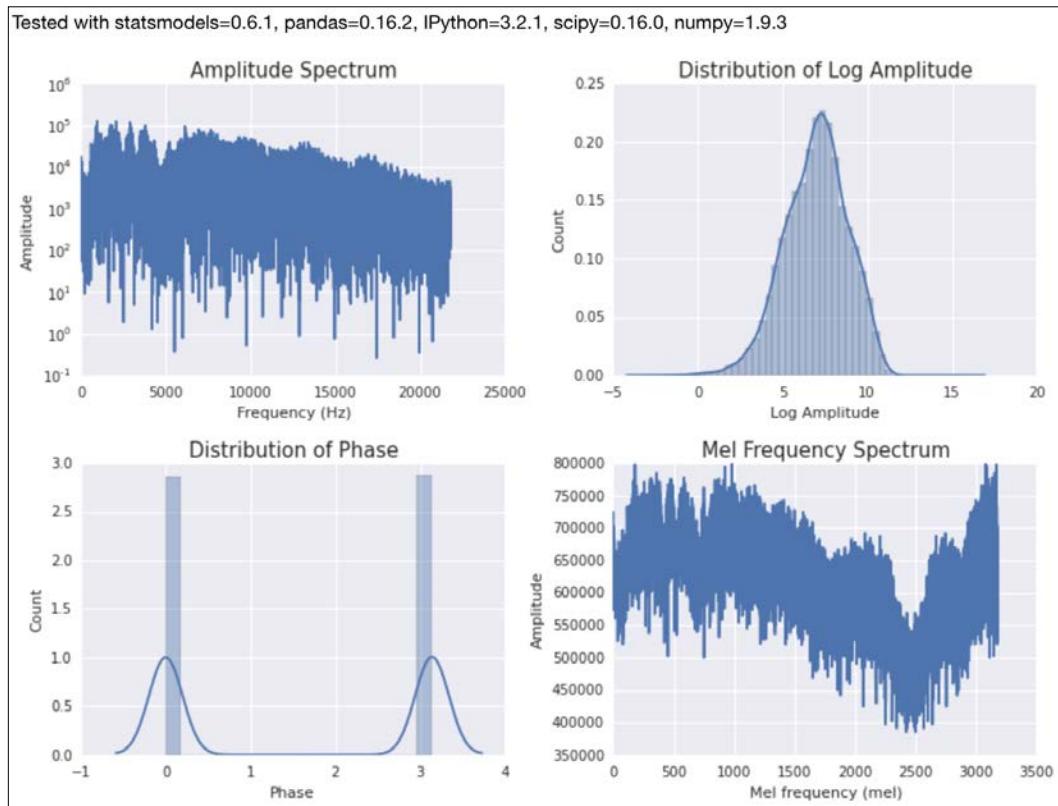
5. Plot the distribution of the phase:

```
sns.distplot(np.angle(transformed), ax=sp.next_ax())
sp.label()
```

6. Plot the mel amplitude spectrum as follows:

```
magnitude = ch6util.amplitude(audio)
cepstrum = dl.ts.power(np.fft.ifft(np.log(magnitude ** 2)))
mel = 1127 * np.log(1 + freqs[indices]/700)
sp.next_ax().plot(mel, ch6util.amplitude(dct(np.
log(magnitude[indices] ** 2))))
sp.label()
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code for this recipe is in the `analyzing_dct.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about the mel scale at https://en.wikipedia.org/wiki/Mel_scale (retrieved September 2015)
- ▶ The Wikipedia page about the DCT at https://en.wikipedia.org/wiki/Discrete_cosine_transform (retrieved September 2015)

Block bootstrapping time series data

The usual bootstrapping method doesn't preserve the ordering of time series data, and it is, therefore, unsuitable for trend estimation. In the **block bootstrapping** approach, we split data into non-overlapping blocks of equal size and use those blocks to generate new samples. In this recipe, we will apply a very naive and easy-to-implement linear model with annual temperature data. The procedure for this recipe is as follows:

1. Split the data into blocks and generate new data samples.
2. Fit the data to a line or calculate the first differences of the new data.
3. Repeat the previous step to build a list of slopes or medians of the first differences.

How to do it...

1. The imports are as follows:

```
import dautil as dl
import random
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns
import ch6util
from IPython.display import HTML
```

2. Define the following function to bootstrap the data:

```
def shuffle(temp, blocks):
    random.shuffle(blocks)
    df = pd.DataFrame({'TEMP': dl.collect.flatten(blocks)},
                      index=temp.index)
    df = df.resample('A')

    return df
```

3. Load the data and create blocks from it:

```
temp = dl.data.Weather.load() ['TEMP'].resample('M').dropna()
blocks = list(dl.collect.chunk(temp.values, 100))
random.seed(12033)
```

-
4. Plot a couple of random realizations as a sanity check:

```
sp = dl.plotting.Subplotter(2, 2, context)
cp = dl.plotting.CyclePlotter(sp.ax)
medians = []
slopes = []

for i in range(240):
    df = shuffle(temp, blocks)
    slopes.append(ch6util.fit(df))
    medians.append(ch6util.diff_median(df))

    if i < 5:
        cp.plot(df.index, df.values)

sp.label(ylabel_params=dl.data.Weather.get_header('TEMP'))
```

5. Plot the distribution of the first difference medians using the bootstrapped data:

```
sns.distplot(medians, ax=sp.next_ax(), norm_hist=True)
sp.label()
```

6. Plot the distribution of the linear regression slopes using the bootstrapped data:

```
sns.distplot(slopes, ax=sp.next_ax(), norm_hist=True)
sp.label()
```

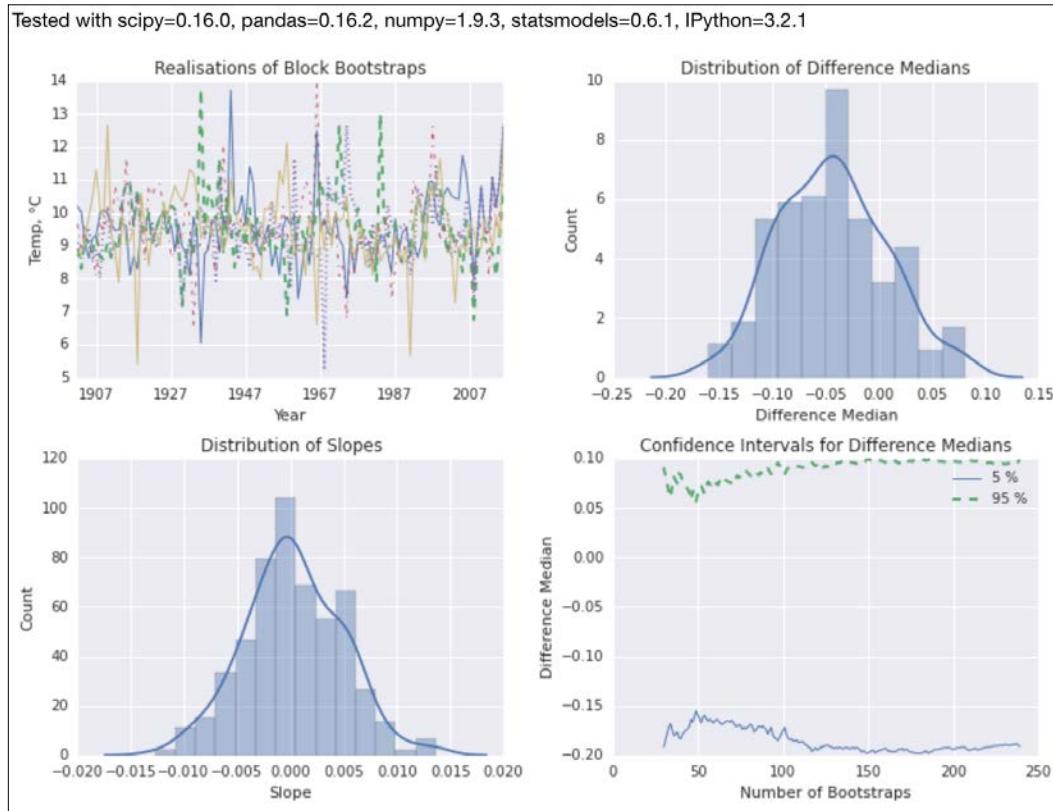
7. Plot the confidence intervals for a varying number of bootstraps:

```
mins = []
tops = []
xrng = range(30, len(medians))

for i in xrng:
    min, max = dl.stats.outliers(medians[:i])
    mins.append(min)
    tops.append(max)

cp = dl.plotting.CyclePlotter(sp.next_ax())
cp.plot(xrng, mins, label='5 %')
cp.plot(xrng, tops, label='95 %')
sp.label()
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The following code comes from the `block_boot.ipynb` file in this book's code bundle.

See also

- ▶ The relevant Wikipedia page at https://en.wikipedia.org/wiki/Bootstrap_pinging_%28statistics%29#Block_bootstrap (retrieved September 2015)

Moving block bootstrapping time series data

If you followed along with the *Block bootstrapping time series data* recipe, you are now aware of a simple bootstrapping scheme for time series data. The **moving block bootstrapping** algorithm is a bit more complicated. In this scheme, we generate overlapping blocks by moving a fixed size window, similar to the moving average. We then assemble the blocks to create new data samples.

In this recipe, we will apply the moving block bootstrap to annual temperature data to generate lists of second difference medians and the slope of an AR(1) model. This is an autoregressive model with lag 1. Also, we will try to neutralize outliers and noise with a median filter.

How to do it...

The following code snippets are from the `moving_boot.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import dautil as dl
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import ch6util
from scipy.signal import medfilt
from IPython.display import HTML
```

2. Define the following function to bootstrap the data:

```
def shuffle(temp):
    indices = np.random.choice(start, n/12)
    sample = dl.collect.flatten([temp.values[i: i + 12] for i in
                                indices])
    sample = medfilt(sample)
    df = pd.DataFrame({'TEMP': sample}, index=temp.
index[:len(sample)])
    df = df.resample('A', how=np.median)

    return df
```

3. Load the data as follows:

```
temp = dl.data.Weather.load()['TEMP'].resample('M', how=np.
median).dropna()
n = len(temp)
start = np.arange(n - 11)
np.random.seed(2609787)
```

-
4. Plot a few random realizations as a sanity check:

```
sp = dl.plotting.Subplotter(2, 2, context)
cp = dl.plotting.CyclePlotter(sp.ax)
medians = []
slopes = []

for i in range(240):
    df = shuffle(temp)
    slopes.append(dl.ts.ar1(df.values.flatten())['slope'])
    medians.append(ch6util.diff_median(df, 2))

    if i < 5:
        cp.plot(df.index, df.values)

sp.label(ylabel_params=dl.data.Weather.get_header('TEMP'))
```

5. Plot the distribution of the second difference medians using the bootstrapped data:

```
sns.distplot(medians, ax=sp.next_ax())
sp.label()
```

6. Plot the distribution of the AR(1) model slopes using the bootstrapped data:

```
sns.distplot(slopes, ax=sp.next_ax())
sp.label()
```

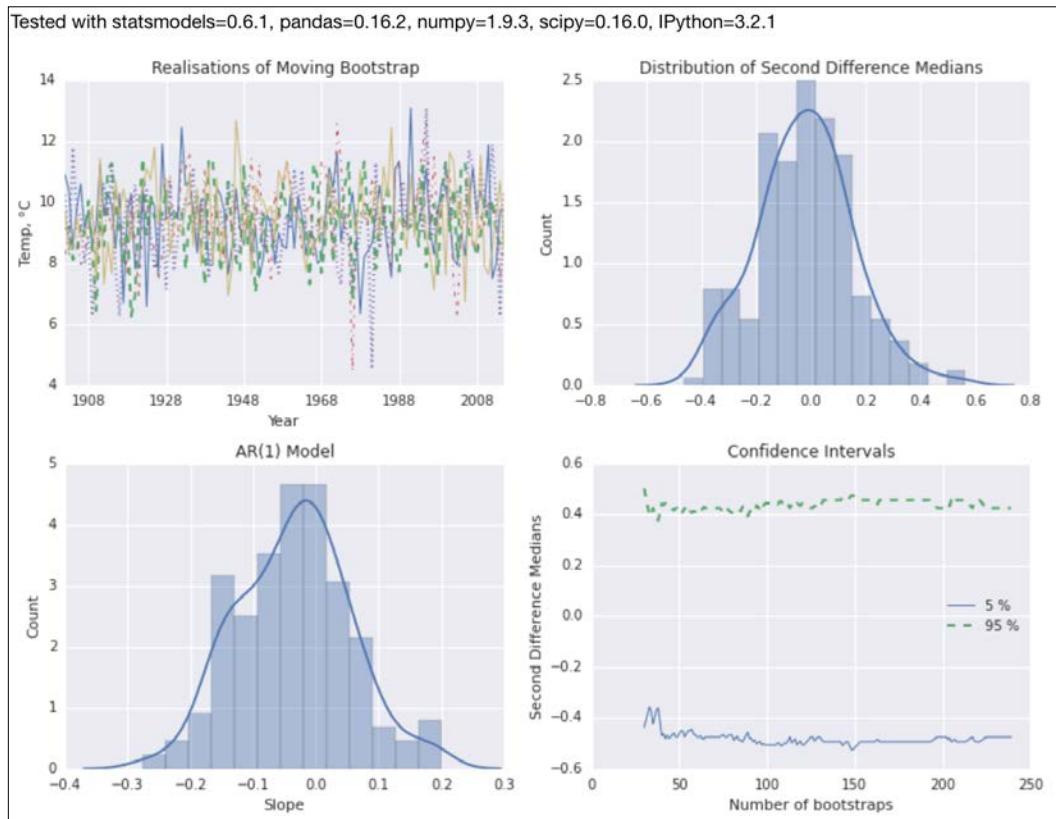
7. Plot the confidence intervals for a varying number of bootstraps:

```
mins = []
tops = []
xrng = range(30, len(medians))

for i in xrng:
    min, max = dl.stats.outliers(medians[:i])
    mins.append(min)
    tops.append(max)

cp = dl.plotting.CyclePlotter(sp.next_ax())
cp.plot(xrng, mins, label='5 %')
cp.plot(xrng, tops, label='95 %')
sp.label()
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



See also

- ▶ The relevant Wikipedia page at https://en.wikipedia.org/wiki/Bootstrap_pinging_%28statistics%29#Block_bootstrap (retrieved September 2015)
- ▶ The `medfilt()` documentation at <https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.medfilt.html> (retrieved September 2015)

Applying the discrete wavelet transform

The **discrete wavelet transform (DWT)** captures information in both the time and frequency domains. The mathematician Alfred Haar created the first wavelet. We will use this **Haar wavelet** in this recipe too. The transform returns **approximation** and **detail coefficients**, which we need to use together to get the original signal back. The approximation coefficients are the result of a low-pass filter. A high-pass filter produces the detail coefficients. The Haar wavelet algorithm is of order O(n) and, similar to the STFT algorithm (refer to the *Analyzing the frequency spectrum of audio* recipe), combines frequency and time information.

The difference with the Fourier transform is that we express the signal as a sum of sine and cosine terms, while the wavelet is represented by a single wave (wavelet function). Just as in the STFT, we split the signal in the time domain and then apply the wavelet function to each segment. The DWT can have multiple levels in this recipe, we don't go further than the first level. To obtain the next level, we apply the wavelet to the approximation coefficients of the previous level. This means that we can have multiple level detail coefficients.

As the dataset, we will have a look at the famous Nile river flow, which even the Greek historian Herodotus wrote about. More recently, in the previous century, the hydrologist Hurst discovered a power law for the **rescaled range** of the Nile river flow in the year. Refer to the See also section for more information. The rescaled range is not difficult to compute, but there are lots of steps as described in the following equations:

$$(6.12) \quad Y_t = X_t - m \text{ for } t = 1, 2, \dots, n$$

$$(6.13) \quad Z_t = \sum_{i=1}^t Y_i \text{ for } t = 1, 2, \dots, n$$

$$(6.14) \quad R_t = \max(Z_1, Z_2, \dots, Z_t) - \min(Z_1, Z_2, \dots, Z_t) \text{ for } t = 1, 2, \dots, n$$

$$(6.15) \quad S_t = \sqrt{\frac{1}{t} \sum_{i=1}^t (X_i - m(t))^2} \text{ for } t = 1, 2, \dots, n$$

$$(6.16) \quad (R / S)_t = \frac{R_t}{S_t} \text{ for } t = 1, 2, \dots, n$$

The **Hurst exponent** from the power law is an indicator of trends. We can also get the Hurst exponent with a more efficient procedure from the wavelet coefficients.

Getting started

Install pywavelets, as follows:

```
$ pip install pywavelets
```

I used pywavelets 0.3.0 for this recipe.

How to do it...

1. The imports are as follows:

```
from statsmodels import datasets
import matplotlib.pyplot as plt
import pywt
import pandas as pd
import dautil as dl
import numpy as np
import seaborn as sns
import warnings
from IPython.display import HTML
```

2. Filter warnings as follows (optional step):

```
warnings.filterwarnings(action='ignore',
                        message='.*Mean of empty slice.*')
warnings.filterwarnings(action='ignore',
                        message='.*Degrees of freedom <= 0 for
slice.*')
```

3. Define the following function to calculate the rescaled range:

```
def calc_rescaled_range(X):
    N = len(X)

    # 1. Mean
    mean = X.mean()

    # 2. Y mean adjusted
    Y = X - mean

    # 3. Z cumulative deviates
    Z = np.array([Y[:i].sum() for i in range(N)])

    # 4. Range R
    R = np.array([0] + [np.ptp(Z[:i]) for i in range(1, N)])
```

```

# 5. Standard deviation S
S = np.array([X[:i].std() for i in range(N)])

# 6. Average partial R/S
return [np.nanmean(R[:i]/S[:i]) for i in range(N)]

```

4. Load the data and transform it with a Haar wavelet:

```

data = datasets.get_rdataset('Nile', cache=True).data
cA, cD = pywt.dwt(data['Nile'].values, 'haar')
coeff = pd.DataFrame({'cA': cA, 'cD': cD})

```

5. Plot the Nile river flow as follows:

```

sp = dl.plotting.Subplotter(2, 2, context)
sp.ax.plot(data['time'], data['Nile'])
sp.label()

```

6. Plot the approximation and detail coefficients of the transformed data:

```

cp = dl.plotting.CyclePlotter(sp.next_ax())
cp.plot(range(len(cA)), cA, label='Approximation coefficients')
cp.plot(range(len(cD)), cD, label='Detail coefficients')
sp.label()

```

7. Plot the rescaled ranges of the coefficients as follows:

```

sp.next_ax().loglog(range(len(cA)), calc_rescaled_range(cA),
                     label='Approximation coefficients')
sp.ax.loglog(range(len(cD)), calc_rescaled_range(cD),
              label='Detail coefficients')
sp.label()

```

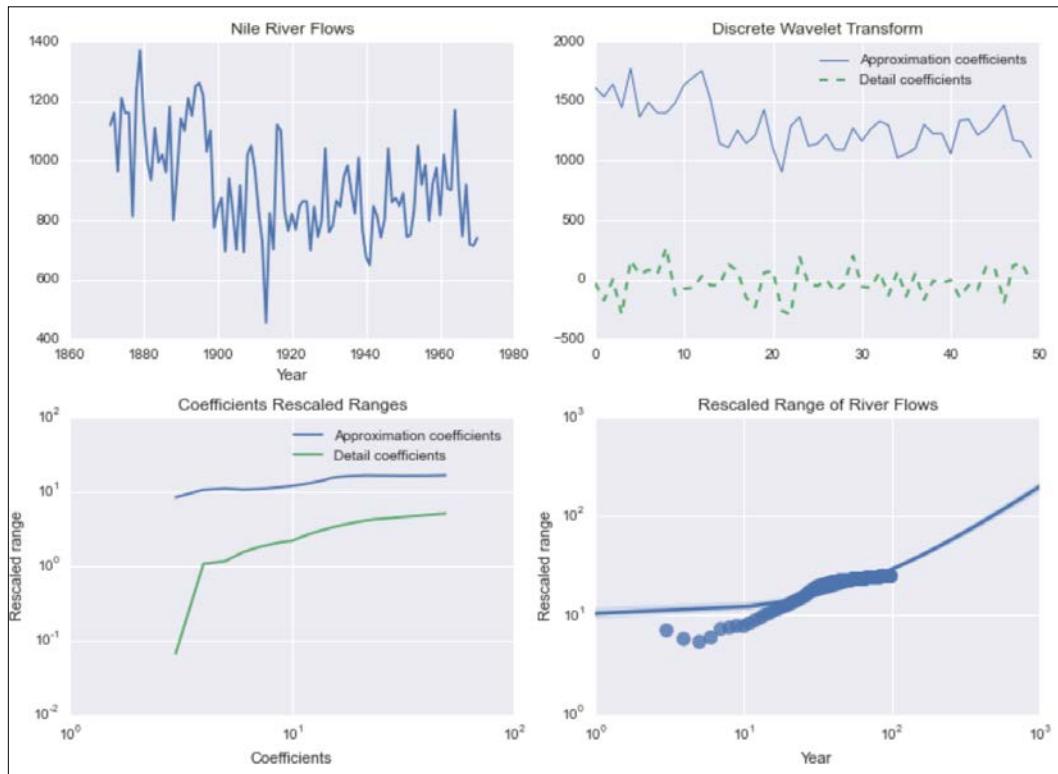
8. Plot the rescaled ranges of the Nile river flow data with a fit:

```

range_df = pd.DataFrame(data={'Year': data.index,
                               'Rescaled range':
                                   calc_rescaled_range(data['Nile'])})
sp.next_ax().set(xscale="log",yscale="log")
sns.regplot('Year', 'Rescaled range', range_df, ax=sp.ax, order=1,
            scatter_kws={"s": 100})
sp.label()
HTML(sp.exit())

```

Refer to the following screenshot for the end result:



The relevant code is in the `discrete_wavelet.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about the discrete wavelet transform at https://en.wikipedia.org/wiki/Discrete_wavelet_transform (retrieved September 2015)
- ▶ The Wikipedia page about the rescaled range at https://en.wikipedia.org/wiki/Rescaled_range (retrieved September 2015)
- ▶ The Wikipedia page about the Hurst exponent at https://en.wikipedia.org/wiki/Hurst_exponent (retrieved September 2015)

7

Selecting Stocks with Financial Data Analysis

In this chapter, we will cover the following recipes:

- ▶ Computing simple and log returns
- ▶ Ranking stocks with the Sharpe ratio and liquidity
- ▶ Ranking stocks with the Calmar and Sortino ratios
- ▶ Analyzing returns statistics
- ▶ Correlating individual stocks with the broader market
- ▶ Exploring risk and return
- ▶ Examining the market with the non-parametric runs test
- ▶ Testing for random walks
- ▶ Determining market efficiency with autoregressive models
- ▶ Creating tables for a stock prices database
- ▶ Populating the stock prices database
- ▶ Optimizing an equal weights two-asset portfolio

Introduction

Finance deals with many subjects, such as money, saving, investing, and insurance. In this chapter, we will focus on stock investing because stock price data is abundant. According to academic theory, an average investor should not invest in individual stocks, but in whole markets, for instance, a basket of stocks representing large companies within a country. Economists make several such arguments for this theory. First, financial markets are random; therefore, beating an average basket by picking stocks is very difficult. Second, individual stocks are volatile with wild price swings. These price moves get averaged in a basket, which makes investing in a group of stocks less risky.

We will analyze stock prices, but nothing prevents you from reusing the recipes to analyze mutual funds and exchange traded funds or other financial assets. To keep the analysis simple, I limited the selection to half a dozen stocks for well-known U.S. companies, which are also represented in the S&P 500 stock index.

Computing simple and log returns

Returns measure the rate of change of (stock) prices. The advantage of using returns is that returns are dimensionless, so we can easily compare the returns of different financial securities. In contrast, the price of financial assets alone doesn't tell us much. In this chapter, we calculate daily returns because our data is sampled daily. With small adjustments, you should be able to apply the same analysis on different time frames.

In fact, there are various types of returns. For the purpose of basic analysis, we only need to know about simple (7.1) and log(arithmetic) returns (7.2), as given by the following equations:

$$(7.1) \quad r = \frac{V_f - V_i}{V_i}$$

$$(7.2) \quad R = \ln\left(\frac{V_f}{V_i}\right)$$

Actually these types of returns can easily be converted – from simple to log returns and back. Log returns are the ones you should prefer if you are given the choice, because they are easier to compute.

How to do it...

1. The imports are as follows:

```
import dautil as dl
import ch7util
import matplotlib.pyplot as plt
```

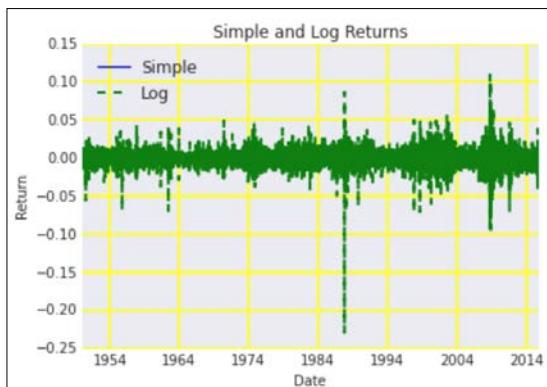
2. Download data for the S&P 500 index:

```
ohlc = dl.data.OHLC()
sp500 = ohlc.get('^GSPC') ['Adj Close']
rets = sp500[1:]/sp500[:-1] - 1
```

3. Plot the simple and log returns:

```
_, ax = plt.subplots()
cp = dl.plotting.CyclePlotter(ax)
cp.plot(sp500.index, rets, label='Simple')
cp.plot(sp500.index[1:], ch7util.log_rets(sp500), label='Log')
ax.set_title('Simple and Log Returns')
ax.set_xlabel('Date')
ax.set_ylabel('Return')
ax.legend(loc='best')
```

Refer to the following screenshot for the end result (the values of simple and log returns are very close):



The code for this recipe is in the `simple_log_rets.ipynb` file in this book's code bundle.

See also

- ▶ The relevant Wikipedia page at https://en.wikipedia.org/wiki/Rate_of_return (retrieved October 2015)

Ranking stocks with the Sharpe ratio and liquidity

The **Sharpe ratio**, defined by William Sharpe, is a fundamental investing metric. The ratio is given as follows:

$$(7.3) \quad S_a = \frac{E[R_a - R_b]}{\sigma_a} = \frac{E[R_a - R_b]}{\sqrt{\text{var}[R_a - R_b]}}$$

The ratio depends on the returns of the asset and the returns of a benchmark. We will use the S&P 500 index as the benchmark. The ratio is supposed to represent a reward to risk ratio. We want to maximize reward while minimizing risk, which corresponds to maximizing the Sharpe ratio.

Another important investing variable is liquidity. Cash is the ultimate liquid asset, but most other assets are less liquid, which means that they change value when we try to sell or buy them. We will use trading volume in this recipe as a measure of liquidity. (Trading volume corresponds to the number of transactions for a financial asset. Liquidity measures how liquid an asset is—how easy it is to buy or sell it.)

How to do it...

You can find the code in the `sharpe_liquidity.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import numpy as np
import dautil as dl
import matplotlib.pyplot as plt
import ch7util
```

2. Define the following function to calculate the ratio and logarithm of the average trading volume:

```
def calc_metrics(ticker, ohlc):
    stock = ohlc.get(ticker)
    sp500 = ohlc.get('^GSPC')
    merged = ch7util.merge_sp500(stock, sp500)
    rets_stock = ch7util.log_rets(merged['Adj Close_stock'])
    rets_sp500 = ch7util.log_rets(merged['Adj Close_sp500'])
    stock_sp500 = rets_stock - rets_sp500
    sharpe_stock = stock_sp500.mean()/stock_sp500.std()
```

```
avg_vol = np.log(merged['Volume_stock'].mean())

return (sharpe_stock, avg_vol)
```

3. Calculate the metrics for our basket of stocks from the ch7util module:

```
dfb = dl.report.DFBuilder(cols=['Ticker', 'Sharpe', 'Log(Average Volume)'])

ohlc = dl.data.OHLC()

for symbol in ch7util.STOCKS:
    sharpe, vol = calc_metrics(symbol, ohlc)
    dfb.row([symbol, sharpe, vol])

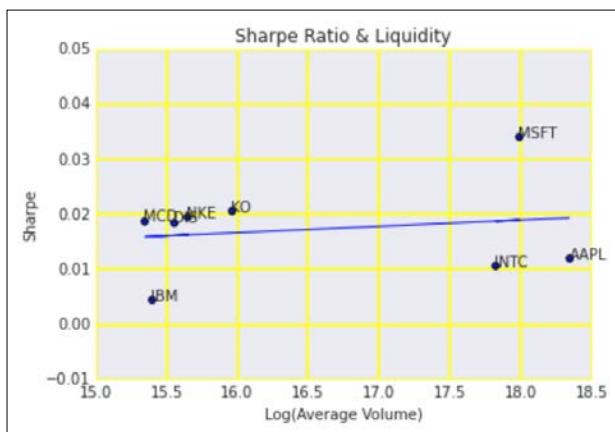
df = dfb.build(index=ch7util.STOCKS)
```

4. Plot the ratio and logarithm average volume for the stocks:

```
_, ax = plt.subplots()
ax.scatter(df['Log(Average Volume)'], df['Sharpe'])
dl.plotting.plot_polyfit(ax, df['Log(Average Volume)'],
df['Sharpe'])

dl.plotting.plot_text(ax, df['Log(Average Volume)'],
df['Sharpe'], ch7util.STOCKS)
ax.set_xlabel('Log(Average Volume)')
ax.set_ylabel('Sharpe')
ax.set_title('Sharpe Ratio & Liquidity')
```

Refer to the following screenshot for the end result:



See also

- ▶ The relevant Wikipedia page at https://en.wikipedia.org/wiki/Sharpe_ratio (retrieved October 2015)

Ranking stocks with the Calmar and Sortino ratios

The **Sortino** and **Calmar ratios** are performance ratios comparable to the Sharpe ratio (refer to the *Ranking stocks with the Sharpe ratio and liquidity* recipe). There are even more ratios; however, the Sharpe ratio has been around the longest, and is therefore very widely used.

The Sortino ratio is named after Frank Sortino, but it was defined by Brian Rom. The ratio defines risk as a downside variance below a benchmark. The benchmark can be an index or a fixed return such as zero. The ratio is defined as follows:

$$(7.4) \quad S = \frac{R - T}{DR}$$

R is the return of the asset, T the target benchmark, and DR the downside risk. The Calmar ratio was invented by Terry Young and was named after his company and newsletter. This ratio defines risk as the **maximum drawdown** (price fall from a peak to a bottom) of an asset.

How to do it...

The following is a breakdown of the `calmar_sortino.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import numpy as np
import dautil as dl
import ch7util
from scipy.signal import argrelemin
from scipy.signal import argrelemax
import matplotlib.pyplot as plt
```

2. Define the following function to calculate the Sortino ratio:

```
def calc_sortino(rets):
    # Returns below target
    semi_var = rets[rets < 0] ** 2
```

```

semi_var = semi_var.sum()/len(rets)
sortino = np.sqrt(semi_var)

return rets.mean()/sortino

```

3. Define the following function to calculate the Calmar ratio:

```

def calc_calmar(rets):
    # Peaks and bottoms indexes in sequence
    mins = np.ravel(argrelmin(rets))
    maxs = np.ravel(argrelmax(rets))
    extrema = np.concatenate((mins, maxs))
    extrema.sort()

    return -rets.mean()/np.diff(rets[extrema]).min()

```

4. Compute the Calmar and Sortino ratios for our list of stocks:

```

ohlc = dl.data.OHLC()
dfb = dl.report.DFBuilder(cols=['Ticker', 'Sortino', 'Calmar'])

for symbol in ch7util.STOCKS:
    stock = ohlc.get(symbol)
    rets = ch7util.log_rets(stock['Adj Close'])
    sortino = calc_sortino(rets)
    calmar = calc_calmar(rets)
    dfb.row([symbol, sortino, calmar])

df = dfb.build(index=ch7util.STOCKS).dropna()

```

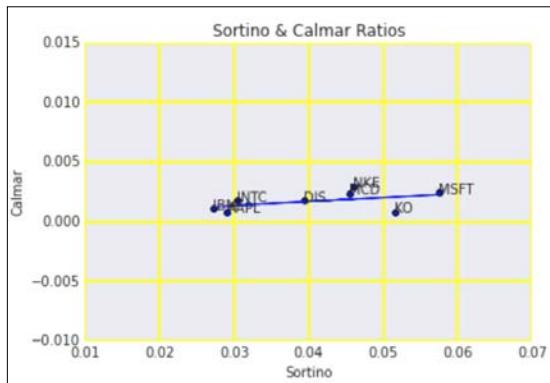
5. Plot the Sortino and Calmar ratios for the stocks:

```

_, ax = plt.subplots()
ax.scatter(df['Sortino'], df['Calmar'])
dl.plotting.plot_polyfit(ax, df['Sortino'], df['Calmar'])
dl.plotting.plot_text(ax, df['Sortino'], df['Calmar'], ch7util.
STOCKS)
ax.set_xlabel('Sortino')
ax.set_ylabel('Calmar')
ax.set_title('Sortino & Calmar Ratios')

```

Refer to the following screenshot for the end result:



See also

- ▶ The Wikipedia page about the Sortino ratio at https://en.wikipedia.org/wiki/Sortino_ratio (retrieved October 2015)
- ▶ The Wikipedia page about the Calmar ratio at https://en.wikipedia.org/wiki/Calmar_ratio (retrieved October 2015)

Analyzing returns statistics

Returns, especially of stock indices, have been extensively studied. In the past, it was assumed that the returns are normally distributed. However, it is now clear that the returns distribution has fat tails (fatter than normal distributions). More information is available at https://en.wikipedia.org/wiki/Fat-tailed_distribution (retrieved October 2015). It is easy enough to check whether data fits the normal distribution. All we need is the mean and standard deviation of the sample.

There are a number of topics that we will explore in this recipe:

- ▶ The skewness and kurtosis of stock returns are interesting to study. Skewness is especially important in the context of stock option models. Analysts usually limit themselves to the mean and standard deviation, which are assumed to correspond to reward and risk, respectively.
- ▶ If we are interested in the existence of a trend, then we should take a look at an autocorrelation plot. This is a plot of autocorrelation—that is, correlation between a signal and the signal at a certain lag (also explained in *Python Data Analysis*, Packt Publishing).

- We will also plot negative returns (absolute value thereof) and corresponding counts on a log-log scale, as those approximately seem to follow a power law (especially the tail values).

How to do it...

The analysis can be found in the `rets_stats.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import dautil as dl
import ch7util
import matplotlib.pyplot as plt
from scipy.stats import skew
from scipy.stats import kurtosis
from pandas.tools.plotting import autocorrelation_plot
import numpy as np
from scipy.stats import norm
from IPython.display import HTML
```

2. Calculate returns for our stocks:

```
ohlc = dl.data.OHLC()
rets_dict = {}

for i, symbol in enumerate(ch7util.STOCKS):
    rrets = ch7util.log_rets(ohlc.get(symbol))['Adj Close']
    rrets_dict[symbol] = rrets

sp500 = ch7util.log_rets(ohlc.get('^GSPC'))['Adj Close']
```

3. Plot the histogram of the S&P 500 returns and corresponding theoretical normal distribution:

```
sp = dl.plotting.Subplotter(2, 2, context)
sp.ax.set_xlim(-0.05, 0.05)
_, bins, _ = sp.ax.hist(sp500, bins=dl.stats.sqrt_bins(sp500),
                       alpha=0.6, normed=True)
sp.ax.plot(bins, norm.pdf(bins, sp500.mean(), sp500.std()), lw=2)
```

4. Plot the skew and kurtosis of returns:

```
skews = [skew(rets_dict[s]) for s in ch7util.STOCKS]
kurts = [kurtosis(rets_dict[s]) for s in ch7util.STOCKS]
sp.label()

sp.next_ax().scatter(skews, kurts)
dl.plotting.plot_text(sp.ax, skews, kurts, ch7util.STOCKS)
sp.label()
```

5. Plot the autocorrelation plot of the S&P 500 returns:

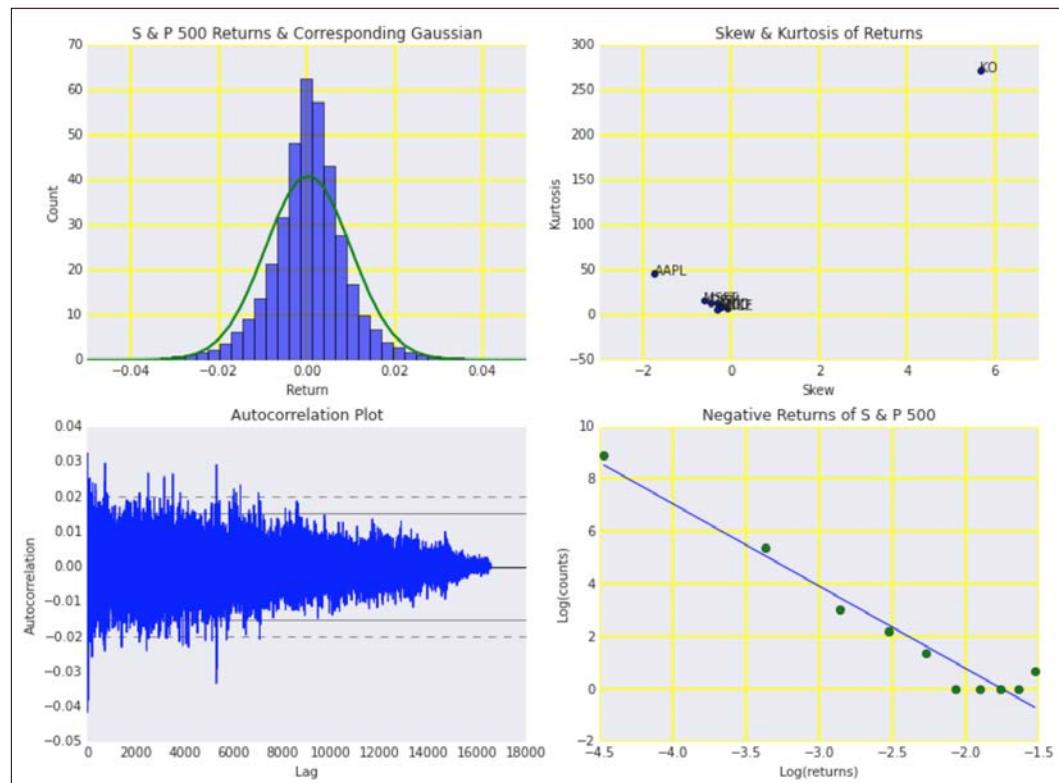
```
autocorrelation_plot(sp500, ax=sp.next_ax())
sp.label()
```

6. Plot the log-log plot of negative returns (absolute value) and counts:

```
# Negative returns
counts, neg_rets = np.histogram(sp500[sp500 < 0])
neg_rets = neg_rets[:-1] + (neg_rets[1] - neg_rets[0])/2
# Adding 1 to avoid log(0)
dl.plotting.plot_polyfit(sp.next_ax(), np.log(np.abs(neg_rets)),
                          np.log(counts + 1), plot_points=True)
sp.label()

HTML(sp.exit())
```

Refer to the following screenshot for the end result:



Correlating individual stocks with the broader market

When we define a stock market or index, we usually choose stocks that are similar in some way. For instance, the stocks might be in the same country or continent. The position of birds can be roughly estimated from the position of the flock they belong to. Similarly, we expect stock returns to be correlated to their market, although not necessarily perfectly.

We will explore the following metrics:

- ▶ The most obvious metric is purportedly the correlation coefficient of the individual stock returns and the S&P 500 index.
- ▶ Another metric is the slope obtained from linear regression instead of correlation.
- ▶ We can also analyze squared differences of returns somewhat similar to squared errors in regression diagnostics.
- ▶ Instead of correlating returns, we can also correlate trading volumes and volatility. To measure volatility, we will use the somewhat uncommon squared value of high and low prices difference. Actually, we are supposed to divide this value by a constant; however, this is not necessary for the correlation coefficient calculation.

How to do it...

The analysis is in the `correlating_market.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import ch7util
import dautil as dl
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import HTML
```

2. Define the following function to compute the volatility:

```
def hl2(df, suffix):
    high = df['High_' + suffix]
    low = df['Low_' + suffix]

    return (high - low) ** 2
```

-
3. Define the following function to correlate the S&P 500 and our stocks:

```
def correlate(stock, sp500):
    merged = ch7util.merge_sp500(stock, sp500)
    rets = ch7util.log_rets(merged['Adj Close_stock'])
    sp500_rets = ch7util.log_rets(merged['Adj Close_sp500'])
    result = {}

    result['corrcoef'] = np.corrcoef(rets, sp500_rets)[0][1]
    slope, _ = np.polyfit(sp500_rets, rets, 1)
    result['slope'] = slope

    srd = (sp500_rets - rets) ** 2
    result['msrd'] = srd.mean()
    result['std_srd'] = srd.std()

    result['vols'] = np.corrcoef(merged['Volume_stock'],
                                merged['Volume_sp500'])[0][1]

    result['hl2'] = np.corrcoef(hl2(merged, 'stock'),
                               hl2(merged, 'sp500'))[0][1]

    return result
```

4. Correlate our set of stocks with the S&P 500 index:

```
ohlc = dl.data.OHLC()
dfs = [ohlc.get(stock) for stock in ch7util.STOCKS]
sp500 = ohlc.get('^GSPC')
corrs = [correlate(df, sp500) for df in dfs]
```

5. Plot correlation coefficients for the stocks:

```
sp = dl.plotting.Subplotter(2, 2, context)
dl.plotting.bar(sp.ax, ch7util.STOCKS,
                [corr['corrcoef'] for corr in corrs])
sp.label()

dl.plotting.bar(sp.next_ax(), ch7util.STOCKS,
                [corr['slope'] for corr in corrs])
sp.label()
```

6. Plot the squared difference statistics:

```
sp.next_ax().set_xlim([0, 0.001])
dl.plotting.plot_text(sp.ax, [corr['msrd'] for corr in corrs],
                      [corr['std_srd'] for corr in corrs],
```

```

    ch7util.STOCKS, add_scatter=True,
    fontsize=9, alpha=0.6)
sp.label()

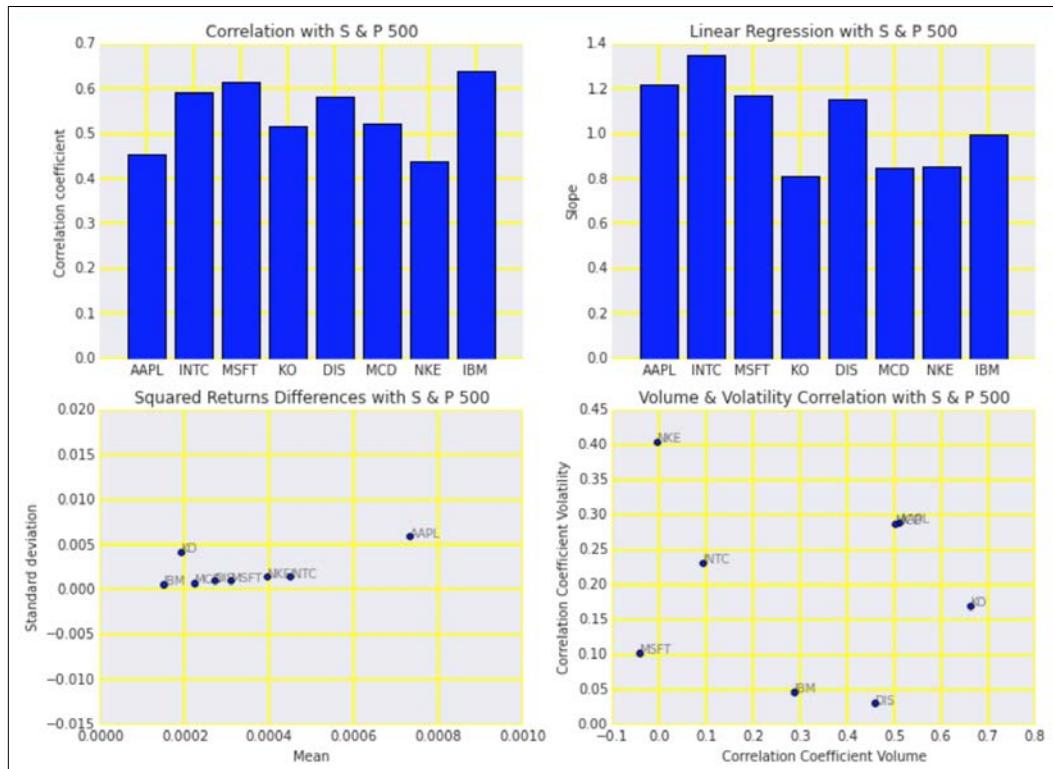
7. Plot volume and volatility correlation coefficients:
d1.plotting.plot_text(sp.next_ax(), [corr['vols'] for corr in
corrs],
[corr['hl2'] for corr in corrs],
ch7util.STOCKS, add_scatter=True,
fontsize=9, alpha=0.6)

sp.label()

HTML(sp.exit())

```

Refer to the following screenshot for the end result:



Exploring risk and return

Beta in finance is the slope of a linear regression model involving the returns of the asset and the returns of a benchmark, for instance the S&P 500 index. The model is defined as follows:

$$(7.5) \quad r_{a,t} = \alpha + \beta r_{b,t} + \varepsilon_t$$

According to the **Capital Asset Pricing Model (CAPM)**, beta is a measure for risk. The expected return is given by the average of the returns. If we plot betas and expected returns for various securities, we obtain the **security market line (SML)** for the corresponding market. The intercept of the SML gives the **risk-free rate**, a return we should theoretically receive without taking any risk. In general, if an asset doesn't lie on the SML, then it is mispriced according to the CAPM.

How to do it...

The program is in the `capm.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import dautil as dl
import numpy as np
import pandas as pd
import ch7util
import matplotlib.pyplot as plt
```

2. Define the following function to calculate the beta:

```
def calc_beta(symbol):
    ohlc = dl.data.OHLC()
    sp500 = ohlc.get('^GSPC') ['Adj Close']
    stock = ohlc.get(symbol) ['Adj Close']
    df = pd.DataFrame({'SP500': sp500, symbol: stock}).dropna()
    sp500_rets = ch7util.log_rets(df['SP500'])
    rets = ch7util.log_rets(df[symbol])
    beta, _ = np.polyfit(sp500_rets, rets, 1)

    # annualize & percentify
    return beta, 252 * rets.mean() * 100
```

3. Compute betas and average returns for our stocks:

```
betas = []
means = []

for symbol in ch7util.STOCKS:
    beta, ret_mean = calc_beta(symbol)
    betas.append(beta)
    means.append(ret_mean)
```

4. Plot the results and the market security line:

```
_, ax = plt.subplots()
dl.plotting.plot_text(ax, betas, means, ch7util.STOCKS, add_scatter=True)
dl.plotting.plot_polyfit(ax, betas, means)
ax.set_title('Capital Asset Pricing Model')
ax.set_xlabel('Beta')
ax.set_ylabel('Mean annual return (%)')
```

Refer to the following screenshot for the end result:



See also

- ▶ The Wikipedia page about beta at https://en.wikipedia.org/wiki/Beta_%28finance%29 (retrieved October 2015)
- ▶ The Wikipedia page about the CAPM at https://en.wikipedia.org/wiki/Capital_asset_pricing_model (retrieved October 2015)

Examining the market with the non-parametric runs test

The **efficient-market hypothesis (EMH)** stipulates that you can't, on average, "beat the market" by picking better stocks or timing the market. According to the EMH, all information about the market is immediately available to every market participant in one form or another, and it is immediately reflected in asset prices, so investing is like playing a game of cards with all the cards revealed. The only way you can win is by betting on very risky stocks and getting lucky.

The French mathematician Bachelor developed a test for the EMH around 1900. The test examines consecutive occurrences of negative and positive price changes. We don't count events during which the price didn't change and only use them to end a run. These types of events are relatively rare anyway for liquid markets.

The statistical test itself is known outside finance and goes by the name of the **Wald-Wolfowitz runs test**. If we denote positive changes with '+' and negative changes with '-', we can have the sequence '+++++---+++-+++++' with 5 runs. The following equations for the mean μ (7.6), standard deviation σ (7.7), and z-score Z (7.8) of the number of runs R also require the number of negative changes N_- , positive changes N_+ , and total number of changes N :

$$(7.6) \quad \mu = \frac{2N_+N_-}{N} + 1$$

$$(7.7) \quad \sigma^2 = \frac{2N_+N_-(2N_+N_- - N)}{N^2(N-1)} = \frac{(\mu-1)(\mu-2)}{N-1}$$

$$(7.8) \quad Z = \frac{R - \mu}{\sigma}$$

We assume that the number of runs follow a normal distribution, which gives us a way to potentially reject the randomness of runs at a confidence level of our choosing.

How to do it...

Have a look at the `non_parametric.ipynb` file in this book's code bundle.

1. The imports are as follows:

```
import dautil as dl
import numpy as np
import pandas as pd
import ch7util
```

```
import matplotlib.pyplot as plt
from scipy.stats import norm
from IPython.display import HTML
```

2. Define the following function to count the number of runs:

```
def count_runs(signs):
    nruns = 0
    prev = None

    for s in signs:
        if s != 0 and s != prev:
            nruns += 1

        prev = s

    return nruns
```

3. Define the following function to calculate the mean, standard deviation, and z-score:

```
def proc_runs(symbol):
    ohlc = dl.data.OHLC()
    close = ohlc.get(symbol)['Adj Close'].values
    diffs = np.diff(close)
    nplus = (diffs > 0).sum()
    nmin = (diffs < 0).sum()
    n = nplus + nmin
    mean = (2 * (nplus * nmin) / n) + 1
    var = (mean - 1) * (mean - 2) / (n - 1)
    std = np.sqrt(var)
    signs = np.sign(diffs)
    nruns = count_runs(np.diff(signs))

    return mean, std, (nruns - mean) / std
```

4. Calculate the metrics for our stocks:

```
means = []
stds = []
zscores = []

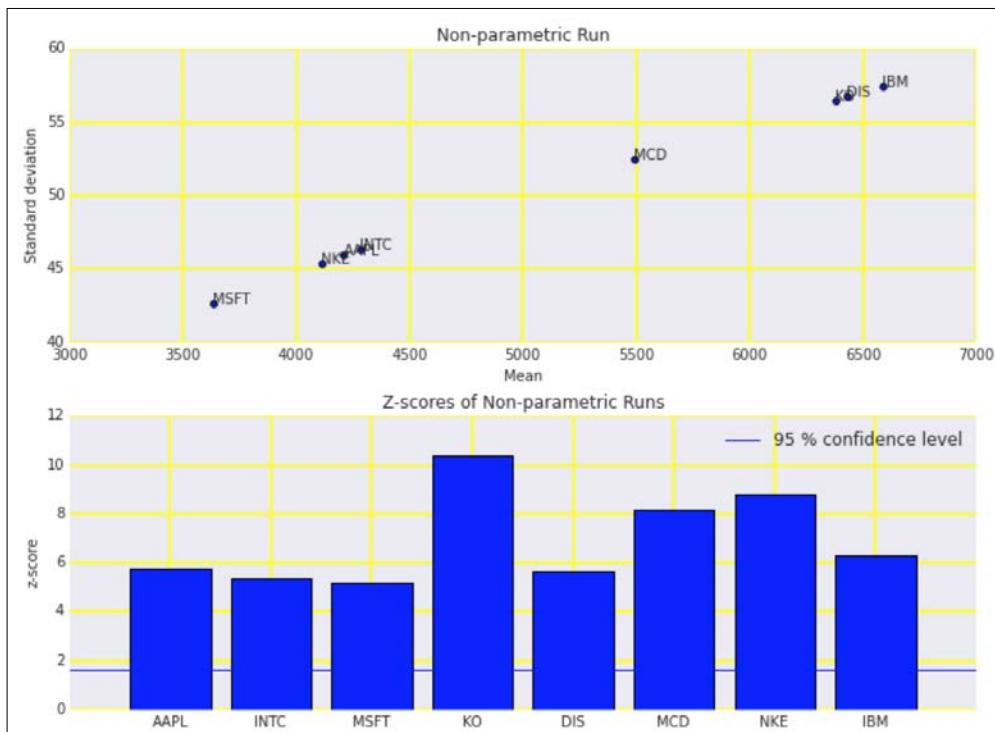
for symbol in ch7util.STOCKS:
    mean, std, zscore = proc_runs(symbol)
    means.append(mean)
    stds.append(std)
    zscores.append(zscore)
```

5. Plot the z-scores with a line indicating the 95% confidence level:

```
sp = dl.plotting.Subplotter(2, 1, context)
dl.plotting.plot_text(sp.ax, means, stds, ch7util.STOCKS, add_
scatter=True)
sp.label()

dl.plotting.bar(sp.next_ax(), ch7util.STOCKS, zscores)
sp.ax.axhline(norm.ppf(0.95), label='95 % confidence level')
sp.label()
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



See also

- The Wikipedia page about the Wald-Wolfowitz runs test at https://en.wikipedia.org/wiki/Wald%20-%20Wolfowitz_runs_test (retrieved October 2015)
- The Wikipedia page about the EMH at https://en.wikipedia.org/wiki/Efficient-market_hypothesis (retrieved October 2015)

Testing for random walks

The **random walk hypothesis (RWH)** just like the efficient-market hypothesis (refer to the *Examining the market with the non-parametric runs test* recipe) claims that the market cannot be beaten. The RWH stipulates that asset prices perform a random walk. You can in fact generate pretty convincing stock price charts just by flipping a coin repeatedly.

In 1988, finance professors Lo and MacKinlay constructed a test for the RWH using the natural log(arithm) of asset prices as data. The test specifies the log prices to drift around a mean (7.9). We expect price changes for different frequencies (for instance, one-day and two-day periods) to be random. Furthermore, the variances (7.10 and 7.11) at two different frequencies are related, and according to the following equations, the corresponding ratio (7.12) is normally distributed around zero:

$$(7.9) \quad \hat{\mu} \equiv \frac{1}{2n} \sum_{k=1}^{2n} (X_k - X_{k-1}) = \frac{1}{2n} (X_{2n} - X_0)$$

$$(7.10) \quad \hat{\sigma}_a^2 \equiv \frac{1}{2n} \sum_{k=1}^{2n} (X_k - X_{k-1} - \hat{\mu})^2$$

$$(7.11) \quad \hat{\sigma}_b^2 \equiv \frac{1}{2n} \sum_{k=1}^n (X_{2k} - X_{2k-2} - 2\hat{\mu})^2$$

$$(7.12) \quad J_r \equiv \frac{\hat{\sigma}_b^2}{\hat{\sigma}_a^2} - 1 \quad \sqrt{2n} J_r \sim N(0, 2)$$

How to do it...

The code is in the `random_walk.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import dautil as dl
import numpy as np
import matplotlib.pyplot as plt
import ch7util
```

2. Calculate the ratios for our stocks:

```
ratios = []

for symbol in ch7util.STOCKS:
    ohlc = dl.data.OHLC()
    P = ohlc.get(symbol)['Adj Close'].values
    N = len(P)
```

```

mu = (np.log(P[-1]) - np.log(P[0]))/N
var_a = 0
var_b = 0

for k in range(1, N):
    var_a = (np.log(P[k]) - np.log(P[k - 1]) - mu) ** 2
    var_a = var_a / N

for k in range(1, N//2):
    var_b = (np.log(P[2 * k]) - np.log(P[2 * k - 2]) - 2 * mu)
** 2
    var_b = var_b / N

ratios.append(np.sqrt(N) * (var_b/var_a - 1))

```

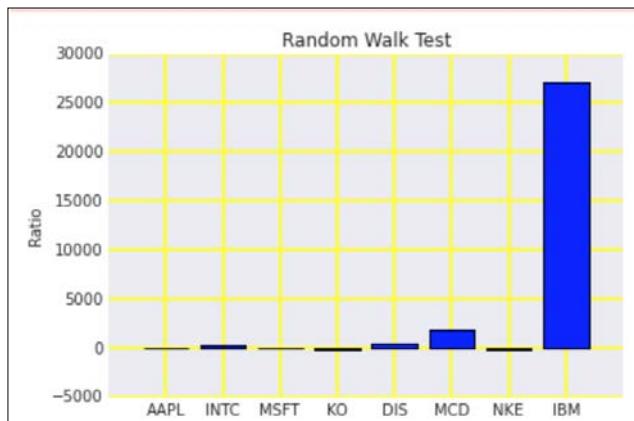
3. Plot the ratios, which we expect to be close to zero (7.12):

```

_, ax = plt.subplots()
dl.plotting.bar(ax, ch7util.STOCKS, ratios)
ax.set_title('Random Walk Test')
ax.set_ylabel('Ratio')

```

Refer to the following screenshot for the end result:



See also

- ▶ The Wikipedia page about the random walk hypothesis at https://en.wikipedia.org/wiki/Random_walk_hypothesis (retrieved October 2015)
- ▶ Rev. Financ. Stud. (1988) 1(1): 41-66. doi: 10.1093/rfs/1.1.41 at <http://rfs.oxfordjournals.org/content/1/1/41.full> (retrieved October 2015)

Determining market efficiency with autoregressive models

According to the efficient-market hypothesis (refer to the *Examining the market with the non-parametric runs test* recipe), all information about an asset is immediately reflected in the price of the asset. This means that previous prices don't influence the current price. The following equations specify an autoregressive model (7. 13) and a restricted model (7. 14) with all the coefficients set to zero:

$$(7.13) \quad X_t = c + \sum_{i=1}^p \varphi_i X_{t-i} + \varepsilon_t$$

$$(7.14) \quad X_t = c + \varepsilon_t$$

$$(7.15) \quad \text{MarketEfficiency} = 1 - \frac{R_{\text{Restricted}}^2}{R_{\text{Unrestricted}}^2}$$

If we believe the market to be efficient, we would expect the unrestricted model to have nothing to add over the restricted model, and, therefore, the ratio (7. 15) of the respective R-squared coefficients should be close to one.

How to do it...

The script is in the `autoregressive_test.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import dautil as dl
import ch7util
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.api as sm
from IPython.display import HTML
```

2. Fit the models using (7.13) and (7.14) and then calculate the market efficiency using (7.15) for our list of stocks:

```
ohlc = dl.data.OHLC()
efficiencies = []
restricted_r2 = []
unrestricted_r2 = []

for stock in ch7util.STOCKS:
    rets = ch7util.log_rets(ohlc.get(stock) ['Adj Close'])
    restricted = sm.OLS(rets, rets.mean() * np.ones_like(rets)) .
    fit()
```

```

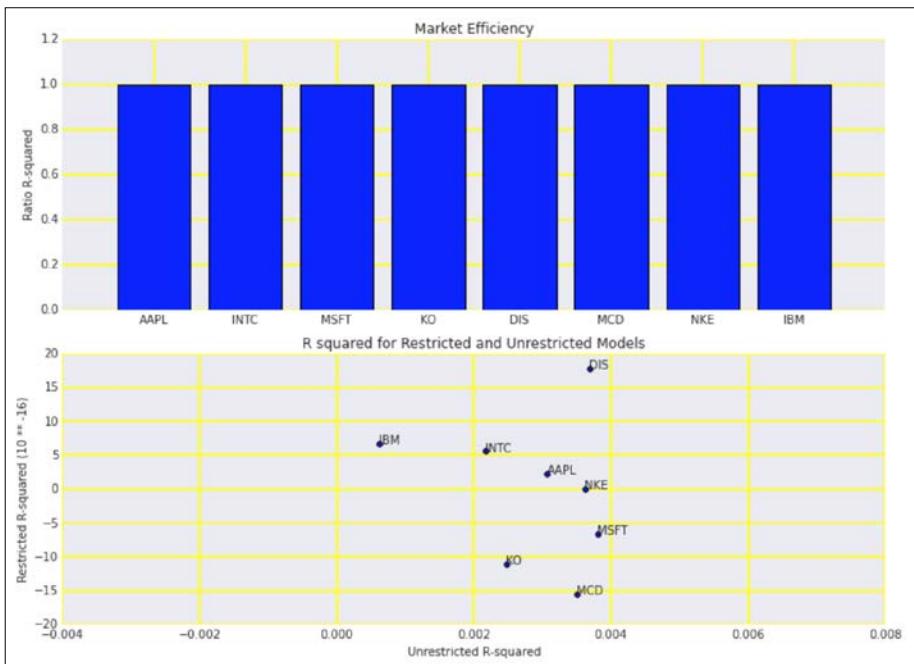
rets_1 = rets[3:-1]
rets_2 = rets[2:-2]
rets_3 = rets[1:-3]
rets_4 = rets[:-4]
x = np.vstack((rets_1, rets_2, rets_3, rets_4)).T
x = sm.add_constant(x)
y = rets[4:]
unrestricted = sm.OLS(y, x).fit()
restricted_r2.append(unrestricted.rsquared)
unrestricted_r2.append(unrestricted.rsquared)
efficiencies.append(1 - restricted.rsquared/unrestricted.rsquared)
    
```

3. Plot the market efficiency and R-squared values as follows:

```

sp = dl.plotting.Subplotter(2, 1, context)
dl.plotting.bar(sp.ax, ch7util.STOCKS, efficiencies)
sp.label()
dl.plotting.plot_text(sp.next_ax(), unrestricted_r2,
np.array(restricted_r2)/10 ** -16,
ch7util.STOCKS, add_scatter=True)
sp.label()
HTML(sp.exit())
    
```

Refer to the following screenshot for the end result:



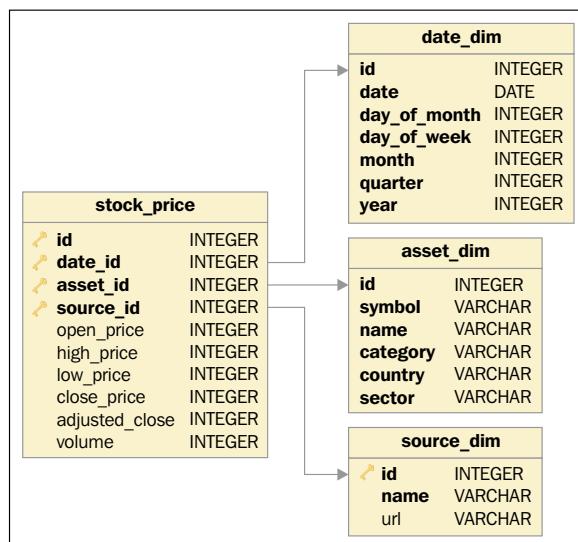
See also

- ▶ The Wikipedia page about the autoregressive model \ at https://en.wikipedia.org/wiki/Autoregressive_model (retrieved October 2015)

Creating tables for a stock prices database

Storing stock prices only is in general not very useful. We usually want to store additional static information about companies and related derivatives such as stock options and futures. Economic theory tells us that looking for cycles and trends in historical price data is more or less a waste of time; therefore, creating a database seems be even more pointless. Of course you don't have to believe the theory, and anyway creating a stock prices database is a fun technical challenge. Also a database is useful for portfolio optimization (see the recipe, *Optimizing an equal weights 2 asset portfolio*).

We will base the design on the star schema pattern covered in *Implementing a star schema with fact and dimension tables*. The fact table will hold the prices, with a date dimension table, asset dimension table, and a source dimension table as in the following diagram:



Obviously, the schema will evolve over time with tables, indexes, and columns added or removed as needed. We will use the schema in the *Populating the stock prices database* recipe.

How to do it...

The schema is defined in the `database_tables.py` file in this book's code bundle:

1. The imports are as follows:

```
from sqlalchemy import Column
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Date
from sqlalchemy import ForeignKey
from sqlalchemy import Integer
from sqlalchemy import String

Base = declarative_base()
```

2. Define the following class for the stock prices fact table:

```
class StockPrice(Base):
    __tablename__ = 'stock_price'
    id = Column(Integer, primary_key=True)
    date_id = Column(Integer, ForeignKey('date_dim.id'),
                      primary_key=True)
    asset_id = Column(Integer, ForeignKey('asset_dim.id'),
                      primary_key=True)
    source_id = Column(Integer, ForeignKey('source_dim.id'),
                      primary_key=True)
    open_price = Column(Integer)
    high_price = Column(Integer)
    low_price = Column(Integer)
    close_price = Column(Integer)
    adjusted_close = Column(Integer)
    volume = Column(Integer)
```

3. Define the following class for the date dimension table:

```
class DateDim(Base):
    __tablename__ = 'date_dim'
    id = Column(Integer, primary_key=True)
    date = Column(Date, nullable=False, unique=True)
    day_of_month = Column(Integer, nullable=False)
    day_of_week = Column(Integer, nullable=False)
    month = Column(Integer, nullable=False)
    quarter = Column(Integer, nullable=False)
    year = Column(Integer, nullable=False)
```

4. Define the following class to hold information about the stocks:

```
class AssetDim(Base):
    __tablename__ = 'asset_dim'
    id = Column(Integer, primary_key=True)
    symbol = Column(String, nullable=False, unique=True)
    name = Column(String, nullable=False)
    # Could make this a reference to separate table
    category = Column(String, nullable=False)
    country = Column(String, nullable=False)
    # Could make this a reference to separate table
    sector = Column(String, nullable=False)
```

5. Define the following class for the source dimension table (we only need one entry for Yahoo Finance):

```
class SourceDim(Base):
    __tablename__ = 'source_dim'
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    url = Column(String)
```

Populating the stock prices database

In the *Creating tables for a stock prices database* recipe, we defined a schema for a historical stock prices database. In this recipe, we will populate the tables with data from Yahoo Finance and plot average volumes for different time frames and business sectors.

Stock market researchers have found several strange phenomena that have to do with seasonal effects. Also, there are certain recurring events such as earnings announcements, dividend payments, and options expirations. Again, economic theory tells us that any patterns we observe are either illusions or already known to all market participants. Whether this is true or not is hard to confirm; however, this recipe is great as exercise in data analysis. Also, you can use the database to optimize your portfolio as explained in the *Optimizing an equal weights 2 asset portfolio* recipe.

How to do it...

The code is in the `populate_database.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import database_tables as tables
import pandas as pd
import os
import dautil as dl
```

```
import ch7util
import sqlite3
import matplotlib.pyplot as plt
import seaborn as sns
from IPython.display import HTML
```

2. Define the following function to populate the date dimension table:

```
def populate_date_dim(session):
    for d in pd.date_range(start='19000101', end='20250101'):
        adate = tables.DateDim(date=d.date(), day_of_month=d.day,
                               day_of_week=d.dayofweek, month=d.
month,
                               quarter=d.quarter, year=d.year)
        session.add(adate)

    session.commit()
```

3. Define the following function to populate the asset dimension table:

```
def populate_asset_dim(session):
    asset = tables.AssetDim(symbol='AAPL', name='Apple Inc.',
                           category='Common Stock',
                           country='USA',
                           sector='Consumer Goods')
    session.add(asset)

    asset = tables.AssetDim(symbol='INTC', name='Intel
Corporation',
                           category='Common Stock',
                           country='USA',
                           sector='Technology')
    session.add(asset)

    asset = tables.AssetDim(symbol='MSFT', name='Microsoft
Corporation',
                           category='Common Stock',
                           country='USA',
                           sector='Technology')
    session.add(asset)

    asset = tables.AssetDim(symbol='KO', name='The Coca-Cola
Company',
                           category='Common Stock',
                           country='USA',
```

```

        sector='Consumer Goods')

session.add(asset)

asset = tables.AssetDim(symbol='DIS', name='The Walt Disney
Company',
                        category='Common Stock',
country='USA',
                        sector='Services')
session.add(asset)

asset = tables.AssetDim(symbol='MCD', name='McDonald\''s
Corp.',
                        category='Common Stock',
country='USA',
                        sector='Services')
session.add(asset)

asset = tables.AssetDim(symbol='NKE', name='NIKE, Inc.',
                        category='Common Stock',
country='USA',
                        sector='Consumer Goods')
session.add(asset)

asset = tables.AssetDim(symbol='IBM',
                        name='International Business Machines
Corporation',
                        category='Common Stock',
country='USA',
                        sector='Technology')
session.add(asset)

session.commit()

```

4. Define the following function to populate the source dimension table:

```

def populate_source_dim(session):
    session.add(tables.SourceDim(name='Yahoo Finance',
                                url='https://finance.yahoo.com'))
    session.commit()

```

5. Define the following function to populate the fact table holding stock prices:

```

def populate_prices(session):
    symbols = dl.db.map_to_id(session, tables.AssetDim.symbol)
    dates = dl.db.map_to_id(session, tables.DateDim.date)
    source_id = session.query(tables.SourceDim).first().id
    ohlc = dl.data.OHLC()

```

```
conn = sqlite3.connect(dbname)
c = conn.cursor()
insert = '''INSERT INTO stock_price (id, date_id,
    asset_id, source_id, open_price, high_price, low_price,
    close_price, adjusted_close, volume) VALUES({id}, {date_
id},
    {asset_id}, {source_id}, {open_price}, {high_price},
    {low_price}, {close_price}, {adj_close}, {volume})'''
logger = dl.log_api.conf_logger(__name__)

for symbol in ch7util.STOCKS:
    df = ohlc.get(symbol)
    i = 0

    for index, row in df.iterrows():
        date_id = dates[index.date()]
        asset_id = symbols[symbol]
        i += 1
        stmt = insert.format(id=i, date_id=date_id,
            asset_id=asset_id,
            source_id=source_id,
            open_price=dl.data.

centify(row['Open']),
            high_price=dl.data.

centify(row['High']),
            low_price=dl.data.

centify(row['Low']),
            close_price=dl.data.

centify(row['Close']),
            adj_close=dl.data.

centify(row['Adj Close']),
            volume=int(row['Volume']))
        c.execute(stmt)

        if i % 1000 == 0:
            logger.info("Progress %s %s", symbol, i)

        conn.commit()

    conn.commit()

    c.close()
    conn.close()
```

6. Define the following function to populate all the tables:

```
def populate(session):
    if session.query(tables.SourceDim).count() == 0:
        populate_source_dim(session)
        populate_asset_dim(session)
        populate_date_dim(session)
        populate_prices(session)
```

7. Define the following function to plot the average volumes:

```
def plot_volume(col, ax):
    df = pd.read_sql(sql.format(col=col), conn)
    sns.barplot(x=col, y='AVG(P.Volume/1000)', data=df,
                 hue='sector', ax=ax)

    ax.legend(loc='best')

dbname = os.path.join(dl.data.get_data_dir(), 'stock_prices.db')
session = dl.db.create_session(dbname, tables.Base)
populate(session)
sql = '''
    SELECT
        A.sector,
        D.{col},
        AVG(P.Volume/1000)
    FROM stock_price P
    INNER JOIN date_dim D ON (P.Date_Id = D.Id)
    INNER JOIN asset_dim A ON (P.asset_id = a.Id)
    GROUP BY
        A.sector,
        D.{col}
'''
```

8. Plot the average volumes with the following code:

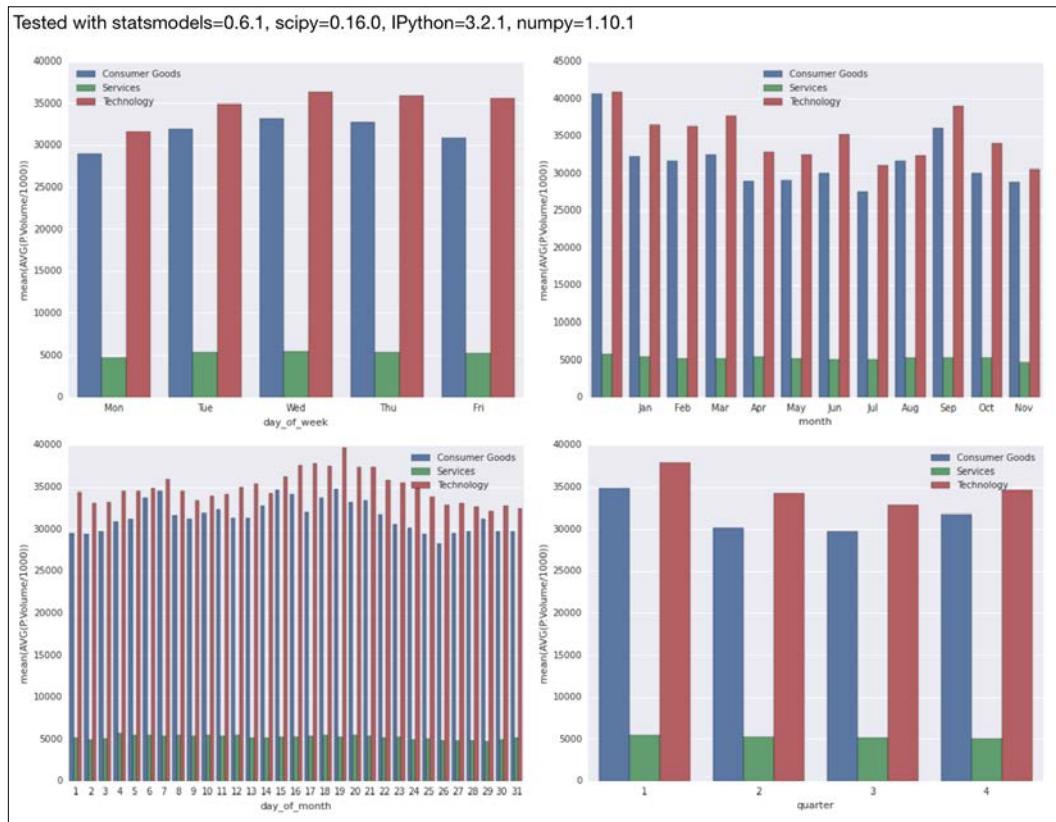
```
conn = sqlite3.connect(dbname)

sp = dl.plotting.Subplotter(2, 2, context)
plot_volume('day_of_week', sp.ax)
sp.ax.set_xticklabels(['Mon', 'Tue', 'Wed', 'Thu', 'Fri'])

plot_volume('month', sp.next_ax())
sp.ax.set_xticklabels(dl.ts.short_months())

plot_volume('day_of_month', sp.next_ax())
plot_volume('quarter', sp.next_ax())
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



Optimizing an equal weights two-asset portfolio

Buying and selling stocks is a bit like shopping. Shopping is something that supermarkets and online bookstores know well. These types of business often apply techniques such as basket analysis and recommendation engines. If, for example, you are a fan of a writer who writes historically inaccurate novels, a recommendation engine will probably recommend another novel by the same writer or other historically inaccurate novels.

A recommendation engine for stocks can't work this way. For instance, if you only have stocks of oil producers in your portfolio and the oil price moves against you, then the whole portfolio will lose value. So, we should try to have stocks from different sectors, industries, or geographical regions. We can measure similarity with the correlation of returns.

Analogous to the Sharpe ratio (refer to the *Ranking stocks with the Sharpe ratio and liquidity* recipe), we want to maximize the average returns of our portfolio and minimize the variance of the portfolio returns. These ideas are also present in the **Modern Portfolio Theory (MPT)**, the inventor of which was awarded the Nobel Prize. For a two-asset portfolio, we have the following equations:

$$(7.16) \quad E(R_p) = w_A E(R_A) + w_B E(R_B) = w_A E(R_A) + (1 - w_A) E(R_B)$$

$$(7.17) \quad \sigma_p^2 = w_A^2 \sigma_A^2 + w_B^2 \sigma_B^2 + 2w_A w_B \sigma_A \sigma_B \rho_{AB}$$

The weights wA and wB are the portfolio weights and sum up to 1. The weights can be negative—as investors can sell short (selling without owning, which incurs borrowing costs) a security. We can solve the portfolio optimization problem with linear algebra methods or general optimization algorithms. However, for a two-asset portfolio with equal weights and a handful of stocks, the brute force approach is good enough.

How to do it...

The following is a breakdown of the `portfolio_optimization.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import dautil as dl
import ch7util
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

2. Define the following function to calculate the expected return (7.16):

```
def expected_return(stocka, stockb, means):
    return 0.5 * (means[stocka] + means[stockb])
```

3. Define the following function to calculate the variance of the portfolio return (7.17):

```
def variance_return(stocka, stockb, stds):
    ohlc = dl.data.OHLC()
    dfa = ohlc.get(stocka)
    dfb = ohlc.get(stockb)
    merged = pd.merge(left=dfa, right=dfb,
                      right_index=True, left_index=True,
                      suffixes=('_A', '_B')).dropna()
    retsa = ch7util.log_rets(merged['Adj Close_A'])
    retsb = ch7util.log_rets(merged['Adj Close_B'])
```

```
corr = np.corrcoef(retsa, retsb)[0][1]

return 0.25 * (stds[stocka] ** 2 + stds[stockb] ** 2 +
               2 * stds[stocka] * stds[stockb] * corr)
```

4. Define the following function to calculate the ratio of the expected return and variance:

```
def calc_ratio(stocka, stockb, means, stds, ratios):
    if stocka == stockb:
        return np.nan

    key = stocka + '_' + stockb
    ratio = ratios.get(key, None)

    if ratio:
        return ratio

    expected = expected_return(stocka, stockb, means)
    var = variance_return(stocka, stockb, stds)
    ratio = expected/var
    ratios[key] = ratio

    return ratio
```

5. Compute the average return and standard deviations for each stock:

```
means = {}
stds = {}

ohlc = dl.data.OHLC()

for stock in ch7util.STOCKS:
    close = ohlc.get(stock) ['Adj Close']
    rets = ch7util.log_rets(close)
    means[stock] = rets.mean()
    stds[stock] = rets.std()
```

6. Calculate the ratios in a grid for all the combinations of our stocks:

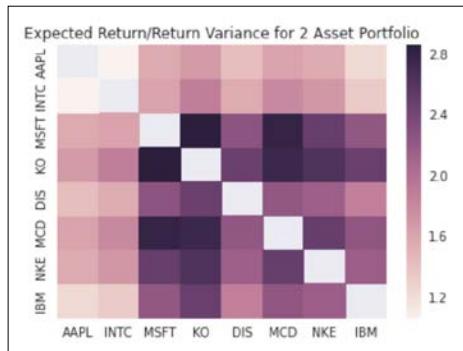
```
pairs = dl.collect.grid_list(ch7util.STOCKS)
sorted_pairs = [[sorted(row[i]) for row in pairs]
                for i in range(len(ch7util.STOCKS))]
ratios = {}

grid = [[calc_ratio(row[i][0], row[i][1], means, stds, ratios)
            for row in sorted_pairs] for i in range(len(ch7util.
STOCKS))]
```

7. Plot the grid in a heatmap as follows:

```
%matplotlib inline  
plt.title('Expected Return/Return Variance for 2 Asset Portfolio')  
sns.heatmap(grid, xticklabels=ch7util.STOCKS, yticklabels=ch7util.  
STOCKS)
```

Refer to the following screenshot for the end result:



See also

- ▶ The Wikipedia page about the MPT at https://en.wikipedia.org/wiki/Modern_portfolio_theory (retrieved October 2015)

8

Text Mining and Social Network Analysis

In this chapter, we will cover the following recipes:

- ▶ Creating a categorized corpus
- ▶ Tokenizing news articles in sentences and words
- ▶ Stemming, lemmatizing, filtering, and TF-IDF scores
- ▶ Recognizing named entities
- ▶ Extracting topics with non-negative matrix factorization
- ▶ Implementing a basic terms database
- ▶ Computing social network density
- ▶ Calculating social network closeness centrality
- ▶ Determining the betweenness centrality
- ▶ Estimating the average clustering coefficient
- ▶ Calculating the assortativity coefficient of a graph
- ▶ Getting the clique number of a graph
- ▶ Creating a document graph with cosine similarity

Introduction

Humans have communicated through language for thousands of years. Handwritten texts have been around for ages, the Gutenberg press was of course a huge development, but now that we have computers, the Internet, and social media, things have definitely spiraled out of control.

This chapter will help you cope with the flood of textual and social media information. The main Python libraries we will use are NLTK and NetworkX. You have to really appreciate how many features can be found in these libraries. Install NLTK with either pip or conda as follows:

```
$ conda/pip install nltk
```

The code was tested with NLTK 3.0.2. If you need to download corpora, follow the instructions given at <http://www.nltk.org/data.html> (retrieved November 2015).

Install NetworkX with either pip or conda, as follows:

```
$ conda/pip install networkx
```

The code was tested with Network 1.9.1.

Creating a categorized corpus

As Pythonistas, we are interested in news about Python programming or related technologies; however, if you search for Python articles, you may also get articles about snakes. One solution for this issue is to train a classifier, which recognizes relevant articles. This requires a training set—a categorized corpus with, for instance, the categories "Python programming" and "other".

NLTK has the `CategorizedPlaintextCorpusReader` class for the construction of a categorized corpus. To make things extra exciting, we will get the links for the news articles from RSS feeds. I chose feeds from the BBC, but of course you can use any other feeds. The BBC feeds are already categorized. I selected the world news and technology news feeds, so this gives us two categories. The feeds don't contain the full text of the articles, hence we need to do a bit of scraping using Selenium as more thoroughly described in *Chapter 5, Web Mining, Databases, and Big Data*. You may need to post-process the text files because the BBC web pages don't contain only the text of the news stories, but also side sections.

Getting ready

Install NLTK following the instructions in the *Introduction* section of this chapter. Install feedparser for the processing of RSS feeds:

```
$ pip/conda install feedparser
```

I tested this recipe with feedparser 5.2.1.

How to do it...

1. The imports are as follows:

```
import feedparser as fp
import urllib
from selenium import webdriver
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.webdriver.common.by import By
import dautil as dl
from nltk.corpus.reader import CategorizedPlaintextCorpusReader
import os
```

2. Create the following variables to help with scraping:

```
DRIVER = webdriver.PhantomJS()
NAP_SECONDS = 10
LOGGER = dl.log_api.conf_logger('corpus')
```

3. Define the following function to store text content:

```
def store_txt(url, fname, title):
    try:
        DRIVER.get(url)

        elems = WebDriverWait(DRIVER, NAP_SECONDS).until(
            EC.presence_of_all_elements_located((By.XPATH, '//p'))
        )

        with open(fname, 'w') as txt_file:
            txt_file.write(title + '\n\n')
            lines = [e.text for e in elems]
            txt_file.write('\n'.join(lines))
    except Exception:
        LOGGER.error("Error processing HTML", exc_info=True)
```

4. Define the following function to retrieve the stories:

```
def fetch_news(dir):
    base = 'http://newsrss.bbc.co.uk/rss/newsonline_uk_edition/{}//'
    rss.xml'

    for category in ['world', 'technology']:
        rss = fp.parse(base.format(category))
```

```
        for i, entry in enumerate(rss.entries):
            fname = '{0}_{1}.txt'.format(i, category)
            fname = os.path.join(dir, fname)

            if not dl.conf.file_exists(fname):
                store_txt(entry.link, fname, entry.title)
```

5. Call the functions with the following code:

```
if __name__ == "__main__":
    dir = os.path.join(dl.data.get_data_dir(), 'bbc_news_corpus')

    if not os.path.exists(dir):
        os.mkdir(dir)

    fetch_news(dir)
    reader = CategorizedPlaintextCorpusReader(dir, r'.*bbc.*\.txt',
                                                cat_pattern=r'.*bbc_(\w+)\.txt')
    printer = dl.log_api.Printer(nelems=3)
    printer.print('Categories', reader.categories())
    printer.print('World fileids', reader.fileids(categories=['world']))
    printer.print('Technology fileids',
                  reader.fileids(categories=['technology']))
```

Refer to the following screenshot for the end result:

```
'Categories'
['technology', 'world']

'World fileids'
['0_bbc_world.txt', '...', '33_bbc_world.txt', '...', '9_bbc_world.txt']

'Technology fileids'
['0_bbc_technology.txt',
 '...',
 '23_bbc_technology.txt',
 '...',
 '9_bbc_technology.txt']
```

The code is in the `corpus.py` file in this book's code bundle.

See also

- ▶ The documentation for `CategorizedPlaintextCorpusReader` at <http://www.nltk.org/api/nltk.corpus.reader.html#nltk.corpus.reader.CategorizedPlaintextCorpusReader> (retrieved October 2015)

Tokenizing news articles in sentences and words

The corpora that are part of the NLTK distribution are already tokenized, so we can easily get lists of words and sentences. For our own corpora, we should apply tokenization too. This recipe demonstrates how to implement tokenization with NLTK. The text file we will use is in this book's code bundle. This particular text is in English, but NLTK supports other languages too.

Getting ready

Install NLTK, following the instructions in the *Introduction* section of this chapter.

How to do it...

The program is in the `tokenizing.py` file in this book's code bundle:

1. The imports are as follows:

```
from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize
import dautil as dl
```

2. The following code demonstrates tokenization:

```
fname = '46_bbc_world.txt'
printer = dl.log_api.Printer(nelems=3)

with open(fname, "r", encoding="utf-8") as txt_file:
    txt = txt_file.read()
    printer.print('Sentences', sent_tokenize(txt))
    printer.print('Words', word_tokenize(txt))
```

Refer to the following screenshot for the end result:

```
'Sentences'
[["The day Iceland's women went on strike\n",
 '\n',
 'Forty years ago, the women of Iceland went on strike - they refused to ',
 'work, cook and look after children for a day.',
 '',
 '...',
 '"I think at first they thought it was something funny, but I can\'t ',
 'remember any of them being angry," says Vigdis.',
 '',
 '...',
 'The male model who slept on the streets \n',
 'Aerial images show how migrant camp at Calais has grown \n',
 'Emergency delivery raises questions in Taiwan']

'Words'
['The', '...', 'Adalheidur', '...', 'Taiwan']
```

See also

- ▶ The relevant documentation is at http://www.nltk.org/api/nltk.tokenize.html?highlight=sent_tokenize#nltk.tokenize.sent_tokenize (retrieved October 2015).

Stemming, lemmatizing, filtering, and TF-IDF scores

The bag-of-words model represents a corpus literally as a bag of words, not taking into account the position of the words—only their count. Stop words are common words such as "a", "is," and "the", which don't add information value.

TF-IDF scores can be computed for single words (**unigrams**) or combinations of multiple consecutive words (**n-grams**). TF-IDF is roughly the ratio of **term frequency** and **inverse document frequency**. I say "roughly" because we usually take the logarithm of the ratio or apply a weighting scheme. Term frequency is the frequency of a word or n-gram in a document. The inverse document frequency is the inverse of the number of documents in which the word or n-gram occurs. We can use TF-IDF scores for clustering or as a feature of classification. In the *Extracting topics with non-negative matrix factorization* recipe, we will use the scores to discover topics.

NLTK represents the scores by a sparse matrix with one row for each document in the corpus and one column for each word or n-gram. Even though the matrix is sparse, we should try to filter words as much as possible depending on the type of problems we are trying to solve. The filtering code is in `ch8util.py` and implements the following operations:

- ▶ Converts all words to lower case. In English, sentences start with upper case and in the bag-of-words model, we don't care about the word position. Obviously, if we want to detect named entities (as in the *Recognizing named entities* recipe), case matters.
- ▶ Ignores stop words, as those have no semantic value.
- ▶ Ignores words consisting of only one character, as those are either stop words or punctuation pretending to be words.
- ▶ Ignores words that only occur once, as those are unlikely to be important.
- ▶ Only allows words containing letters, so ignores a word like "7th" as it contains a digit.

We will also filter with **lemmatization**. Lemmatization is similar to stemming, which I will also demonstrate. The idea behind both procedures is that words have common roots, for instance, the words "analysis," "analyst," and "analysts" have a common root. In general, **stemming** cuts characters, so the result doesn't have to be a valid word. Lemmatization, in contrast, always produces valid words and performs dictionary look-ups.

The code for the `ch8util.py` file in this book's code bundle is as follows:

```
from collections import Counter
from nltk.corpus import brown
from joblib import Memory

memory = Memory(cachedir='.')

def only_letters(word):
    for c in word:
        if not c.isalpha():
            return False

    return True

@memory.cache
def filter(fid, lemmatizer, sw):
    words = [lemmatizer.lemmatize(w.lower()) for w in brown.words(fid)
             if len(w) > 1 and w.lower() not in sw]
```

```
# Ignore words which only occur once
counts = Counter(words)
rare = set([w for w, c in counts.items() if c == 1])

filtered_words = [w for w in words if w not in rare]

return [w for w in filtered_words if only_letters(w)]
```

I decided to limit the analysis to unigrams, but it's quite easy to extend the analysis to bigrams or trigrams. The scikit-learn TfidfVectorizer class that we will use lets us specify a ngram_range field, so we can consider unigrams and n-grams at the same time. We will pickle the results of this recipe to be reused by other recipes.

Getting ready

Install NLTK by following the instructions in the *Introduction* section.

How to do it...

The script is in the `stemming_lemma.py` file in this book's code bundle:

1. The imports are as follows:

```
from nltk.corpus import brown
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
from nltk.stem import WordNetLemmatizer
import ch8util
from sklearn.feature_extraction.text import TfidfVectorizer
import numpy as np
import pandas as pd
import pickle
import dautil as dl
```

2. Demonstrate stemming and lemmatizing as follows:

```
stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()

print('stem(analyses)', stemmer.stem('analyses'))
print('lemmatize(analyses)', lemmatizer.lemmatize('analyses'))
```

3. Filter the words in the NLTK Brown corpus:

```
sw = set(stopwords.words())
texts = []
```

```
fids = brown.fileids(categories='news')

for fid in fids:
    texts.append(" ".join(ch8util.filter(fid, lemmatizer, sw)))
```

4. Calculate TF-IDF scores as follows:

```
vectorizer = TfidfVectorizer()
matrix = vectorizer.fit_transform(texts)

with open('tfidf.pkl', 'wb') as pkl:
    pickle.dump(matrix, pkl)

sums = np.array(matrix.sum(axis=0)).ravel()

ranks = [(word, val) for word, val in
          zip(vectorizer.get_feature_names(), sums)]

df = pd.DataFrame(ranks, columns=["term", "tfidf"])
df.to_pickle('tfidf_df.pkl')
df = df.sort(['tfidf'])
dl.options.set_pd_options()
print(df)
```

Refer to the following screenshot for the end result:

	stem(analyses)	analys
	lemmatize(analyses)	analysis
	term	tfidf
292	beyond	0.035
1460	informed	0.035
...
2736	state	1.831
2478	said	3.236

[3173 rows x 2 columns]

How it works

As you can see, stemming doesn't return a valid word. It is faster than lemmatization; however, if you want to reuse the results, it makes sense to prefer lemmatization. The TF-IDF scores are sorted in ascending order in the final pandas DataFrame object. A higher TF-IDF score indicates a more important word.

See also

- ▶ The Wikipedia page about the bag-of-words model at https://en.wikipedia.org/wiki/Bag-of-words_model (retrieved October 2015)
- ▶ The Wikipedia page about lemmatization at <https://en.wikipedia.org/wiki/Lemmatisation> (retrieved October 2015)
- ▶ The Wikipedia page about the TF-IDF at <https://en.wikipedia.org/wiki/Tf%E2%80%93idf> (retrieved October 2015)
- ▶ The Wikipedia page about stop words at https://en.wikipedia.org/wiki/Stop_words (retrieved October 2015)
- ▶ The documentation for the TfIdfVectorizer class at http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html (retrieved October 2015)
- ▶ The documentation for the WordNetLemmatizer class at <http://www.nltk.org/api/nltk.stem.html#nltk.stem.wordnet.WordNetLemmatizer> (retrieved November 2015)

Recognizing named entities

Named-entity recognition (NER) tries to detect names of persons, organizations, locations, and other names in texts. Some NER systems are almost as good as humans, but it is not an easy task. Named entities usually start with upper case, such as Ivan. We should, therefore, not change the case of words when applying NER.

NLTK has support for the Stanford NER API. This is a Java API, so you need to have Java on your system. I tested the code with Java 1.8.0_65. The code in this recipe downloads the most recent Stanford NER archive (`stanford-ner-2015-04-20.zip/3.5.2`) as of October 2015. If you want another version, take a look at <http://nlp.stanford.edu/software/CRF-NER.shtml> (retrieved October 2015).

Getting ready

Install NLTK by following the instructions in the *Introduction* section. You may also need to install Java.

How to do it...

The script is in the named `_entity.py` file in this book's code bundle:

1. The imports are as follows:

```
from nltk.tag.stanford import NERTagger
import dautil as dl
import os
from zipfile import ZipFile
from nltk.corpus import brown
```

2. Define the following function to download the NER archive:

```
def download_ner():
    url = 'http://nlp.stanford.edu/software/stanford-
ner-2015-04-20.zip'
    dir = os.path.join(dl.data.get_data_dir(), 'ner')

    if not os.path.exists(dir):
        os.mkdir(dir)

    fname = 'stanford-ner-2015-04-20.zip'
    out = os.path.join(dir, fname)

    if not dl.conf.file_exists(out):
        dl.data.download(url, out)

    with ZipFile(out) as nerzip:
        nerzip.extractall(path=dir)

    return os.path.join(dir, fname.replace('.zip', ''))
```

3. Apply NER to one of the files in the Brown corpus:

```
dir = download_ner()
st = NERTagger(os.path.join(dir, 'classifiers',
                            'english.all.3class.distsim.crf.ser.
gz'),
               os.path.join(dir, 'stanford-ner.jar'))
fid = brown.fileids(categories='news')[0]
printer = dl.log_api.Printer(nelems=9)

tagged = [pair for pair in dl.collect.flatten(st.tag(brown.
words(fid)))
          if pair[1] != 'O']
printer.print(tagged)
```

Refer to the following screenshot for the end result:

```
[('Fulton', 'ORGANIZATION'),
 ('County', 'ORGANIZATION'),
 ('Grand', 'ORGANIZATION'),
 ('Jury', 'ORGANIZATION'),
 ('...', 'ORGANIZATION'),
 ('Party', 'ORGANIZATION'),
 ('...', 'PERSON'),
 ('Williams', 'PERSON'),
 ('Williams', 'PERSON'),
 ('Felix', 'PERSON'),
 ('Tabb', 'PERSON')]
```

How it works

We created a `NerTagger` object by specifying a pre-trained classifier and the NER JAR (Java archive). The classifier tagged words in our corpus as organization, location, person, or other. The classification is case sensitive, which means that if you lowercase all the words, you will get different results.

See also

- ▶ The Wikipedia page about NER at https://en.wikipedia.org/wiki/Named-entity_recognition (retrieved October 2015)

Extracting topics with non-negative matrix factorization

Topics in natural language processing don't exactly match the dictionary definition and correspond to more of a nebulous statistical concept. We speak of **topic models** and probability distributions of words linked to topics, as we know them. When we read a text, we expect certain words that appear in the title or the body of the text to capture the semantic context of the document. An article about Python programming will have words like "class" and "function", while a story about snakes will have words like "eggs" and "afraid." Texts usually have multiple topics; for instance, this recipe is about topic models and non-negative matrix factorization, which we will discuss shortly. We can, therefore, define an additive model for topics by assigning different weights to topics.

One of the topic modeling algorithms is **non-negative matrix factorization (NMF)**. This algorithm factorizes a matrix into a product of two matrices in such a way that the two matrices have no negative values. Usually, we are only able to numerically approximate the solution of the factorization and the time complexity is polynomial. The scikit-learn `NMF` class implements this algorithm. NMF can also be applied to document clustering and signal processing.

How to do it...

We will reuse the results from the *Stemming, lemmatizing, filtering, and TF-IDF scores* recipe:

1. The imports are as follows:

```
from sklearn.decomposition import NMF
import ch8util
```

2. Load the TF-IDF matrix and words from a pickle:

```
terms = ch8util.load_terms()
tfidf = ch8util.load_tfidf()
```

3. Visualize topics as lists of high-ranking words:

```
nmf = NMF(n_components=44, random_state=51).fit(tfidf)

for topic_idx, topic in enumerate(nmf.components_):
    label = '{}: '.format(topic_idx)
    print(label, " ".join([terms[i] for i in topic.argsort()[:-9:-1]]))
```

Refer to the following screenshot for the end result:

```
0: council hawksley charter said law cd martinelli town
1: morti mantle run yankee home baseball mickey hit
2: bundle miss chairman daughter robert thrift drexel queen
3: liberal committee congress law rule battle colmer smith
4: gin stock dallas share cotton morton equipment sole
5: library system headquarters book librarian nassau service collection
6: republican mitchell state hughes campaign candidate jones said
7: palmer player hole tournament round golf stroke green
8: fee city project hughes hemphill said license association
9: emory theater student atlanta said acre friday letter
10: sale farm tax income dealer billion farmer august
11: law labor union act price collective bargaining wage
12: portland hillsboro agency junior achievement blue company harvey
13: recovery bank share economy debenture railroad growth camera
14: belgian congo congolese independence lumumba province government kasavubu
15: ballet toy san concert season francisco car television
16: secret submarine narcotic british evanston dreadnaught youth sub
17: catholic faculty institution community college religious sense university
18: puppet car driven formerly beverly ride lamp hill
19: turnpike textile coal bond interest cent traffic revenue
20: kowalski hengesbach verdict neighbor pohl said havana family
21: garson design ramsey table dallas designer contemporary chicken
22: wendell tomorrow chicago monroe club wedding italian burke
23: holmes thompson georgia frankie pittsburgh chicago went university
24: plane president kennedy paso senate beardens rickards bearden
25: cent police said simpkins snow apartment annapolis street
26: oriole robinson hansen double run league kansa single
27: fulton jury county election said highway resolution department
28: game yard moritz meek halfback play texas rice
29: family house music concert gallery white young brevard
30: medical care grant precinct health case million karns
31: stein huff fiedler missile union witness buchheister leavitt
32: khrushchev meeting premier moscow soviet summit negotiation president
33: democratic mayor leader buckley wagner bill coalition bronx
34: administration tax barnett legislature session davis avenue democratic
35: motel miss bride marr honor pool meredith baker
36: church family christian sunday kern library ballet restaurant
37: providence hospital cranston jury board appeal said car
38: skorich shea arnold award league palmer football eagle
39: texas bill dallas adc would school committee austin
40: church god board said school belief secretary district
41: mantle team pirate bennington bob season game may
42: administration too policy communist united state nato oslo
43: game run bear inning liston baseball sox third
```

The code is in the `topic_extraction.py` file in this book's code bundle.

How it works

The NMF class has a `components_` attribute, which holds the non-negative components of the data. We selected the words corresponding to the highest values in the `components_` attribute. As you can see, the topics are varied, although a bit outdated.

See also

- ▶ The documentation for the NMF class at <http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.NMF.html> (retrieved October 2015)
- ▶ The Wikipedia page about topic models at https://en.wikipedia.org/wiki/Topic_model (retrieved October 2015)
- ▶ The Wikipedia page about NMF at https://en.wikipedia.org/wiki/Non-negative_matrix_factorization (retrieved October 2015)

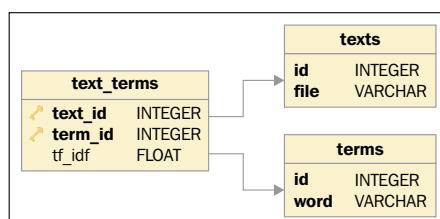
Implementing a basic terms database

As you know, natural language processing has many applications:

- ▶ Full text search as implemented by commercial and open source search engines
- ▶ Clustering of documents
- ▶ Classification, for example to determine the type of text or the sentiment in the context of a product review

To perform these tasks, we need to calculate features such as TF-IDF scores (refer to *Stemming, lemmatizing, filtering, and TF-IDF scores*). Especially, with large datasets, it makes sense to store the features for easy processing. Search engines use inverted indices, which map words to web pages. This is similar to the association table pattern (refer to *Implementing association tables*).

We will implement the association table pattern with three tables. One table contains the words, another will implement the association table pattern with three tables. One table contains the words, another table holds the information about the documents, and the third table links the other two tables as shown in the following schema:



How to do it...

The program is in the `terms_database.py` file in this book's code bundle:

1. The imports are as follows:

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column
from sqlalchemy import ForeignKey
from sqlalchemy import Float
from sqlalchemy import Integer
from sqlalchemy import String
from sqlalchemy.orm import backref
from sqlalchemy.orm import relationship
import os
import dauthil as dl
from nltk.corpus import brown
from sqlalchemy import func
import ch8util

Base = declarative_base()
```

2. Define the following class for the text documents:

```
class Text(Base):
    __tablename__ = 'texts'
    id = Column(Integer, primary_key=True)
    file = Column(String, nullable=False, unique=True)
    terms = relationship('Term', secondary='text_terms')

    def __repr__(self):
        return "Id=%d file=%s" % (self.id, self.file)
```

3. Define the following class for the words in the articles:

```
class Term(Base):
    __tablename__ = 'terms'
    id = Column(Integer, primary_key=True)
    word = Column(String, nullable=False, unique=True)

    def __repr__(self):
        return "Id=%d word=%s" % (self.id, self.word)
```

-
4. Define the following class for the association of documents and words:

```
class TextTerm(Base):
    __tablename__ = 'text_terms'
    text_id = Column(Integer, ForeignKey('texts.id'), primary_
key=True)
    term_id = Column(Integer, ForeignKey('terms.id'), primary_
key=True)
    tf_idf = Column(Float)
    text = relationship('Text', backref=backref('term_assoc'))
    term = relationship('Term', backref=backref('text_assoc'))

    def __repr__(self):
        return "text_id=%s term_id=%s" % (self.text_id, self.term_
id)
```

5. Define the following function to insert entries in the texts table:

```
def populate_texts(session):
    if dl.db.not_empty(session, Text):
        # Cowardly refusing to continue
        return

    fids = brown.fileids(categories='news')

    for fid in fids:
        session.add(Text(file=fid))

    session.commit()
```

6. Define the following function to insert entries in the terms table:

```
def populate_terms(session):
    if dl.db.not_empty(session, Term):
        # Cowardly refusing to continue
        return

    terms = ch8util.load_terms()

    for term in terms:
        session.add(Term(word=term))

    session.commit()
```

7. Define the following function to insert entries in the association table:

```
def populate_text_terms(session):
    if dl.db.not_empty(session, TextTerm):
```

```

# Cowardly refusing to continue
return

text_ids = dl.collect.flatten(session.query(Text.id).all())
term_ids = dl.collect.flatten(session.query(Term.id).all())

tfidf = ch8util.load_tfidf()
logger = dl.log_api.conf_logger(__name__)

for text_id, row, in zip(text_ids, tfidf):
    logger.info('Processing {}'.format(text_id))
    arr = row.toarray()[0]
    session.get_bind().execute(
        TextTerm.__table__.insert(),
        [{ 'text_id': text_id, 'term_id': term_id,
          'tf_idf': arr[i] }
         for i, term_id in enumerate(term_ids)
         if arr[i] > 0]
    )

    session.commit()

```

8. Define the following function to perform a search with keywords:

```

def search(session, keywords):
    terms = keywords.split()
    fsum = func.sum(TextTerm.tf_idf)

    return session.query(TextTerm.text_id, fsum).\
        join(Term, TextTerm).\
        filter(Term.word.in_(terms)).\
        group_by(TextTerm.text_id).\
        order_by(fsum.desc()).all()

```

9. Call the functions we defined with the following code:

```

if __name__ == "__main__":
    dbname = os.path.join(dl.data.get_data_dir(), 'news_terms.db')
    session = dl.db.create_session(dbname, Base)
    populate_texts(session)
    populate_terms(session)
    populate_text_terms(session)
    printer = dl.log_api.Printer()
    printer.print('id, tf_idf', search(session, 'baseball game'))

```

We performed a search for "baseball game." Refer to the following screenshot for the end result (file IDs and TF-IDF sums):

```
[id, tf_idf']
[(12, 2.5665254543738096),
(13, 2.4081218366314148),
(15, 1.5604954671497586),
(39, 1.3231045413703009),
(14, 0.921068156484757),
(11, 0.8502146144930182),
(38, 0.2519025401924042)]
```

How it works

We stored TF-IDF scores using the association table database pattern. As an example of using the database, we queried for "baseball game." The query looked up the IDs of both words in the terms table and then summed the related TF-IDF scores in the association table. The sums serve as a relevancy score. Then, we presented the corresponding file IDs with relevancy scores in descending order. If you are showing the result to end users, you will have to do at least one more query to replace the file IDs with filenames. As it happens, the files we are analyzing are named ca01 to ca44, so the query is not strictly necessary.

Because I had the TF-IDF scores already, I found it convenient to store them directly. However, you can also decide to store the term frequency and inverse document frequency and derive the TF-IDF scores from those. You only need to determine the term frequency for each new document and the words in the document. All the inverse document frequencies need to be updated when documents are added or removed. However, establishing a link via the association table is already enough to calculate the term frequency, inverse document frequency, and TF-IDF scores.

See also

- ▶ The Wikipedia page about search engine indexing at https://en.wikipedia.org/wiki/Search_engine_indexing (retrieved October 2015)

Computing social network density

Humans are social animals and, therefore, social connections are very important. We can view these connections and the persons involved as a network. We represent networks or a subset as a graph. A graph consists of nodes or points connected by edges or lines. Graphs can be directed or undirected—the lines can be arrows.

We will use the Facebook SPAN data, which we also used in the *Visualizing network graphs with hive plots* recipe. Facebook started out small in 2004, but it has more than a billion users as of 2015. The data doesn't include all the users, but it is still enough for a decent analysis. The following equations describe the density of undirected (8.1) and directed (8.2) graphs:

$$(8.1) \quad d = \frac{2m}{n(n-1)}$$

$$(8.2) \quad d = \frac{m}{n(n-1)}$$

In these equations, n is the number of nodes and m is the number of edges.

Getting ready

Install NetworkX with the instructions from the *Introduction* section.

How to do it...

The code is in the `net_density.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import networkx as nx
import dautil as dl
```

2. Create a NetworkX graph as follows:

```
fb_file = dl.data.SPANFB().load()
G = nx.read_edgelist(fb_file,
                     create_using=nx.Graph(),
                     nodetype=int)
```

3. Call the `density()` function as follows:

```
print('Density', nx.density(G))
```

We get the following density:

```
Density 0.010819963503439287
```

See also

- ▶ The `density()` function documented at <https://networkx.github.io/documentation/latest/reference/generated/networkx.classes.function.density.html> (retrieved October 2015)
- ▶ The Wikipedia page about graphs at https://en.wikipedia.org/wiki/Graph_%28abstract_data_type%29 (retrieved October 2015)

Calculating social network closeness centrality

In a social network such as the Facebook SPAN data, we will have influential people. In graph terminology, these are the influential nodes. **Centrality** finds features of important nodes.

Closeness centrality uses shortest paths between nodes as a feature, as shown in the following equation:

$$(8.3) \quad C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)}$$

In (8.3), $d(u, v)$ is the shortest path between u , v , and n is the number of nodes. An influential node is close to other nodes and, therefore, the sum of the shortest paths is low. We can compute closeness centrality for each node separately, and for a large graph, this can be a lengthy calculation. NetworkX allows us to specify which node we are interested in, so we will calculate closeness centrality just for a few nodes.

Getting ready

Install NetworkX with the instructions from the *Introduction* section.

How to do it...

Have a look at the `close_centrality.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import networkx as nx
import dautil as dl
```

2. Create a NetworkX graph from the Facebook SPAN data as follows:

```
fb_file = dl.data.SPANFB().load()
G = nx.read_edgelist(fb_file,
                     create_using=nx.Graph(),
                     nodetype=int)
```

3. Calculate the closeness centrality for node 1 and node 4037:

```
print('Closeness Centrality Node 1',
      nx.closeness_centrality(G, 1))
print('Closeness Centrality Node 4037',
      nx.closeness_centrality(G, 4037))
```

We get the following result for the Facebook SPAN data:

```
Closeness Centrality Node 1 0.2613761408505405
Closeness Centrality Node 4037 0.18400546821599453
```

See also

- ▶ The Wikipedia page about closeness centrality at https://en.wikipedia.org/wiki/Centrality#Closeness_centrality (retrieved October 2015)

Determining the betweenness centrality

Betweenness centrality is a type of centrality similar to closeness centrality (refer to the *Calculating social network closeness centrality* recipe). This metric is given by the following equation:

$$(8.4) \quad c_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$$

It is the total of the fraction of all possible pairs of shortest paths that go through a node.

Getting ready

Install NetworkX with instructions from the *Introduction* section.

How to do it...

The script is in the `between_centrality.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import networkx as nx
import dautil as dl
import pandas as pd
```

2. Load the Facebook SPAN data into a NetworkX graph:

```
fb_file = dl.data.SPANFB().load()
G = nx.read_edgelist(fb_file,
                     create_using=nx.Graph(),
                     nodetype=int)
```

3. Calculate the betweenness centrality with $k = 256$ (number of nodes to use) and store the result in a pandas DataFrame object:

```
key_values = nx.betweenness_centrality(G, k=256)
df = pd.DataFrame.from_dict(key_values, orient='index')

dl.options.set_pd_options()
print('Betweenness Centrality', df)
```

Refer to the following screenshot for the end result:

```
Betweenness Centrality
0      1.406e-01
1      4.996e-06
...
4037  0.000e+00
4038  0.000e+00

[4039 rows x 1 columns]
```

See also

- ▶ The Wikipedia page about betweenness centrality at https://en.wikipedia.org/wiki/Betweenness_centrality (retrieved October 2015)
- ▶ The documentation for the `betweenness_centrality()` function at https://networkx.github.io/documentation/latest/reference/generated/networkx.algorithms.centrality.betweenness_centrality.html (retrieved October 2015)

Estimating the average clustering coefficient

From kindergarten onward, we have friends, close friends, best friends forever, social media friends, and other friends. A social network graph should, therefore, have clumps, unlike what you would observe at a high school party. The question that naturally arises is what would happen if we just invite a group of random strangers to a party or recreate this setup online? We would expect the probability of strangers connecting to be lower than for friends. In graph theory, this probability is measured by the **clustering coefficient**.

The **average clustering coefficient** is a local (single node) version of the clustering coefficient. The definition of this metric considers triangles formed by nodes. With three nodes, we can form one triangle, for instance, the three musketeers. If we add D'Artagnan to the mix, more triangles are possible, but not all the triangles have to be realized. It could happen that D'Artagnan gets in a fight with all three of the musketeers. In (8.5), we define a clustering coefficient as the ratio of realized and possible triangles and average clustering coefficient (8.6):

$$(8.5) \quad C_i = \frac{\lambda_G(v)}{\tau_G(v)}$$

$$(8.6) \quad \bar{C} = \frac{1}{n} \sum_{i=1}^n C_i$$

Getting ready

Install NetworkX with the instructions from the *Introduction* section.

How to do it...

The script is in the `avg_clustering.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import networkx as nx
import dautil as dl
```

2. Load the Facebook SPAN data into a NetworkX graph:

```
fb_file = dl.data.SPANFB().load()
G = nx.read_edgelist(fb_file,
                     create_using=nx.Graph(),
                     nodetype=int)
```

3. Calculate the average clustering coefficient as follows:

```
print('Average Clustering',
      nx.average_clustering(G))
```

We get the following result for the Facebook SPAN data:

```
Average Clustering 0.6055467186200871
```

See also

- ▶ The Wikipedia page about the clustering coefficient at https://en.wikipedia.org/wiki/Clustering_coefficient (retrieved October 2015)
- ▶ The documentation for the `average_clustering()` function at https://networkx.github.io/documentation/latest/reference/generated/networkx.algorithms.approximation.clustering_coefficient.average_clustering.html (retrieved October 2015).

Calculating the assortativity coefficient of a graph

In graph theory, similarity is measured by the **degree distribution**. **Degree** is the number of connections a node has to other nodes. In a directed graph, we have incoming and outgoing connections and corresponding indegree and outdegree. Friends tend to have something in common. In graph theory, this tendency is measured by the **assortativity coefficient**. This coefficient is the Pearson correlation coefficient between a pair of nodes, as given in the following equation:

$$(8.7) \quad r = \frac{\sum_{jk} jk (e_{jk} - q_j q_k)}{\sigma_q^2}$$

q_k (distribution of the remaining degree) is the number of connections leaving node k . e_{jk} is the joint probability distribution of the remaining degrees of the node pair.

Getting ready

Install NetworkX with the instructions from the *Introduction* section.

How to do it...

The code is in the `assortativity.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import networkx as nx
import dautil as dl
```

2. Load the Facebook SPAN data into a NetworkX graph:

```
fb_file = dl.data.SPANFB().load()
G = nx.read_edgelist(fb_file,
                     create_using=nx.Graph(),
                     nodetype=int)
```

3. Calculate the assortativity coefficient as follows:

```
print('Degree Assortativity Coefficient',
      nx.degree_assortativity_coefficient(G))
```

We get the following result for the Facebook SPAN data:

```
Degree Assortativity Coefficient 0.0635772291856
```

See also

- ▶ The Wikipedia page about degree distribution at https://en.wikipedia.org/wiki/Degree_distribution (retrieved October 2015)
- ▶ The Wikipedia page about assortativity at <https://en.wikipedia.org/wiki/Assortativity> (retrieved October 2015)
- ▶ The documentation for the `degree_assortativity_coefficient()` function at https://networkx.github.io/documentation/latest/reference/generated/networkx.algorithms.assortativity.degree_assortativity_coefficient.html (retrieved October 2015)

Getting the clique number of a graph

A **complete graph** is a graph in which every pair of nodes is connected by a unique connection. A **clique** is a subgraph that is complete. This is equivalent to the general concept of cliques in which every person knows all the other people. The **maximum clique** is the clique with the most nodes. The **clique number** is the number of nodes in the maximum clique. Unfortunately finding the clique number takes a long time, so we will not use the complete Facebook SPAN data.

Getting ready

Install NetworkX with the instructions from the *Introduction* section.

How to do it...

The code is in the `clique_number.py` file in this book's code bundle:

1. The imports are as follows:

```
import networkx as nx
import dautil as dl
```

2. Load the Facebook SPAN data into a NetworkX graph:

```
fb_file = dl.data.SPANFB().load()
G = nx.read_edgelist(fb_file,
                     create_using=nx.Graph(),
                     nodetype=int)
```

3. Determine the clique number for a subgraph:

```
print('Graph Clique Number',
      nx.graph_clique_number(G.subgraph(list(range(2048)))))
```

We get the following result for the partial Facebook SPAN data:

Graph Clique Number 38

See also

- ▶ The Wikipedia page about complete graphs at https://en.wikipedia.org/wiki/Complete_graph (retrieved October 2015)
- ▶ The Wikipedia page about cliques at https://en.wikipedia.org/wiki/Clique_%28graph_theory%29 (retrieved October 2015)
- ▶ The documentation for the `graph_clique_number()` function at https://networkx.github.io/documentation/latest/reference/generated/networkx.algorithms.clique.graph_clique_number.html (retrieved October 2015)

Creating a document graph with cosine similarity

The Internet is a large web of documents linked to each other. We can view it as a document graph in which each node corresponds to a document. You will expect documents to link to similar documents; however, web pages sometimes link to other unrelated web pages. This can be by mistake or on purpose, for instance in the context of advertising or attempts to improve search engine rankings. A more trustworthy source such as Wikipedia will probably yield a better graph. However, some Wikipedia pages are very basic stubs, so we may be missing out on quality links.

The **cosine similarity** is a common distance metric to measure the similarity of two documents. For this metric, we need to compute the inner product of two feature vectors. The cosine similarity of vectors corresponds to the cosine of the angle between vectors, hence the name. The cosine similarity is given by the following equation:

$$(8.8) \quad k(x, y) = \frac{xy^T}{\|x\|\|y\|}$$

The feature vectors in this recipe are the TF-IDF scores, corresponding to a document. The cosine similarity of a document with itself is equal to 1 (zero angle); therefore for documents to be similar, the cosine similarity should be as close to 1 as possible.

We will perform the following steps to create a document graph of the news articles in the Brown corpus:

1. Calculate cosine similarities using the TF-IDF scores that we stored in a pickle from the code of the *Stemming, lemmatizing, filtering, and TF-IDF scores* recipe. The result is similar to a correlation matrix.
2. For each document, add a connection in the graph to each document, which is similar enough. I used the 90th percentile of similarities as threshold; however, you can use another value if you prefer.
3. For each document, select the top three words using the TF-IDF scores. I used the words to annotate the document nodes.
4. Calculate graph metrics with NetworkX, as discussed in this chapter.

How to do it...

The code to create the document graph with cosine similarity is in the `cos_similarity.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
from sklearn.metrics.pairwise import cosine_similarity
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import dutil as dl
import ch8util
```

2. Define the following function to add nodes to the NetworkX graph annotated with the top three most important words per document:

```
def add_nodes(G, nodes, start, terms):
    for n in nodes:
        words = top_3_words(tfidf, n, terms)
        G.add_node(n, words='{}0}: {}1}'.format(n, " ".join(words.tolist())))
        G.add_edge(start, n)
```

3. Define the following function to find the top three words for a document:

```
def top_3_words(tfidf, row, terms):
    indices = np.argsort(tfidf[row].toarray().ravel())[-3:]

    return terms[indices]
```

4. Load the necessary data, calculate cosine similarities, and create a NetworkX graph:

```
tfidf = ch8util.load_tfidf()
terms = ch8util.load_terms()

sims = cosine_similarity(tfidf, tfidf)
G = nx.Graph()
```

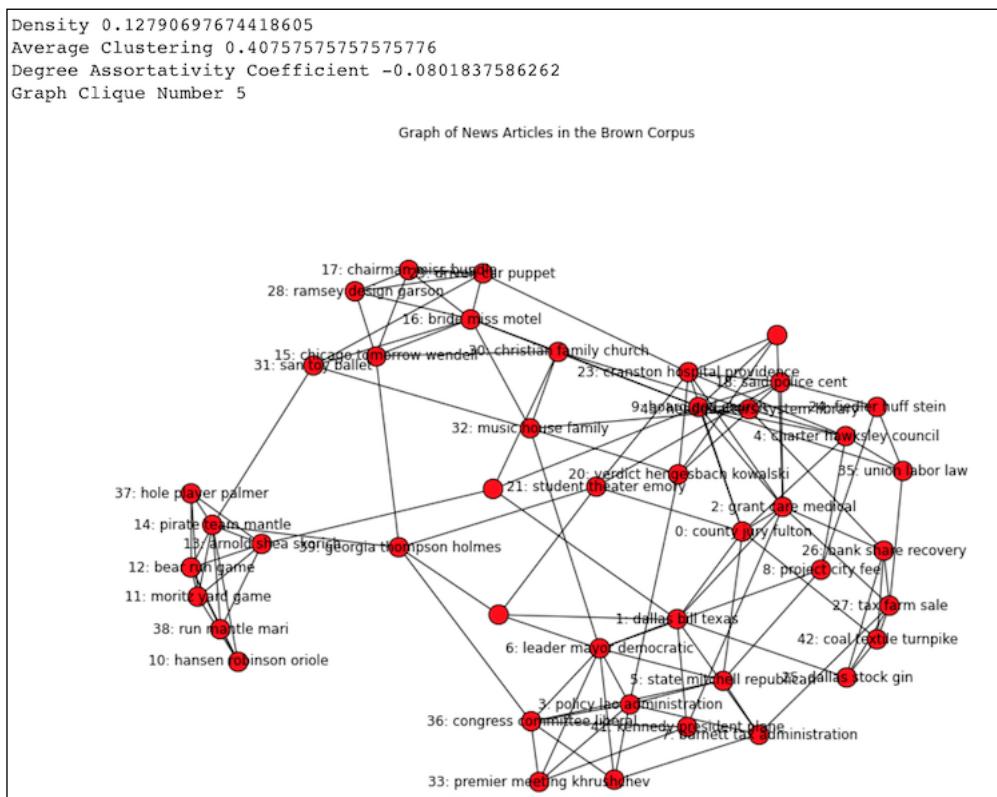
5. Iterate through the cosine similarities and add nodes to the graph:

```
for i, row in enumerate(sims):
    over_limit = np.where(row > np.percentile(row, 90))[0]
    nodes = set(over_limit.tolist())
    nodes.remove(i)
    add_nodes(G, nodes, i, terms)
```

6. Plot the graph and print some metrics using NetworkX:

```
labels = nx.get_node_attributes(G, 'words')
nx.draw_networkx(G, pos=nx.spring_layout(G), labels=labels)
plt.axis('off')
plt.title('Graph of News Articles in the Brown Corpus')
print('Density', nx.density(G))
print('Average Clustering',
      nx.average_clustering(G))
print('Degree Assortativity Coefficient',
      nx.degree_assortativity_coefficient(G))
print('Graph Clique Number', nx.graph_clique_number(G))
```

Refer to the following screenshot for the end result:



See also

- ▶ The *Computing social network density* recipe
- ▶ The *Estimating average clustering coefficient* recipe
- ▶ The *Calculating the assortativity coefficient of a graph* recipe
- ▶ The *Getting the clique number of a graph* recipe
- ▶ The Wikipedia page about the cosine similarity at https://en.wikipedia.org/wiki/Cosine_similarity (retrieved October 2015)
- ▶ The documentation about the `cosine_similarity()` function at <http://scikit-learn.org/stable/modules/metrics.html#cosine-similarity> (retrieved October 2015)

9

Ensemble Learning and Dimensionality Reduction

In this chapter, we will cover the following recipes:

- ▶ Recursively eliminating features
- ▶ Applying principal component analysis for dimensionality reduction
- ▶ Applying linear discriminant analysis for dimensionality reduction
- ▶ Stacking and majority voting for multiple models
- ▶ Learning with random forests
- ▶ Fitting noisy data with the RANSAC algorithm
- ▶ Bagging to improve results
- ▶ Boosting for better learning
- ▶ Nesting cross-validation
- ▶ Reusing models with joblib
- ▶ Hierarchically clustering data
- ▶ Taking a Theano tour

Introduction

In the 1983 *War Games* movie, a computer made life and death decisions that could have resulted in World War III. As far as I know, technology wasn't able to pull off such feats at the time. However, in 1997, the Deep Blue supercomputer did manage to beat a world chess champion. In 2005, a Stanford self-driving car drove by itself for more than 130 kilometers in a desert. In 2007, the car of another team drove through regular traffic for more than 50 kilometers. In 2011, the Watson computer won a quiz against human opponents. If we assume that computer hardware is the limiting factor, then we can try to extrapolate into the future. Ray Kurzweil did just that, and according to him, we can expect human-level intelligence around 2029.

In this chapter, we will focus on the simpler problem of forecasting weather for the next day. We will assume that the weather today depends on yesterday's weather. Theoretically, if a butterfly flaps its wings at one location, this could trigger a chain of events causing a snow storm in a place thousands kilometers further away (the butterfly effect). This is not impossible, but very improbable. However, if we have many such incidents, a similar scenario will occur more often than you would suspect.

It is impossible to take into account all possible factors. In fact, we will try to make our life easier by ignoring some of the data we have available. We will apply classification and regression algorithms, as well as hierarchical clustering. Let's defer results evaluation to *Chapter 10, Evaluating Classifiers, Regressors, and Clusters*. If you are curious about the confusion matrix mentioned in the classification recipes, please jump to the *Getting classification straight with the confusion matrix* recipe.

Most artificial intelligence systems are nowadays, in fact, not so smart. A judge in a court of law could make wrong decisions because he or she is biased or having a bad day. A group of multiple judges should perform better. This is comparable to a machine learning project, in which we worry about overfitting and underfitting. **Ensemble learning** is a solution to this conundrum, and it basically means combining multiple learners in a clever way.

A major part of this chapter is about **hyperparameter** optimization—these are parameters of classifiers and regressors. To check for overfitting or underfitting, we can use **learning curves**, which show training and test scores for varying training set sizes. We can also vary the value of a single hyperparameter with **validation curves**.

Recursively eliminating features

If we have many features (explanatory variables), it is tempting to include them all in our model. However, we then run the risk of overfitting—getting a model that works very well for the training data and very badly for unseen data. Not only that, but the model is bound to be relatively slow and require a lot of memory. We have to weigh accuracy (or an other metric) against speed and memory requirements.

We can try to ignore features or create new better compound features. For instance, in online advertising, it is common to work with ratios, such as the ratio of views and clicks related to an ad. Common sense or domain knowledge can help us select features. In the worst-case scenario, we may have to rely on correlations or other statistical methods. The scikit-learn library offers the RFE class (recursive feature elimination), which can automatically select features. We will use this class in this recipe. We also need an external estimator. The RFE class is relatively new, and unfortunately there is no guarantee that all estimators will work together with the RFE class.

How to do it...

1. The imports are as follows:

```
from sklearn.feature_selection import RFE
from sklearn.svm import SVC
from sklearn.svm import SVR
from sklearn.preprocessing import MinMaxScaler
import dautil as dl
import warnings
import numpy as np
```

2. Create a SVC classifier and an RFE object as follows:

```
warnings.filterwarnings("ignore", category=DeprecationWarning)
clf = SVC(random_state=42, kernel='linear')
selector = RFE(clf)
```

3. Load the data, scale it using a MinMaxScaler function, and add the day of the year as a feature:

```
df = dl.data.Weather.load().dropna()
df['RAIN'] = df['RAIN'] == 0
df['DOY'] = [float(d.dayofyear) for d in df.index]
scaler = MinMaxScaler()

for c in df.columns:
    if c != 'RAIN':
        df[c] = scaler.fit_transform(df[c])
```

4. Print the first row of the data as a sanity check:

```
dl.options.set_pd_options()
print(df.head(1))
X = df[:-1].values
np.set_printoptions(formatter={'all': '{:.3f}'.format()})
print(X[0])
np.set_printoptions()
```

5. Determine support and rankings for the features using rain or no rain as classes (in the context of classification):

```
y = df['RAIN'][1:].values
selector = selector.fit(X, y)
print('Rain support', df.columns[selector.support_])
print('Rain rankings', selector.ranking_)
```

6. Determine support and rankings for the features using temperature as a feature:

```
reg = SVR(kernel='linear')
selector = RFE(reg)
y = df['TEMP'][1:].values
selector = selector.fit(X, y)
print('Temperature support', df.columns[selector.support_])
print('Temperature ranking', selector.ranking_)
```

Refer to the following screenshot for the end result:

	WIND_DIR	WIND_SPEED	TEMP	RAIN	PRESSURE	DOY
YYYYMMDD						
1906-01-01	0.311	0.594	0.269	True	0.712	0
	[0.311 0.594 0.269 1.000 0.712 0.000]					
Rain support	Index(['WIND_DIR', 'RAIN', 'PRESSURE'], dtype='object')					
Rain rankings	[1 2 4 1 3]					
Temperature support	Index(['WIND_SPEED', 'TEMP', 'PRESSURE'], dtype='object')					
Temperature ranking	[4 1 1 3 1 2]					

The code for this recipe is in the `feature_elimination.py` file in this book's code bundle.

How it works

The RFE class selects half of the features by default. The algorithm is as follows:

1. Train the external estimator on the data and assign weights to the features.
2. The features with smallest weights are removed.
3. Repeat the procedure until we have the necessary number of features.

See also

- The documentation for the RFE class at http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html (retrieved November 2015)

Applying principal component analysis for dimension reduction

Principal component analysis (PCA), invented by Karl Pearson in 1901, is an algorithm that transforms data into uncorrelated orthogonal features called **principal components**. The principal components are the eigenvectors of the covariance matrix.

Sometimes, we get better results by scaling the data prior to applying PCA, although this is not strictly necessary. We can interpret PCA as projecting data to a lower dimensional space. Some of the principal components contribute relatively little information (low variance); therefore, we can omit them. We have the following transformation:

$$(9.1) \quad T_L = XW_L$$

The result is the matrix T_L , with the same number of rows as the original matrix but a lower number of columns.

Dimensionality reduction is, of course, useful for visualization and modeling and to reduce the chance of overfitting. In fact, there is a technique called **Principal component regression (PCR)**, which uses this principle. In a nutshell, PCR performs the following steps:

1. Transforms the data to a lower dimensional space with PCA.
2. Performs linear regression in the new space.
3. Transforms the result back to the original coordinate system.

How to do it...

1. The imports are as follows:

```
import dautil as dl
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
from sklearn.preprocessing import scale
```

2. Load the data as follows and group by the day of the year:

```
df = dl.data.Weather.load().dropna()
df = df.groupby_yday(df).mean()
X = df.values
```

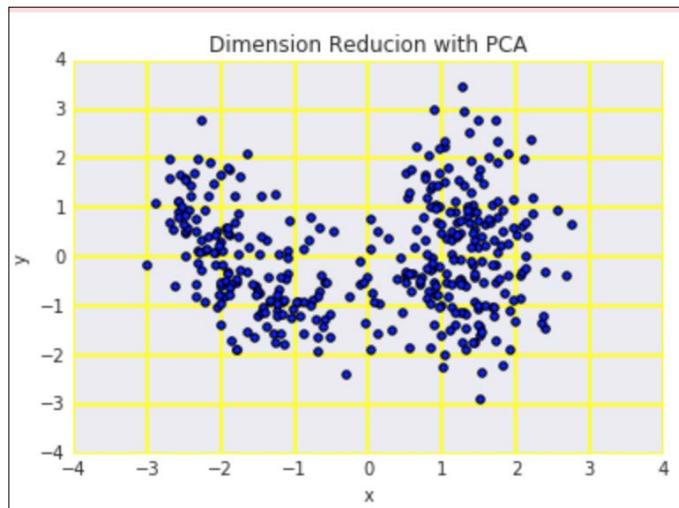
3. Apply PCA to project the data into a two-dimensional space:

```
pca = PCA(n_components=2)
X_r = pca.fit_transform(scale(X)).T
```

-
4. Plot the result of the transformation:

```
plt.scatter(X_r[0], X_r[1])
plt.xlabel('x')
plt.ylabel('y')
plt.title('Dimension Reducion with PCA')
```

Refer to the following screenshot for the end result:



The code is in the `applying_pca.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about PCA at https://en.wikipedia.org/wiki/Principal_component_analysis (retrieved November 2015)
- ▶ The Wikipedia page about PCR at https://en.wikipedia.org/wiki/Principal_component_regression (retrieved November 2015)
- ▶ The documentation for the PCA class at <http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html> (retrieved November 2015)

Applying linear discriminant analysis for dimension reduction

Linear discriminant analysis (LDA) is an algorithm that looks for a linear combination of features in order to distinguish between classes. It can be used for classification or dimensionality reduction by projecting to a lower dimensional subspace. LDA requires a target attribute both for classification and dimensionality reduction.

If we represent class densities as multivariate Gaussians, then LDA assumes that the classes have the same covariance matrix. We can use training data to estimate the parameters of the class distributions.

In scikit-learn, `lda.LDA` has been deprecated in 0.17 and renamed `discriminant_analysis.LinearDiscriminantAnalysis`. The default solver of this class uses singular value decomposition, does not need to calculate the covariance matrix, and is therefore fast.

How to do it...

The code is in the `applying_lda.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import dutil as dl
from sklearn.discriminant_analysis import
LinearDiscriminantAnalysis
import matplotlib.pyplot as plt
```

2. Load the data as follows:

```
df = dl.data.Weather.load().dropna()
X = df.values
y = df['WIND_DIR'].values
```

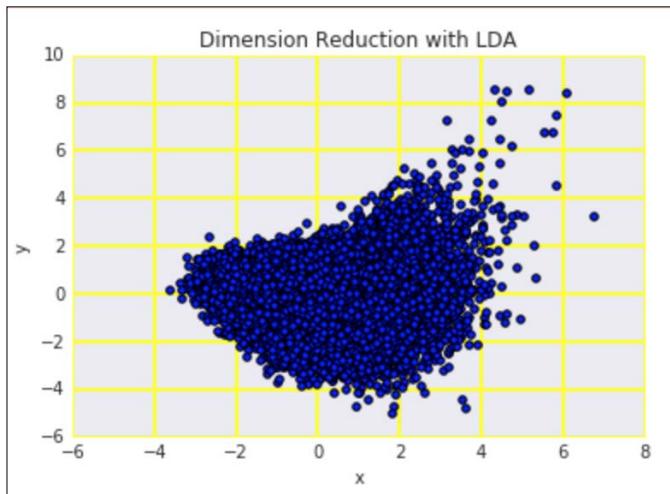
3. Apply LDA to project the data into a two-dimensional space:

```
lda = LinearDiscriminantAnalysis(n_components=2)
X_r = lda.fit(X, y).transform(X).T
```

4. Plot the result of the transformation:

```
plt.scatter(X_r[0], X_r[1])
plt.xlabel('x')
plt.ylabel('y')
plt.title('Dimension Reduction with LDA')
```

Refer to the following screenshot for the end result:



See also

- ▶ The Wikipedia page about LDA at https://en.wikipedia.org/wiki/Linear_discriminant_analysis (retrieved November 2015)
- ▶ The relevant scikit-learn documentation is http://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.LinearDiscriminantAnalysis.html (retrieved November 2015)

Stacking and majority voting for multiple models

It is generally believed that two people know more than one person alone. A democracy should work better than a dictatorship. In machine learning, we don't have humans making decisions, but algorithms. When we have multiple classifiers or regressors working together, we speak of **ensemble learning**.

There are many ensemble learning schemes. The simplest setup does majority voting for classification and averaging for regression. In scikit-learn 0.17, you can use the `VotingClassifier` class to do majority voting. This classifier lets you emphasize or suppress classifiers with weights.

Stacking takes the outputs of machine learning estimators and then uses those as inputs for another algorithm. You can, of course, feed the output of the higher-level algorithm to another predictor. It is possible to use any arbitrary topology, but for practical reasons, you should try a simple setup first.

How to do it...

1. The imports are as follows:

```
import dautil as dl
from sklearn.tree import DecisionTreeClassifier
import numpy as np
import ch9util
from sklearn.ensemble import VotingClassifier
from sklearn.grid_search import GridSearchCV
from IPython.display import HTML
```

2. Load the data and create three decision tree classifiers:

```
X_train, X_test, y_train, y_test = ch9util.rain_split()
default = DecisionTreeClassifier(random_state=53, min_samples_
leaf=3,
                                max_depth=4)
entropy = DecisionTreeClassifier(criterion='entropy',
                                  min_samples_leaf=3, max_depth=4,
                                  random_state=57)
random = DecisionTreeClassifier(splitter='random', min_samples_
leaf=3,
                                max_depth=4, random_state=5)
```

3. Use the classifiers to take a vote:

```
clf = VotingClassifier([('default', default),
                      ('entropy', entropy), ('random', random)])
params = {'voting': ['soft', 'hard'],
          'weights': [None, (2, 1, 1), (1, 2, 1), (1, 1, 2)]}
gscv = GridSearchCV(clf, param_grid=params, n_jobs=-1, cv=5)
gscv.fit(X_train, y_train)
votes = gscv.predict(X_test)

preds = []
```

```
for clf in [default, entropy, random] :  
    clf.fit(X_train, y_train)  
    preds.append(clf.predict(X_test))  
  
preds = np.array(preds)
```

4. Plot the confusion matrix for the votes-based forecast:

```
%matplotlib inline  
context = dl.nb.Context('stacking_multiple')  
dl.nb.RcWidget(context)  
  
sp = dl.plotting.Subplotter(2, 2, context)  
html = ch9util.report_rain(votes, y_test, gscv.best_params_,  
    sp.ax)  
sp.ax.set_title(sp.ax.get_title() + ' | Voting')
```

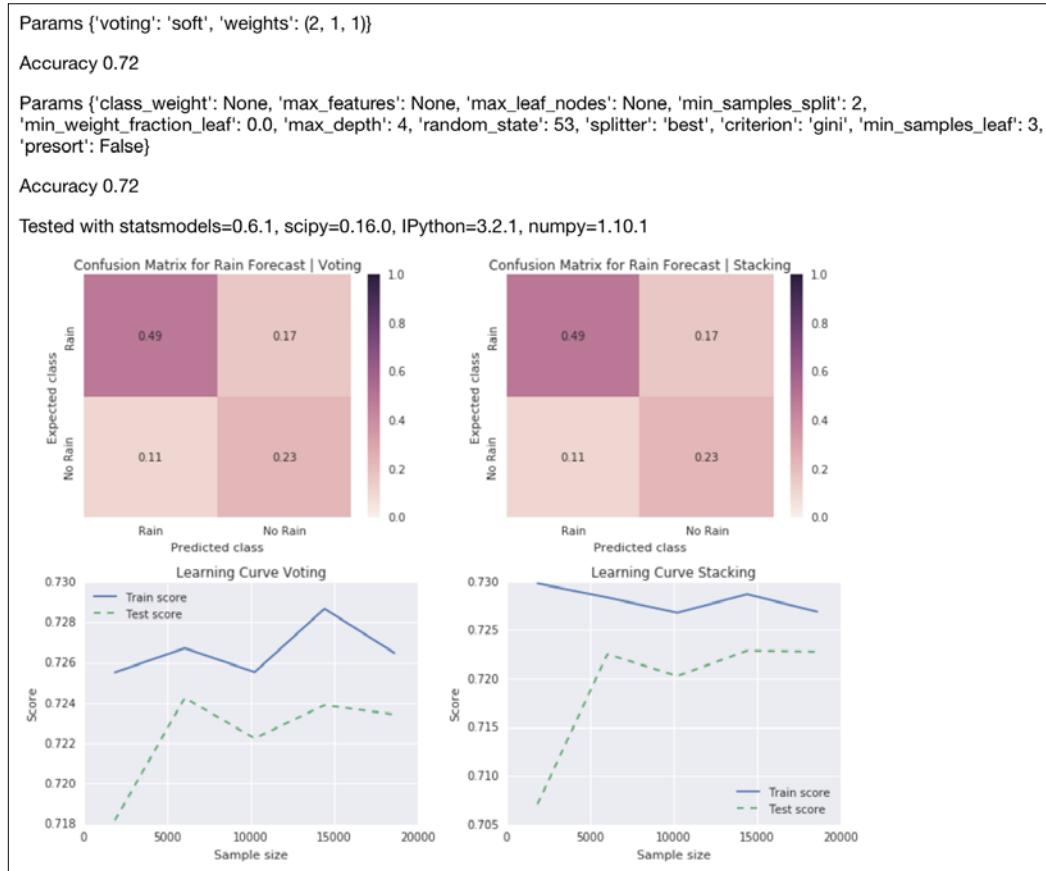
5. Plot the confusion matrix for the stacking-based forecast:

```
default.fit(preds_train.T, y_train)  
stacked_preds = default.predict(preds.T)  
html += ch9util.report_rain(stacked_preds,  
    y_test, default.get_params(), sp.next_  
    ax())  
sp.ax.set_title(sp.ax.get_title() + ' | Stacking')  
ch9util.report_rain(default.predict(preds.T), y_test)
```

6. Plot the learning curves of the voting and stacking classifiers:

```
ch9util.plot_learn_curve(sp.next_ax(), gscv.best_estimator_, X_  
    train,  
    y_train, title='Voting')  
  
ch9util.plot_learn_curve(sp.next_ax(), default, X_train,  
    y_train, title='Stacking')
```

Refer to the following screenshot for the end result:



The code is in the `stacking_multiple.ipynb` file in this book's code bundle.

See also

- ▶ The documentation for the `VotingClassifier` class at <http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html> (retrieved November 2015)
- ▶ The Wikipedia section about stacking at https://en.wikipedia.org/wiki/Ensemble_learning#Stacking (retrieved November 2015)

Learning with random forests

The `if a: else b` statement is one of the most common statements in Python programming. By nesting and combining such statements, we can build a so-called decision tree. This is similar to an old fashioned flowchart, although flowcharts also allow loops. The application of decision trees in machine learning is called **decision tree learning**. The end nodes of the trees in decision tree learning, also known as **leaves**, contain the class labels of a classification problem. Each non-leaf node is associated with a Boolean condition involving feature values.

Decision trees can be used to deduce relatively simple rules. Being able to produce such results is, of course, a huge advantage. However, you have to wonder how good these rules are. If we add new data, would we get the same rules?

If one decision tree is good, a whole forest should be even better. Multiple trees should reduce the chance of overfitting. However, as in a real forest, we don't want only one type of tree. Obviously, we would have to average or decide by majority voting what the appropriate result should be.

In this recipe, we will apply the **random forest** algorithm invented by Leo Breiman and Adele Cutler. The "random" in the name refers to randomly selecting features from the data. We use all the data but not in the same decision tree.

Random forests also apply **bagging (bootstrap aggregating)**, which we will discuss in the *Bagging to improve results* recipe. The bagging of decision trees consists of the following steps:

1. Sample training examples with replacement and assign them to a tree.
2. Train the trees on their assigned data.

We can determine the correct number of trees by cross-validation or by plotting the test and train error against the number of trees.

How to do it...

The code is in the `random_forest.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import dutil as dl
from sklearn.grid_search import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
import ch9util
import numpy as np
from IPython.display import HTML
```

-
2. Load the data and do a prediction as follows:

```
X_train, X_test, y_train, y_test = ch9util.rain_split()
clf = RandomForestClassifier(random_state=44)
params = {
    'max_depth': [2, 4],
    'min_samples_leaf': [1, 3],
    'criterion': ['gini', 'entropy'],
    'n_estimators': [100, 200]
}

rfc = GridSearchCV(estimator=RandomForestClassifier(),
                    param_grid=params, cv=5, n_jobs=-1)
rfc.fit(X_train, y_train)
preds = rfc.predict(X_test)
```

3. Plot the rain forecast confusion matrix as follows:

```
sp = dl.plotting.Subplotter(2, 2, context)
html = ch9util.report_rain(preds, y_test, rfc.best_params_, sp.ax)
```

4. Plot a validation curve for a range of forest sizes:

```
ntrees = 2 ** np.arange(9)
ch9util.plot_validation(sp.next_ax(), rfc.best_estimator_,
                        X_train, y_train, 'n_estimators', ntrees)
```

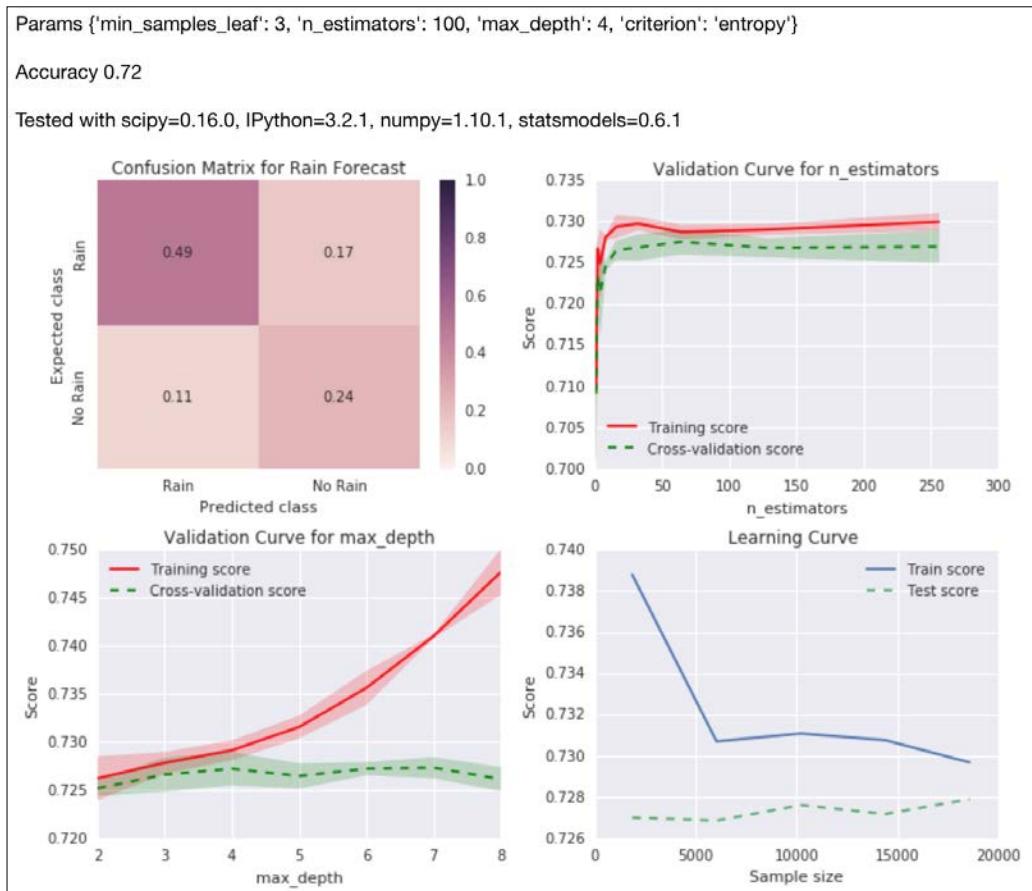
5. Plot a validation curve for a range of depths:

```
depths = np.arange(2, 9)
ch9util.plot_validation(sp.next_ax(), rfc.best_estimator_,
                        X_train, y_train, 'max_depth', depths)
```

6. Plot the learning curve of the best estimator:

```
ch9util.plot_learn_curve(sp.next_ax(),
                         rfc.best_estimator_, X_train, y_train)
HTML(html + sp.exit())
```

Refer to the following screenshot for the end result:



There's more...

Random forests classification is considered such a versatile algorithm that we can use it for almost any classification task. **Genetic algorithms** and **genetic programming** can do a grid search or optimization in general.

We can consider a program to be a sequence of operators and operands that generates a result. This is a very simplified model of programming, of course. However, in such a model, it is possible to evolve programs using natural selection modeled after biological theories. A genetic program is self-modifying with huge adaptability, but we get a lower level of determinism.

The TPOT project is an attempt to evolve machine learning pipelines (currently uses a small number of classifiers including random forests). I forked TPOT 0.1.3 on GitHub and made some changes. TPOT uses `deap` for the genetic programming parts, which you can install as follows:

```
$ pip install deap
```

I tested the code with `deap` 1.0.2. Install my changes under tag `r1` as follows:

```
$ git clone git@github.com:ivanidris/tpot.git
$ cd tpot
$ git checkout r1
$ python setup.py install
```

You can also get the code from <https://github.com/ivanidris/tpot/releases/tag/r1>. The following code from the `rain_pot.py` file in this book's code bundle demonstrates how to fit and score rain predictions with TPOT:

```
import ch9util
from tpot import TPOT

X_train, X_test, y_train, y_test = ch9util.rain_split()
tpot = TPOT(generations=7, population_size=110, verbosity=2)
tpot.fit(X_train, y_train)
print(tpot.score(X_train, y_train, X_test, y_test))
```

See also

- ▶ The Wikipedia page about random forests at https://en.wikipedia.org/wiki/Random_forest (retrieved November 2015)
- ▶ The documentation for the `RandomForestClassifier` class at <http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html> (retrieved November 2015)

Fitting noisy data with the RANSAC algorithm

We discussed the issue of outliers in the context of regression elsewhere in this book (refer to the *See also* section at the end of this recipe). The issue is clear—the outliers make it difficult to properly fit our models. The **RANdom SAmple Consensus algorithm (RANSAC)** does a best effort attempt to fit our data in an iterative manner. RANSAC was introduced by Fishler and Bolles in 1981.

We often have some knowledge about our data, for instance the data may follow a normal distribution. Or, the data may be a mix produced by multiple processes with different characteristics. We could also have abnormal data due to glitches or errors in data transformation. In such cases, it should be easy to identify outliers and deal with them appropriately. The RANSAC algorithm doesn't know your data, but it also assumes that there are inliers and outliers.

The algorithm goes through a fixed number of iterations. The object is to find a set of inliers of specified size (**consensus set**).

RANSAC performs the following steps:

1. Randomly select as small a subset of the data as possible and fit the model.
2. Check whether each data point is consistent with the fitted model in the previous step. Mark inconsistent points as outliers using a residuals threshold.
3. Accept the model if enough inliers have been found.
4. Re-estimate parameters with the full consensus set.

The scikit-learn RANSACRegressor class can use a suitable estimator for fitting. We will use the default LinearRegression estimator. We can also specify the minimum number of samples for fitting, the residuals threshold, a decision function for outliers, a function that decides whether a model is valid, the maximum number of iterations, and the required number of inliers in the consensus set.

How to do it...

The code is in the `fit_ransac.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import ch9util
from sklearn import linear_model
from sklearn.grid_search import GridSearchCV
import numpy as np
import dutil as dl
from IPython.display import HTML
```

-
2. Load the data and do a temperature prediction as follows:

```
X_train, X_test, y_train, y_test = ch9util.temp_split()  
ransac = linear_model.RANSACRegressor(random_state=27)  
params = {  
    'max_trials': [50, 100, 200],  
    'stop_probability': [0.98, 0.99]  
}  
  
gscv = GridSearchCV(estimator=ransac, param_grid=params, cv=5)  
gscv.fit(X_train, y_train)  
preds = gscv.predict(X_test)
```

3. Scatter plot the predictions against the actual values:

```
sp = dl.plotting.Subplotter(2, 2, context)  
html = ch9util.scatter_predictions(preds, y_test, gscv.best_  
params_,  
                                    gscv.best_score_, sp.ax)
```

4. Plot a validation curve for a range of trial numbers:

```
trials = 10 * np.arange(5, 20)  
ch9util.plot_validation(sp.next_ax(), gscv.best_estimator_,  
                        X_train, y_train, 'max_trials', trials)
```

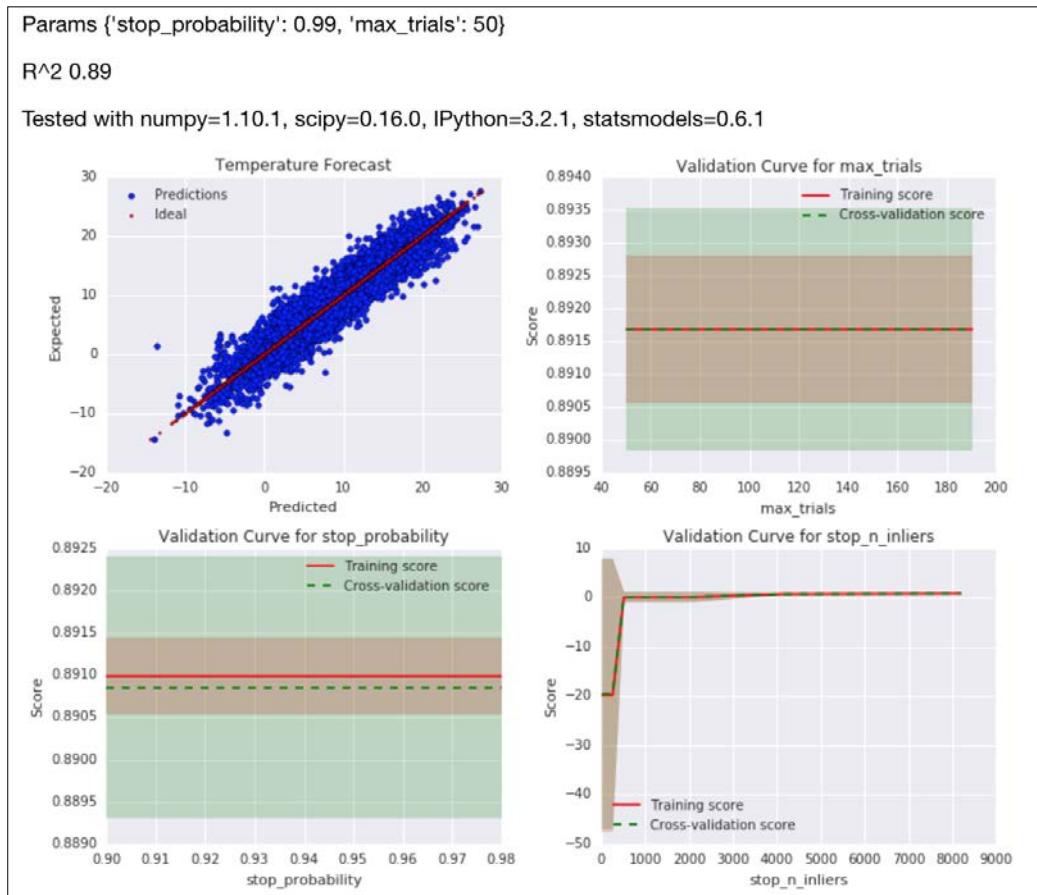
5. Plot a validation curve for a range of stop probabilities:

```
probs = 0.01 * np.arange(90, 99)  
ch9util.plot_validation(sp.next_ax(), gscv.best_estimator_,  
                        X_train, y_train, 'stop_probability',  
                        probs)
```

6. Plot a validation curve for a range of consensus set sizes:

```
ninliers = 2 ** np.arange(4, 14)  
ch9util.plot_validation(sp.next_ax(), gscv.best_estimator_,  
                        X_train, y_train, 'stop_n_inliers',  
                        ninliers)  
HTML(html + sp.exit())
```

Refer to the following screenshot for the end result:



See also

- ▶ The Wikipedia page about the RANSAC algorithm at <https://en.wikipedia.org/wiki/RANSAC> (retrieved November 2015)
- ▶ The relevant scikit-learn documentation at http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.RANSACRegressor.html (retrieved November 2015)
- ▶ The *Fitting a robust linear model* recipe
- ▶ The *Taking variance into account with weighted least squares* recipe

Bagging to improve results

Bootstrap aggregating or **bagging** is an algorithm introduced by Leo Breiman in 1994, which applies bootstrapping to machine learning problems. Bagging was also mentioned in the *Learning with random forests* recipe.

The algorithm aims to reduce the chance of overfitting with the following steps:

1. We generate new training sets from input training data by sampling with replacement.
2. Fit models to each generated training set.
3. Combine the results of the models by averaging or majority voting.

The scikit-learn `BaggingClassifier` class allows us to bootstrap training examples, and we can also bootstrap features as in the random forests algorithm. When we perform a grid search, we refer to hyperparameters of the base estimator with the prefix `base_estimator_`. We will use a decision tree as the base estimator so that we can reuse some of the hyperparameter configuration from the *Learning with random forests* recipe.

How to do it...

The code is in the `bagging.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import ch9util
from sklearn.ensemble import BaggingClassifier
from sklearn.grid_search import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
import numpy as np
import dutil as dl
from IPython.display import HTML
```

2. Load the data and create a `BaggingClassifier`:

```
X_train, X_test, y_train, y_test = ch9util.rain_split()
clf = BaggingClassifier(base_estimator=DecisionTreeClassifier(
    min_samples_leaf=3, max_depth=4), random_state=43)
```

-
3. Grid search, fit, and predict as follows:

```
params = {
    'n_estimators': [320, 640],
    'bootstrap_features': [True, False],
    'base_estimator_criterion': ['gini', 'entropy']
}

gscv = GridSearchCV(estimator=clf, param_grid=params,
                     cv=5, n_jobs=-1)

gscv.fit(X_train, y_train)
preds = gscv.predict(X_test)
```

4. Plot the rain forecast confusion matrix as follows:

```
sp = dl.plotting.Subplotter(2, 2, context)
html = ch9util.report_rain(preds, y_test, gscv.best_params_,
                           sp.ax)
```

5. Plot a validation curve for a range of ensemble sizes:

```
ntrees = 2 ** np.arange(4, 11)
ch9util.plot_validation(sp.next_ax(), gscv.best_estimator_,
                        X_train, y_train, 'n_estimators', ntrees)
```

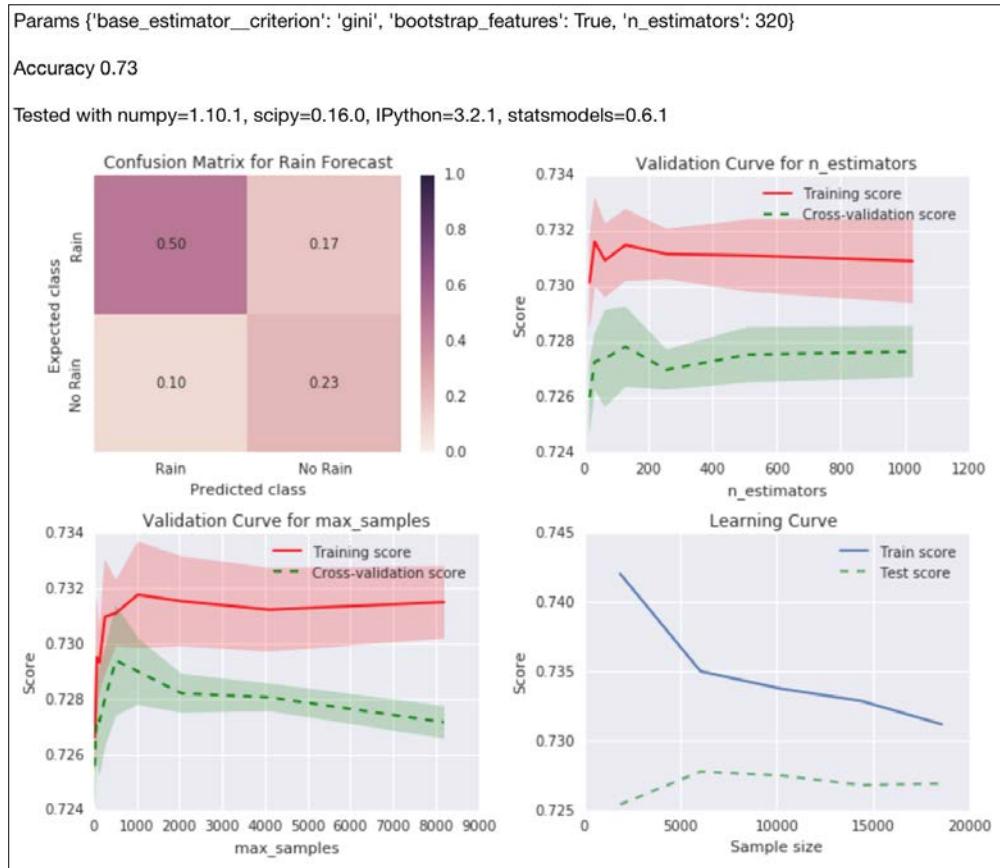
6. Plot a validation curve for the max_samples parameter:

```
nsamples = 2 ** np.arange(4, 14)
ch9util.plot_validation(sp.next_ax(), gscv.best_estimator_,
                        X_train, y_train, 'max_samples', nsamples)
```

7. Plot the learning curve as follows:

```
ch9util.plot_learn_curve(sp.next_ax(), gscv.best_estimator_,
                         X_train, y_train)
HTML(html + sp.exit())
```

Refer to the following screenshot for the end result:



See also

- ▶ The Wikipedia page for bagging at https://en.wikipedia.org/wiki/Bootstrap_aggregating (retrieved November 2015)
- ▶ The documentation for the BaggingClassifier at <http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.BaggingClassifier.html> (retrieved November 2015)

Boosting for better learning

Strength in numbers is the reason why large countries tend to be more successful than small countries. That doesn't mean that a person in a large country has a better life. But for the big picture, the individual person doesn't matter that much, just like in an ensemble of decision trees the results of a single tree can be ignored if we have enough trees.

In the context of classification, we define **weak learners** as learners that are just a little better than a baseline such as randomly assigning classes. Although weak learners are weak individually, like ants, together they can do amazing things just like ants can.

It makes sense to take into account the strength of each individual learner using weights. This general idea is called **boosting**. There are many boosting algorithms, of which we will use **AdaBoost** in this recipe. Boosting algorithms differ mostly in their weighting scheme.

AdaBoost uses a weighted sum to produce the final result. It is an adaptive algorithm that tries to boost the results for individual training examples. If you have studied for an exam, you may have applied a similar technique by identifying the type of questions you had trouble with and focusing on the difficult problems. In the case of AdaBoost, boosting is done by tweaking the weak learners.

How to do it...

The program is in the `boosting.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import ch9util
from sklearn.grid_search import GridSearchCV
from sklearn.ensemble import AdaBoostRegressor
from sklearn.tree import DecisionTreeRegressor
import numpy as np
import daultil as dl
from IPython.display import HTML
```

2. Load the data and create an AdaBoostRegressor class:

```
X_train, X_test, y_train, y_test = ch9util.temp_split()
params = {
    'loss': ['linear', 'square', 'exponential'],
    'base_estimator__min_samples_leaf': [1, 2]
}
reg = AdaBoostRegressor(base_estimator=DecisionTreeRegressor(random_state=28),
                        random_state=17)
```

-
3. Grid search, fit, and predict as follows:

```
gscv = GridSearchCV(estimator=reg,
                     param_grid=params, cv=5, n_jobs=-1)
gscv.fit(X_train, y_train)
preds = gscv.predict(X_test)
```

4. Scatter plot the predictions against the actual values:

```
sp = dl.plotting.Subplotter(2, 2, context)
html = ch9util.scatter_predictions(preds, y_test, gscv.best_
params_,
                                    gscv.best_score_, sp.ax)
```

5. Plot a validation curve for a range of ensemble sizes:

```
n_estimators = 2 ** np.arange(3, 9)
ch9util.plot_validation(sp.next_ax(), gscv.best_estimator_,
                        X_train, y_train, 'n_estimators',
                        n_estimators)
```

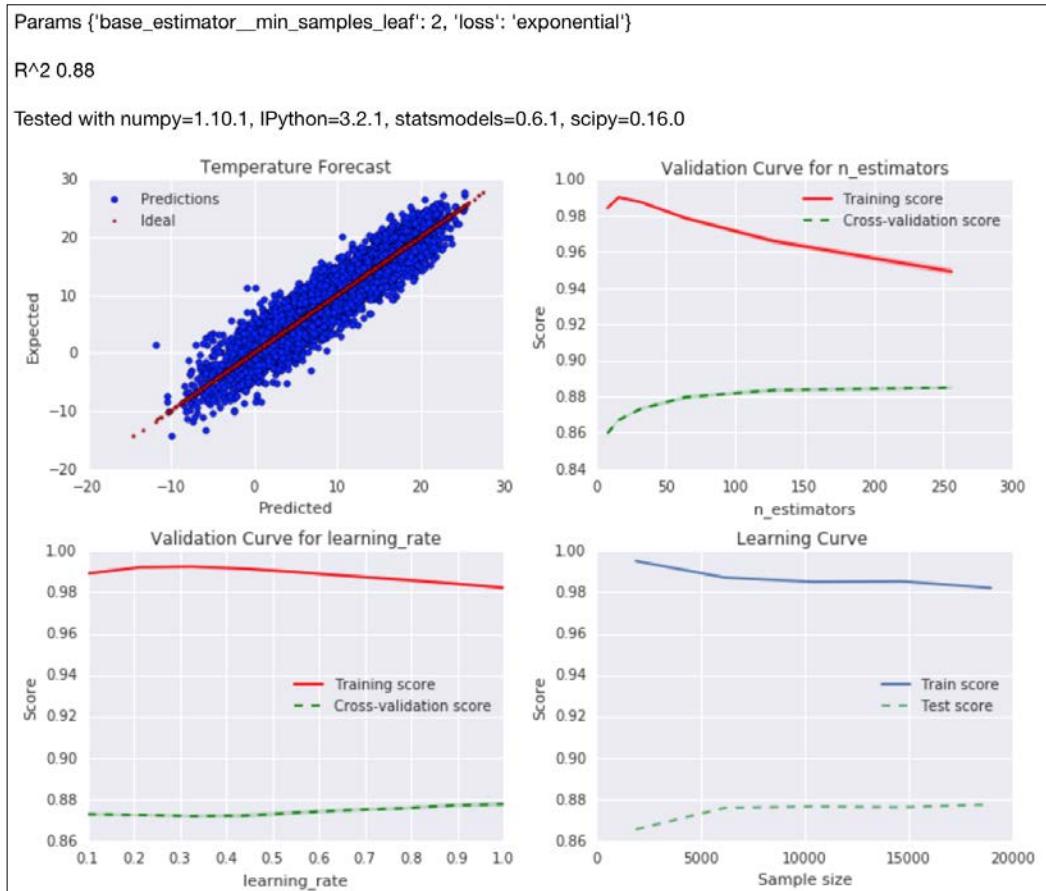
6. Plot a validation curve for a range of learning rates:

```
learn_rate = np.linspace(0.1, 1, 9)
ch9util.plot_validation(sp.next_ax(), gscv.best_estimator_,
                        X_train, y_train, 'learning_rate', learn_
rate)
```

7. Plot the learning curve as follows:

```
ch9util.plot_learn_curve(sp.next_ax(), gscv.best_estimator_,
                         X_train, y_train)
HTML(html + sp.exit())
```

Refer to the following screenshot for the end result:



See also

- ▶ The Wikipedia page about boosting at https://en.wikipedia.org/wiki/Boosting_%28machine_learning%29 (retrieved November 2015)
- ▶ The Wikipedia page about AdaBoost at <https://en.wikipedia.org/wiki/AdaBoost> (retrieved November 2015)
- ▶ The documentation for the AdaBoostRegressor class at <http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostRegressor.html> (retrieved November 2015)

Nesting cross-validation

If we are fitting data to a straight line, the parameters of the mathematical model will be the slope and intercept of the line. When we determine the parameters of a model, we fit the model on a subset of the data (training set), and we evaluate the performance of the model on the rest of the data (test set). This is called **validation** and there are more elaborate schemes. The scikit-learn GridSearchCV class uses k-fold cross-validation, for example.

Classifiers and regressors usually require extra parameters (hyperparameters) such as the number of components of an ensemble, which usually have nothing to do with the linear model as mentioned in the first sentence. It's a bit confusing to talk about models because we have models with plain parameters and a bigger model with hyperparameters.

Let's call the bigger model a level 2 model, although this is not standard nomenclature as far as I know. If we are using GridSearchCV to obtain the hyperparameters of the level 2 model, we have another set of parameters (not hyperparameters or level 1 parameters) to worry about—the number of folds and the metric used for comparison. Evaluation metrics have not passed the review yet (refer to *Chapter 10, Evaluating Classifiers, Regressors, and Clusters*), but there are more metrics than we used in this chapter. Also, we might worry whether we determined the hyperparameters on the same data as used to evaluate the results. One solution is to apply **nested cross-validation**.

Nested cross-validation consists of the following cross-validations:

- ▶ The inner cross-validation does hyperparameter optimization, for instance using grid search
- ▶ The outer cross-validation is used to evaluate performance and do statistical analysis

In this recipe, we will look at the following distributions:

- ▶ The distribution of all the scores
- ▶ The distribution of the best scores for each outer cross-validation iteration reported by GridSearchCV
- ▶ The distribution of the mean scores for each fold
- ▶ The distribution of the standard deviations of scores within a GridSearchCV iteration

How to do it...

The code is in the `nested_cv.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
from sklearn.grid_search import GridSearchCV
from sklearn.cross_validation import ShuffleSplit
from sklearn.cross_validation import cross_val_score
```

```
import daultil as dl
from sklearn.ensemble import ExtraTreesRegressor
from joblib import Memory
import numpy as np
from IPython.display import HTML

memory = Memory(cachedir='.'
```

2. Get the R-squared scores as described in the previous section:

```
@memory.cache
def get_scores():
    df = dl.data.Weather.load()[['WIND_SPEED', 'TEMP',
'PRESSURE']].dropna()
    X = df.values[:-1]
    y = df['TEMP'][1:]

    params = { 'min_samples_split': [1, 3],
               'min_samples_leaf': [3, 4]}

    gscv = GridSearchCV(ExtraTreesRegressor(bootstrap=True,
                                            random_state=37),
                         param_grid=params, n_jobs=-1, cv=5)
    cv_outer = ShuffleSplit(len(X), n_iter=500,
                           test_size=0.3, random_state=55)

    r2 = []
    best = []
    means = []
    stds = []

    for train_indices, test_indices in cv_outer:
        train_i = X[train_indices], y[train_indices]
        gscv.fit(*train_i)
        test_i = X[test_indices]
        gscv.predict(test_i)
        grid_scores = dl.collect.flatten([g.cv_validation_scores
                                         for g in gscv.grid_scores_])
        r2.extend(grid_scores)
        means.extend(dl.collect.flatten([g.mean_validation_score
                                         for g in gscv.grid_scores_]))
        stds.append(np.std(grid_scores))
        best.append(gscv.best_score_)

    return {'r2': r2, 'best': best, 'mean': means, 'std': stds}
```

3. Get the scores and load them into NumPy arrays:

```
scores = get_scores()
r2 = np.array(scores['r2'])
avgs = np.array(scores['mean'])
stds = np.array(scores['std'])
best = np.array(scores['best'])
```

4. Plot the distributions as follows:

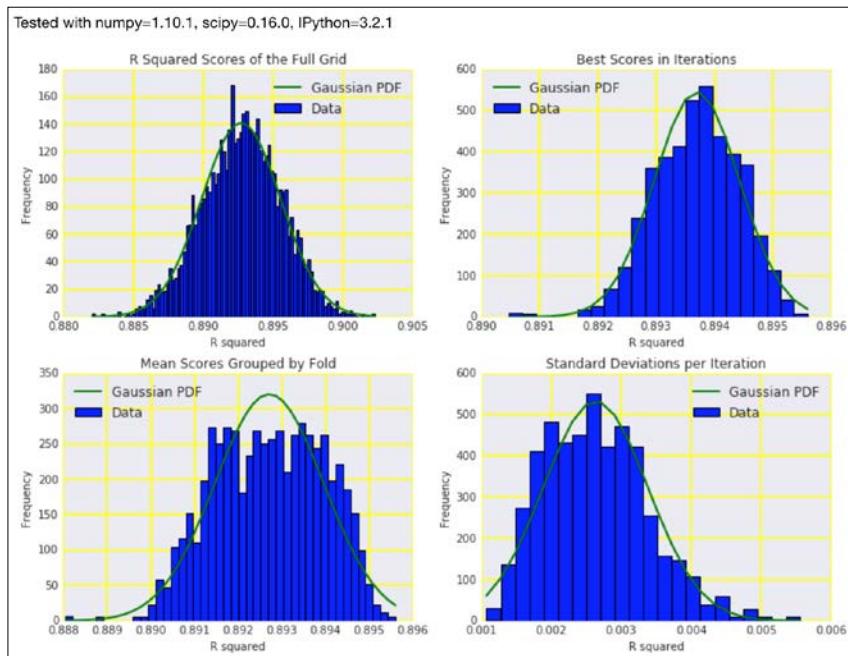
```
sp = dl.plotting.Subplotter(2, 2, context)
dl.plotting.hist_norm_pdf(sp.ax, r2)
sp.label()

dl.plotting.hist_norm_pdf(sp.next_ax(), best)
sp.label()

dl.plotting.hist_norm_pdf(sp.next_ax(), avgs)
sp.label()

dl.plotting.hist_norm_pdf(sp.next_ax(), stds)
sp.label()
HTML(sp.exit())
```

Refer to the following screenshot for the end result (distributions of cross-validation results):



See also

- ▶ The Wikipedia page about cross-validation at https://en.wikipedia.org/wiki/Cross-validation_%28statistics%29 (retrieved November 2015)
- ▶ The Wikipedia page about hyperparameter optimization at https://en.wikipedia.org/wiki/Hyperparameter_optimization (retrieved November 2015)

Reusing models with joblib

The joblib Memory class is a utility class that facilitates caching of function or method results to disk. We create a Memory object by specifying a caching directory. We can then decorate the function to cache or specify methods to cache in a class constructor. If you like, you can specify the arguments to ignore. The default behavior of the Memory class is to remove the cache any time the function is modified or the input values change. Obviously, you can also remove the cache manually by moving or deleting cache directories and files.

In this recipe, I describe how to reuse a scikit-learn regressor or classifier. The naïve method would be to store the object in a standard Python pickle or use joblib. However, in most cases, it is better to store the hyperparameters of the estimator.

We will use the ExtraTreesRegressor class as estimator. **Extra trees (extremely randomized trees)** are a variation of the random forest algorithm, which is covered in the *Learning with random forests* recipe.

How to do it...

1. The imports are as follows:

```
from sklearn.grid_search import GridSearchCV
from sklearn.ensemble import ExtraTreesRegressor
import ch9util
from tempfile import mkdtemp
import os
import joblib
```

2. Load the data and define a hyperparameter grid search dictionary:

```
X_train, X_test, y_train, y_test = ch9util.temp_split()
params = {'min_samples_split': [1, 3],
          'bootstrap': [True, False],
          'min_samples_leaf': [3, 4]}
```

3. Do a grid search as follows:

```
gscv = GridSearchCV(ExtraTreesRegressor(random_state=41),
                     param_grid=params, cv=5)
```

4. Fit and predict as follows:

```
gscv.fit(X_train, y_train)
preds = gscv.predict(X_test)
```

5. Store the best parameters found by the grid search:

```
dir = mkdtemp()
pkl = os.path.join(dir, 'params.pkl')
joblib.dump(gscv.best_params_, pkl)
params = joblib.load(pkl)
print('Best params', gscv.best_params_)
print('From pkl', params)
```

6. Create a new estimator and compare the predictions:

```
est = ExtraTreesRegressor(random_state=41)
est.set_params(**params)
est.fit(X_train, y_train)
preds2 = est.predict(X_test)
print('Max diff', (preds - preds2).max())
```

Refer to the following screenshot for the end result:

```
Best params {'min_samples_split': 1, 'min_samples_leaf': 4, 'bootstrap': True}
From pkl {'bootstrap': True, 'min_samples_leaf': 4, 'min_samples_split': 1}
Max diff 0.0
```

The code is in the `reusing_models.py` file in this book's code bundle.

See also

- ▶ The documentation for the `Memory` class at <https://pythonhosted.org/joblib/memory.html> (retrieved November 2015)
- ▶ The Wikipedia page about random forests at https://en.wikipedia.org/wiki/Random_forest (retrieved November 2015)

Hierarchically clustering data

In *Python Data Analysis*, you learned about clustering—separating data into clusters without providing any hints—which is a form of unsupervised learning. Sometimes, we need to take a guess for the number of clusters, as we did in the *Clustering streaming data with Spark* recipe.

There is no restriction against having clusters contain other clusters. In such a case, we speak of **hierarchical clustering**. We need a distance metric to separate data points. Take a look at the following equations:

$$(9.2) \quad \|a - b\|_2 = \sqrt{\sum_i (a_i - b_i)^2}$$

$$(9.3) \quad \min \{d(a, b) : a \in A, b \in B\}$$

In this recipe, we will use Euclidean distance (9.2), provided by the SciPy `pdist()` function. The distance between sets of points is given by the linkage criteria. In this recipe, we will use the single-linkage criteria (9.3) provided by the SciPy `linkage()` function.

How to do it...

The script is in the `clustering_hierarchy.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
from scipy.spatial.distance import pdist
from scipy.cluster.hierarchy import linkage
from scipy.cluster.hierarchy import dendrogram
import dtautil as dl
import matplotlib.pyplot as plt
```

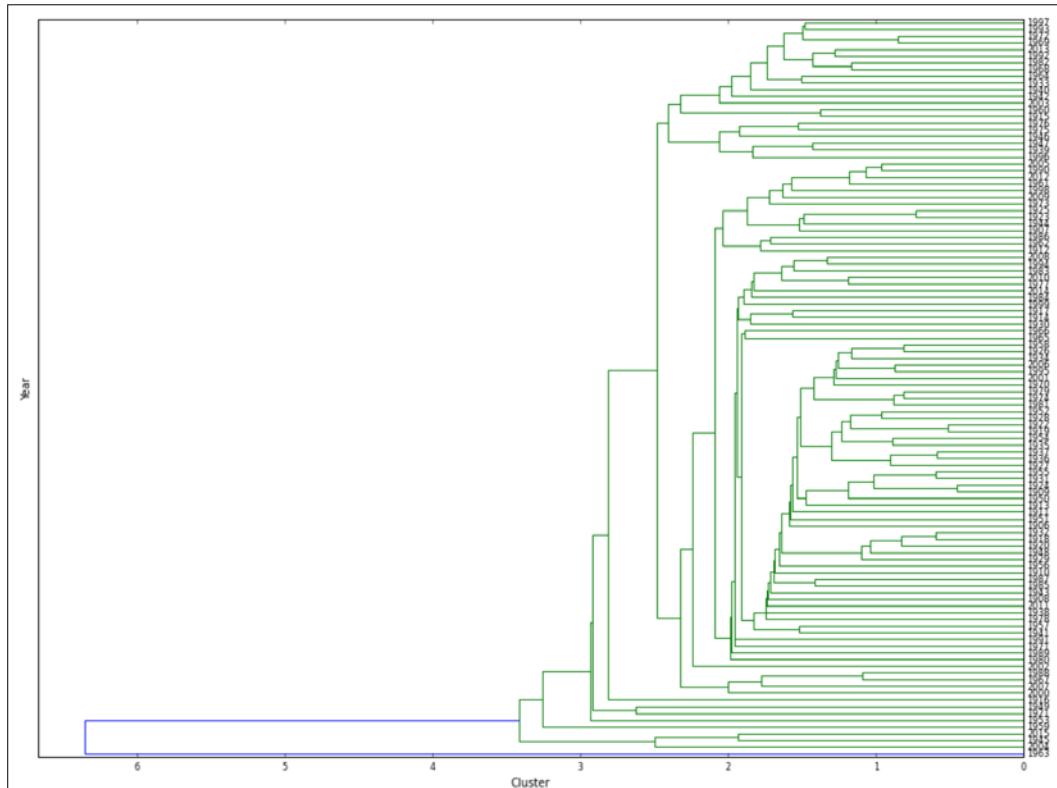
2. Load the data, resample to annual values, and compute distances:

```
df = dl.data.Weather.load().resample('A').dropna()
dist = pdist(df)
```

3. Plot the hierarchical cluster as follows:

```
dendrogram(linkage(dist), labels=[d.year for d in df.index],
            orientation='right')
plt.tick_params(labelsize=8)
plt.xlabel('Cluster')
plt.ylabel('Year')
```

Refer to the following screenshot for the end result:



See also

- ▶ The Wikipedia page about hierarchical clustering at https://en.wikipedia.org/wiki/Hierarchical_clustering (retrieved November 2015)
- ▶ The documentation for the `pdist()` function at <https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.pdist.html> (retrieved November 2015)
- ▶ The documentation for the `linkage()` function at <https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html> (retrieved November 2015)

Taking a Theano tour

Theano is a Python library created by a machine learning group in Montreal and is often associated with deep learning, although that is not necessarily its core purpose. Theano is tightly integrated with NumPy and can run code on CPU or GPU. If you are interested in the GPU option, refer to the documentation listed in the *See also* section. Theano also supports symbolic differentiation through symbolic variables.

According to its documentation, Theano is a cross between NumPy and SymPy. It is possible to implement machine learning algorithms with Theano, but it's not as easy or convenient as using scikit-learn. However, you may get the potential advantages of higher parallelism and numerical stability.

In this recipe, we will perform linear regression of temperature data using **gradient descent**. Gradient descent is an optimization algorithm that we can use in a regression context to minimize fit residuals. The gradient measures how steep a function is. The algorithm takes many steps proportional to how steep the gradient is in order to find a local minimum. We are trying to go downhill, but we don't know in which direction we can find a local minimum. So, going for a large move down should on average get us down faster, but there is no guarantee. In some cases, it may help to smooth the function (smoother hill), so we don't spend a lot of time oscillating.

Getting ready

Install Theano with the following command:

```
$ pip install --no-deps git+git://github.com/Theano/Theano.git
```

I tested the code with the bleeding edge version as of November 2015.

How to do it...

The code is in the `theano_tour.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import theano
import numpy as np
import theano.tensor as T
import ch9util
from sklearn.cross_validation import train_test_split
from sklearn.metrics import r2_score
import dautil as dl
from IPython.display import HTML
```

2. Load the temperature data and define Theano symbolic variables:

```
temp = dl.data.Weather.load()['TEMP'].dropna()
X = temp.values[:-1]
y = temp.values[1:]
X_train, X_test, y_train, y_test = train_test_split(X, y, random_
state=16)
w = theano.shared(0., name ='w')
c = theano.shared(0., name ='c')

x = T.vector('x')
y = T.vector('y')
```

3. Define prediction and cost (loss) functions to minimize:

```
prediction = T.dot(x, w) + c
cost = T.sum(T.pow(prediction - y, 2))/(2 * X_train.shape[0])
Define gradient functions as follows:
gw = T.grad(cost, w)
gc = T.grad(cost, c)

learning_rate = 0.01
training_steps = 10000
```

4. Define the training function as follows:

```
train = theano.function([x, y], cost, updates =
[(w, w - learning_rate * gw),
(c, c - learning_rate * gc)])
predict = theano.function([x], prediction)
```

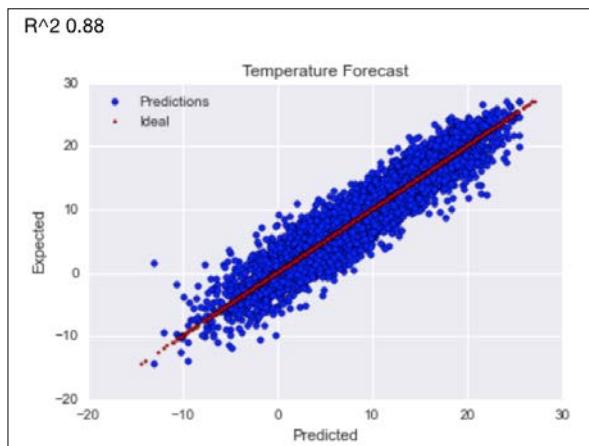
5. Train the estimator as follows:

```
for i in range(training_steps):
    train(X_train.astype(np.float), y_train)
```

6. Predict and visualize the prediction as follows:

```
preds = predict(X_test)
r2 = r2_score(preds, y_test)
HTML(ch9util.scatter_predictions(preds, y_test, '', r2))
```

Refer to the following screenshot for the end result:



See also

- ▶ The Theano documentation at <http://deeplearning.net/software/theano/> (retrieved November 2015)
- ▶ The Wikipedia page about gradient descent at https://en.wikipedia.org/wiki/Gradient_descent (retrieved November 2015)

10

Evaluating Classifiers, Regressors, and Clusters

In this chapter, we will cover the following recipes:

- ▶ Getting classification straight with the confusion matrix
- ▶ Computing precision, recall, and F1-score
- ▶ Examining a receiver operating characteristic and the area under a curve
- ▶ Visualizing the goodness of fit
- ▶ Computing MSE and median absolute error
- ▶ Evaluating clusters with the mean silhouette coefficient
- ▶ Comparing results with a dummy classifier
- ▶ Determining MAPE and MPE
- ▶ Comparing with a dummy regressor
- ▶ Calculating the mean absolute error and the residual sum of squares
- ▶ Examining the kappa of classification
- ▶ Taking a look at the Matthews correlation coefficient

Introduction

Evaluating classifiers, regressors, and clusters is a critical multidimensional problem involving many aspects. Purely from an engineering perspective, we worry about speed, memory, and correctness. Under some circumstances, speed is everything. If memory is scarce, of course, we have to make that our priority. The world is a giant labyrinth full of choices, and you are sometimes forced to choose one model over others instead of using multiple models in an ensemble. We should, of course, inform our rational decision with appropriate evaluation metrics.

There are so many evaluation metrics out there that you would need multiple books to describe them all. Obviously, many of the metrics are very similar. Some of them are accepted and popular, and of those metrics, some are implemented in scikit-learn.

We will evaluate the classifiers and regressors from *Chapter 9, Ensemble Learning and Dimensionality Reduction*. We applied those estimators to the sample problem of weather forecasting. This is not necessarily a problem at which humans are good. Achieving human performance is the goal for some problems, such as face recognition, character recognition, spam classification, and sentiment analysis. As a baseline to beat, we often choose some form of random guessing.

Getting classification straight with the confusion matrix

Accuracy is a metric that measures how well a model has performed in a given context. Accuracy is the default evaluation metric of scikit-learn classifiers. Unfortunately, accuracy is one-dimensional, and it doesn't help when the classes are unbalanced. The rain data we examined in *Chapter 9, Ensemble Learning and Dimensionality Reduction*, is pretty balanced. The number of rainy days is almost equal to the number of days on which it doesn't rain. In the case of e-mail spam classification, at least for me, the balance is shifted toward spam.

A **confusion matrix** is a table that is usually used to summarize the results of classification. The two dimensions of the table are the predicted class and the target class. In the context of binary classification, we talk about positive and negative classes. Naming a class negative is arbitrary—it doesn't necessarily mean that it is bad in some way. We can reduce any multi-class problem to one class versus the rest of the problem; so, when we evaluate binary classification, we can extend the framework to multi-class classification. A class can either be correctly predicted or not; we label those instances with the words true and false accordingly.

We have four combinations of true, false, positive, and negative, as described in the following table:

	Predicted class	
Target class	True positives: It rained and we correctly predicted it.	False positives: We incorrectly predicted that it would rain.
	False negatives: It did rain, but we predicted that it wouldn't.	True negatives: It didn't rain, and we correctly predicted it.

How to do it...

1. The imports are as follows:

```
import numpy as np
from sklearn.metrics import confusion_matrix
import seaborn as sns
import dautil as dl
from IPython.display import HTML
import ch10util
```

2. Define the following function to plot the confusion matrix:

```
def plot_cm(preds, y_test, title, cax):
    cm = confusion_matrix(preds.T, y_test)
    normalized_cm = cm/cm.sum().astype(float)
    sns.heatmap(normalized_cm, annot=True, fmt='%.2f', vmin=0,
    vmax=1,
                xticklabels=['Rain', 'No Rain'],
                yticklabels=['Rain', 'No Rain'], ax=cax)
    cax.set_xlabel('Predicted class')
    cax.set_ylabel('Expected class')
    cax.set_title('Confusion Matrix for Rain Forecast | ' + title)
```

3. Load the target values and plot the confusion matrix for the random forest classifier, bagging classifier, voting, and stacking classifier:

```
y_test = np.load('rain_y_test.npy')
sp = dl.plotting.Subplotter(2, 2, context)

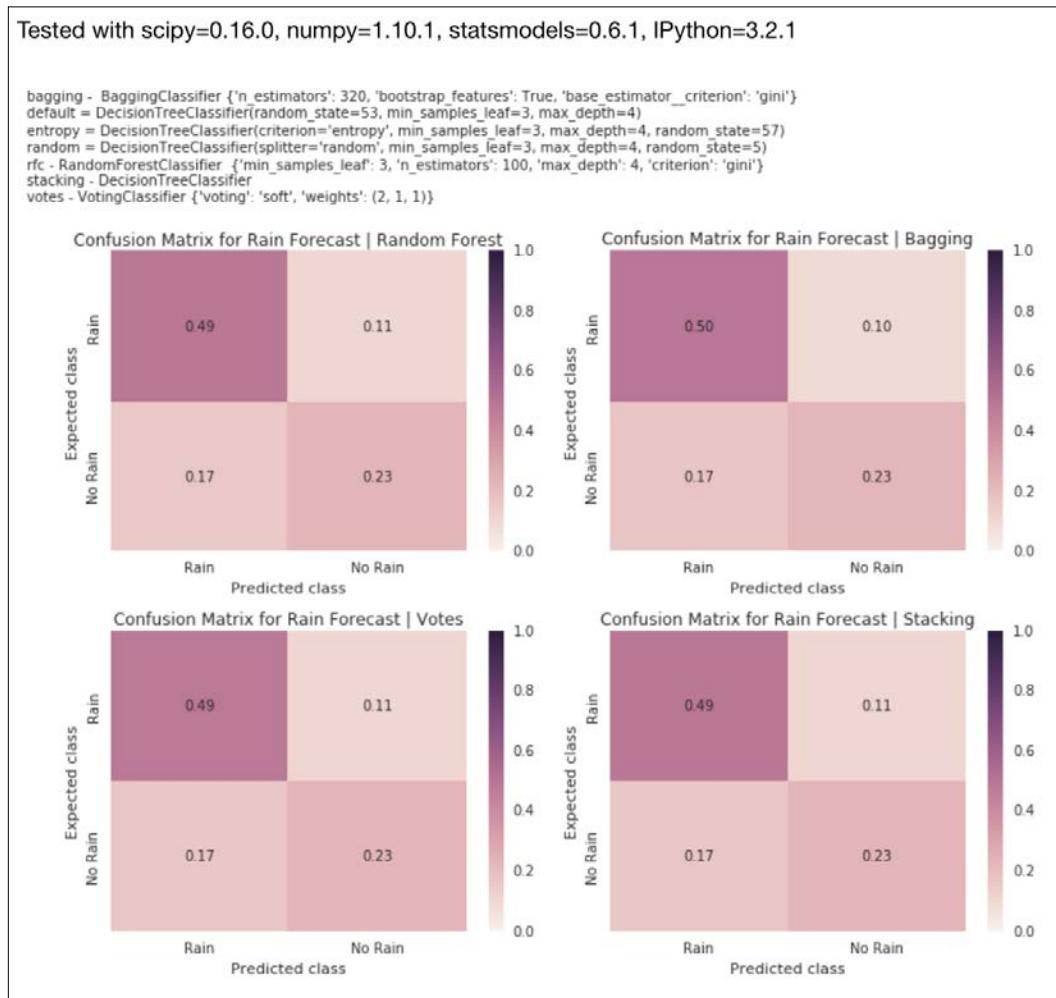
plot_cm(y_test, np.load('rfc.npy'), 'Random Forest', sp.ax)

plot_cm(y_test, np.load('bagging.npy'), 'Bagging', sp.next_ax())

plot_cm(y_test, np.load('votes.npy'), 'Votes', sp.next_ax())

plot_cm(y_test, np.load('stacking.npy'), 'Stacking', sp.next_ax())
sp.fig.text(0, 1, ch10util.classifiers())
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The source code is in the `conf_matrix.ipynb` file in this book's code bundle.

How it works

We displayed four confusion matrices for four classifiers, and the four numbers of each matrix seem to be repeating. Of course, the numbers are not exactly equal; however, you have to allow for some random variation.

See also

- ▶ The Wikipedia page about the confusion matrix at https://en.wikipedia.org/wiki/Confusion_matrix (retrieved November 2015)
- ▶ The `confusion_matrix()` function documented at http://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html (retrieved November 2015)

Computing precision, recall, and F1-score

In the *Getting classification straight with the confusion matrix* recipe, you learned that we can label classified samples as true positives, false positives, true negatives, and false negatives. With the counts of these categories, we can calculate many evaluation metrics of which we will cover four in this recipe, as given by the following equations:

$$(10.1) \quad Accuracy = \frac{T_p + T_n}{T_p + T_n + F_p + F_n}$$

$$(10.2) \quad Precision = \frac{T_p}{T_p + F_p}$$

$$(10.3) \quad Recall = \frac{T_p}{T_p + F_n}$$

$$(10.4) \quad F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

These metrics range from zero to one, with zero being the worst theoretical score and one being the best. Actually, the worst score would be the one we get by random guessing. The best score in practice may be lower than one because in some cases we can only hope to emulate human performance, and there may be ambiguity about what correct classification should be, for instance, in the case of sentiment analysis (covered in the *Python Data Analysis* book).

- ▶ The accuracy (10.1) is the ratio of correct predictions.
- ▶ **Precision** (10.2) measures relevance as the likelihood of classifying a negative class sample as positive. Choosing which class is positive is somewhat arbitrary, but let's say that a rainy day is positive. High precision would mean that we labeled a relatively small number of non-rainy (negative) days as rainy. For a search (web, database, or other), it would mean a relatively high number of relevant results.

- ▶ **Recall** (10.3) is the likelihood of finding all the positive samples. If again, rainy days are our positive class, the more rainy days are classified correctly, the higher the recall. For a search, we can get a perfect recall by returning all the documents because this will automatically return all the relevant documents. A human brain is a bit like a database, and in that context, recall will mean the likelihood of remembering, for instance, how a certain Python function works.
- ▶ The **F1 score** (10.4) is the harmonic mean of precision and recall (actually, there are multiple variations of the F1 score). The G score uses the geometric mean; but, as far as I know, it is less popular. The idea behind the F1 score, related F scores and G scores, is to combine the precision and recall. That doesn't necessarily make it the best metric. There are other metrics you may prefer, such as the **Matthews correlation coefficient** (refer to the *Taking a look at the Matthews correlation coefficient* recipe) and **Cohen's kappa** (refer to the *Examining kappa of classification* recipe). When we face the choice of so many classification metrics, we obviously want the best metric. However, you have to make the choice based on your situation, as there is no metric that fits all.

How to do it...

1. The imports are as follows:

```
import numpy as np
from sklearn import metrics
import ch10util
import dautil as dl
from IPython.display import HTML
```

2. Load the target values and calculate the metrics:

```
y_test = np.load('rain_y_test.npy')
accuracies = [metrics.accuracy_score(y_test, preds)
              for preds in ch10util.rain_preds()]
precisions = [metrics.precision_score(y_test, preds)
               for preds in ch10util.rain_preds()]
recalls = [metrics.recall_score(y_test, preds)
            for preds in ch10util.rain_preds()]
f1s = [metrics.f1_score(y_test, preds)
       for preds in ch10util.rain_preds()]
```

3. Plot the metrics for the rain forecasts:

```
sp = dl.plotting.Subplotter(2, 2, context)
ch10util.plot_bars(sp.ax, accuracies)
sp.label()
```

```

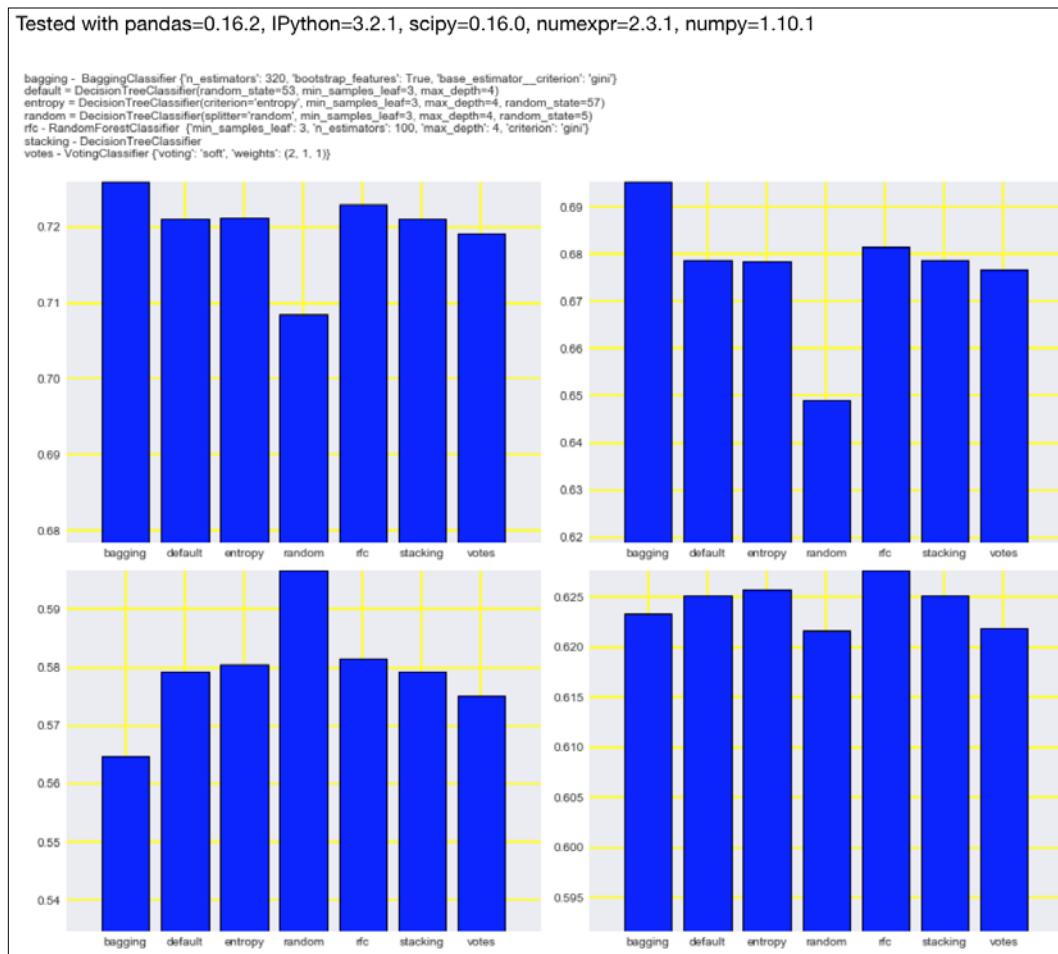
ch10util.plot_bars(sp.next_ax(), precisions)
sp.label()

ch10util.plot_bars(sp.next_ax(), recalls)
sp.label()

ch10util.plot_bars(sp.next_ax(), f1s)
sp.label()
sp.fig.text(0, 1, ch10util.classifiers())
HTML(sp.exit())

```

Refer to the following screenshot for the end result:



The code is in the `precision_recall.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about precision and recall at https://en.wikipedia.org/wiki/Precision_and_recall (retrieved November 2015)
- ▶ The `precision_score()` function documented at http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html (retrieved November 2015)
- ▶ The `recall_score()` function documented at http://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html (retrieved November 2015)
- ▶ The `f1_score()` function documented at http://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html (retrieved November 2015)

Examining a receiver operating characteristic and the area under a curve

The **receiver operating characteristic (ROC)** is a plot of the recall (10.3) and the **false positive rate (FPR)** of a binary classifier. The FPR is given by the following equation:

$$(10.5) \quad FPR = \frac{F_p}{F_p + T_n}$$

In this recipe, we will plot the ROC for the various classifiers we used in *Chapter 9, Ensemble Learning and Dimensionality Reduction*. Also, we will plot the curve associated with random guessing and the ideal curve. Obviously, we want to beat the baseline and get as close as possible to the ideal curve.

The **area under the curve (AUC, ROC AUC, or AUROC)** is another evaluation metric that summarizes the ROC. AUC can also be used to compare models, but it provides less information than ROC.

How to do it...

1. The imports are as follows:

```
from sklearn import metrics
import numpy as np
import ch10util
import dautil as dl
from IPython.display import HTML
```

2. Load the data and calculate metrics:

```
y_test = np.load('rain_y_test.npy')
roc_aucs = [metrics.roc_auc_score(y_test, preds)
            for preds in ch10util.rain_preds()]
```

3. Plot the AUROC for the rain predictors:

```
sp = dl.plotting.Subplotter(2, 1, context)
ch10util.plot_bars(sp.ax, roc_aucs)
sp.label()
```

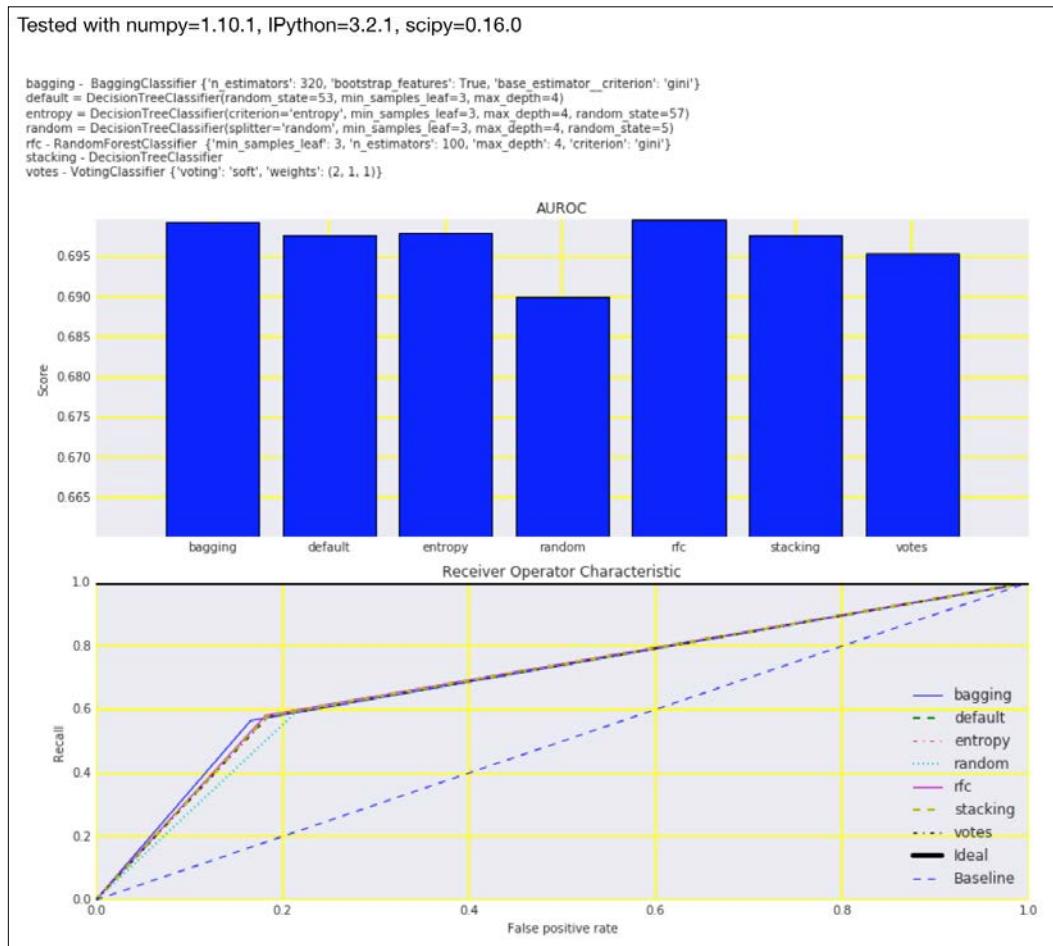
4. Plot the ROC curves for the rain predictors:

```
cp = dl.plotting.CyclePlotter(sp.next_ax())

for preds, label in zip(ch10util.rain_preds(),
                        ch10util.rain_labels()):
    fpr, tpr, _ = metrics.roc_curve(y_test, preds,
                                    pos_label=True)
    cp.plot(fpr, tpr, label=label)

fpr, tpr, _ = metrics.roc_curve(y_test, y_test)
sp.ax.plot(fpr, tpr, 'k', lw=4, label='Ideal')
sp.ax.plot(np.linspace(0, 1), np.linspace(0, 1),
           '--', label='Baseline')
sp.label()
sp.fig.text(0, 1, ch10util.classifiers())
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code is in the `roc_auc.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about the ROC at https://en.wikipedia.org/wiki/Receiver_operating_characteristic (retrieved November 2015)
- ▶ The `roc_auc_score()` function documented at http://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html (retrieved November 2015)

Visualizing the goodness of fit

We expect, or at least hope, that the residuals of regression are just random noise. If that is not the case, then our regressor may be ignoring information. We expect the residuals to be independent and normally distributed. It is relatively easy to check with a histogram or a QQ plot. In general, we want the mean of the residuals to be as close to zero as possible, and we want the variance of the residuals to be as small as possible. An ideal fit will have zero-valued residuals.

How to do it...

1. The imports are as follows:

```
import numpy as np
import matplotlib.pyplot as plt
import dautil as dl
import seaborn as sns
from scipy.stats import probplot
from IPython.display import HTML
```

2. Load the target and predictions for the boosting regressor:

```
y_test = np.load('temp_y_test.npy')
preds = np.load('boosting.npy')
```

3. Plot the actual and predicted values as follows:

```
sp = dl.plotting.Subplotter(2, 2, context)
cp = dl.plotting.CyclePlotter(sp.ax)
cp.plot(y_test)
cp.plot(preds)
sp.ax.set_ylabel(dl.data.Weather.get_header('TEMP'))
sp.label()
```

4. Plot the residuals on their own as follows:

```
residuals = preds - y_test
sp.next_ax().plot(residuals)
sp.label()
```

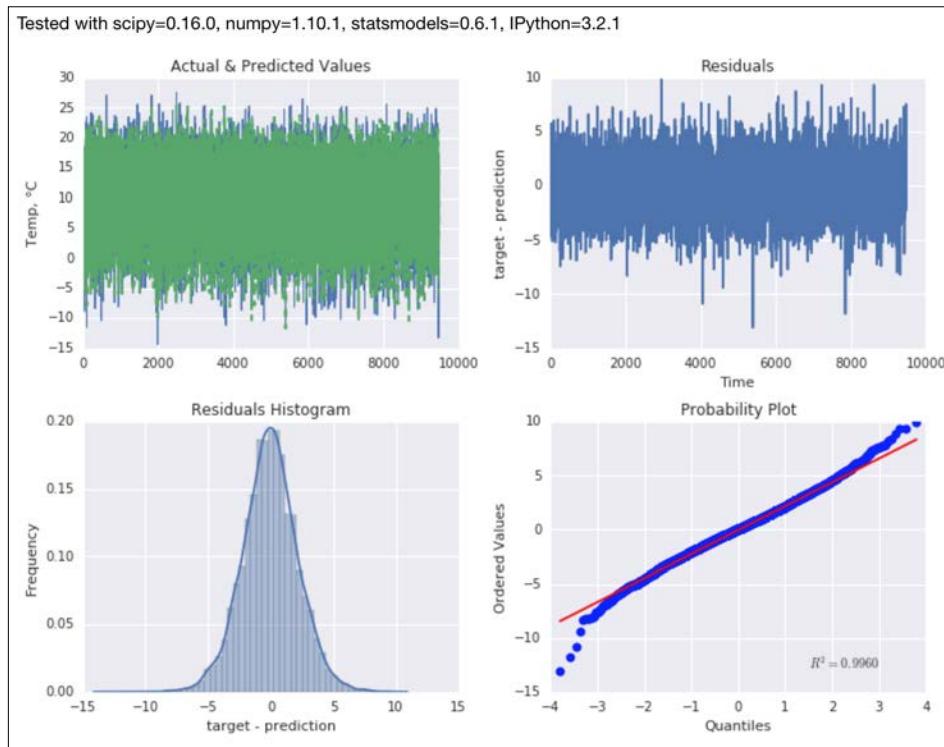
5. Plot the distribution of the residuals:

```
sns.distplot(residuals, ax=sp.next_ax())
sp.label()
```

6. Plot a QQ plot of the residuals:

```
probplot(residuals, plot=sp.next_ax())
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code is in the `visualizing_goodness.ipynb` file in this book's code bundle.

See also

- The `probplot()` function documented at <https://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.stats.probplot.html> (retrieved November 2015)

Computing MSE and median absolute error

The **mean squared error (MSE)** and **median absolute error (MedAE)** are popular regression metrics. They are given by the following equations:

$$(10.6) \quad MSE = \frac{1}{n} \sum_{i=1}^n (\hat{Y}_i - Y_i)^2$$

$$(10.7) \quad MedAE(y, \hat{y}) = \text{median}(|y_1 - \hat{y}_1|, \dots, |y_n - \hat{y}_n|)$$

The MSE (10.6) is analogous to population variance. The square root of the MSE (**RMSE**) is, therefore, analogous to standard deviation. The units of the MSE are the same as the variable under analysis—in our case, temperature. An ideal fit has zero-valued residuals and, therefore, its MSE is equal to zero. Since we are dealing with squared errors, the MSE has values that are larger or ideally equal to zero.

The MedAE is similar to the MSE, but we start with the absolute values of the residuals, and we use the median instead of the mean as the measure for centrality. The MedAE is also analogous to variance and is ideally zero or very small. Taking the absolute value instead of squaring potentially avoids numerical instability and speed issues, and the median is more robust for outliers than the mean. Also, taking the square tends to emphasize larger errors.

In this recipe, we will plot bootstrapped populations of MSE and MedAE for the regressors from Chapter 9, *Ensemble Learning and Dimensionality Reduction*.

How to do it...

1. The imports are as follows:

```
from sklearn import metrics
import ch10util
from IPython.display import HTML
import dautil as dl
from IPython.display import HTML
```

2. Plot the distributions of the metrics for the temperature predictors:

```
sp = dl.plotting.Subplotter(3, 2, context)
ch10util.plot_bootstrap('boosting',
                        metrics.mean_squared_error, sp.ax)
sp.label()

ch10util.plot_bootstrap('boosting',
                        metrics.median_absolute_error, sp.next_
ax())
sp.label()

ch10util.plot_bootstrap('etr',
                        metrics.mean_squared_error, sp.next_ax())
sp.label()

ch10util.plot_bootstrap('etr',
                        metrics.median_absolute_error, sp.next_
ax())
sp.label()
```

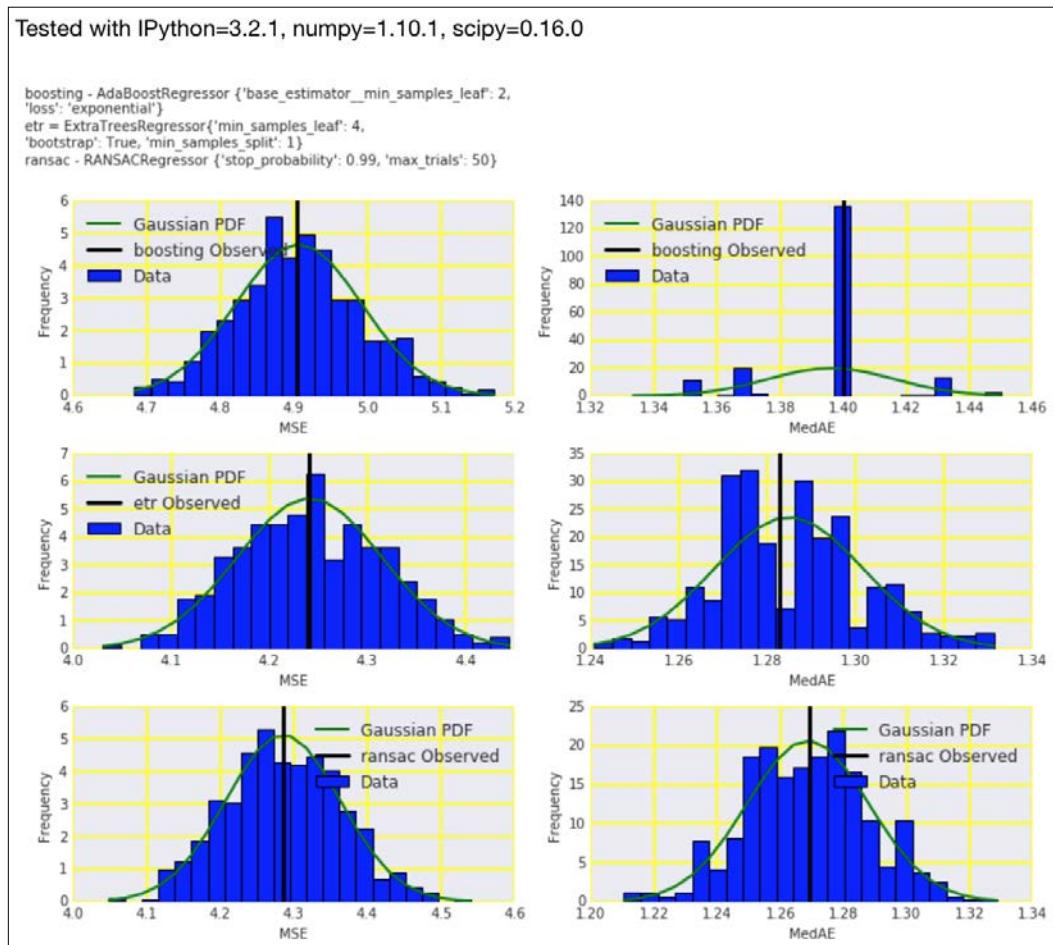
```

ch10util.plot_bootstrap('ransac',
                        metrics.mean_squared_error, sp.next_ax())
sp.label()

ch10util.plot_bootstrap('ransac',
                        metrics.median_absolute_error, sp.next_
ax())
sp.label()
sp.fig.text(0, 1, ch10util.regressors())
HTML(sp.exit())

```

Refer to the following screenshot for the end result:



The code is in the `mse.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about the MSE at https://en.wikipedia.org/wiki/Mean_squared_error (retrieved November 2015)
- ▶ The `mean_squared_error()` function documented at http://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html (retrieved November 2015)
- ▶ The `median_absolute_error()` function documented at http://scikit-learn.org/stable/modules/generated/sklearn.metrics.median_absolute_error.html (retrieved November 2015)

Evaluating clusters with the mean silhouette coefficient

Clustering is an unsupervised machine learning type of analysis. Although we don't know in general what the best clusters are, we can still get an idea of how good the result of clustering is. One way is to calculate the **silhouette coefficients** as defined in the following equation:

$$(10.8) \quad s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

In the preceding equation, $a(i)$ is the average dissimilarity of sample i with respect to other samples in the same cluster. A small $a(i)$ indicates that the sample belongs in its cluster. $b(i)$ is the lowest average dissimilarity of i to other cluster. It indicates the next best cluster for i . If the silhouette coefficients $s(i)$ of a sample is close to 1, it means that the sample is properly assigned. The value of $s(i)$ varies between -1 to 1. The average of the silhouette coefficients of all samples measures the quality of the clusters.

We can use the mean silhouette coefficient to inform our decision for the number of clusters of the K-means clustering algorithm. The K-means clustering algorithm is covered in more detail in the *Clustering streaming data with Spark* recipe in Chapter 5, *Web Mining, Databases and Big Data*.

How to do it...

1. The imports are as follows:

```
import dutil as dl
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.metrics import silhouette_samples
from IPython.display import HTML
```

2. Define the following function to plot the silhouette samples:

```
def plot_samples(ax, years, labels, i, avg):
    silhouette_values = silhouette_samples(X, labels)
    dl.plotting.plot_text(ax, years, silhouette_values,
                          labels, add_scatter=True)
    ax.set_title('KMeans k={0} Silhouette avg={1:.2f}'.format(i,
    avg))
    ax.set_xlabel('Year')
    ax.set_ylabel('Silhouette score')
```

3. Load the data and resample it as follows:

```
df = dl.data.Weather.load().resample('A').dropna()
years = [d.year for d in df.index]
X = df.values
```

4. Plot the clusters for varying numbers of clusters:

```
sp = dl.plotting.Subplotter(2, 2, context)
avgs = []
rng = range(2, 9)

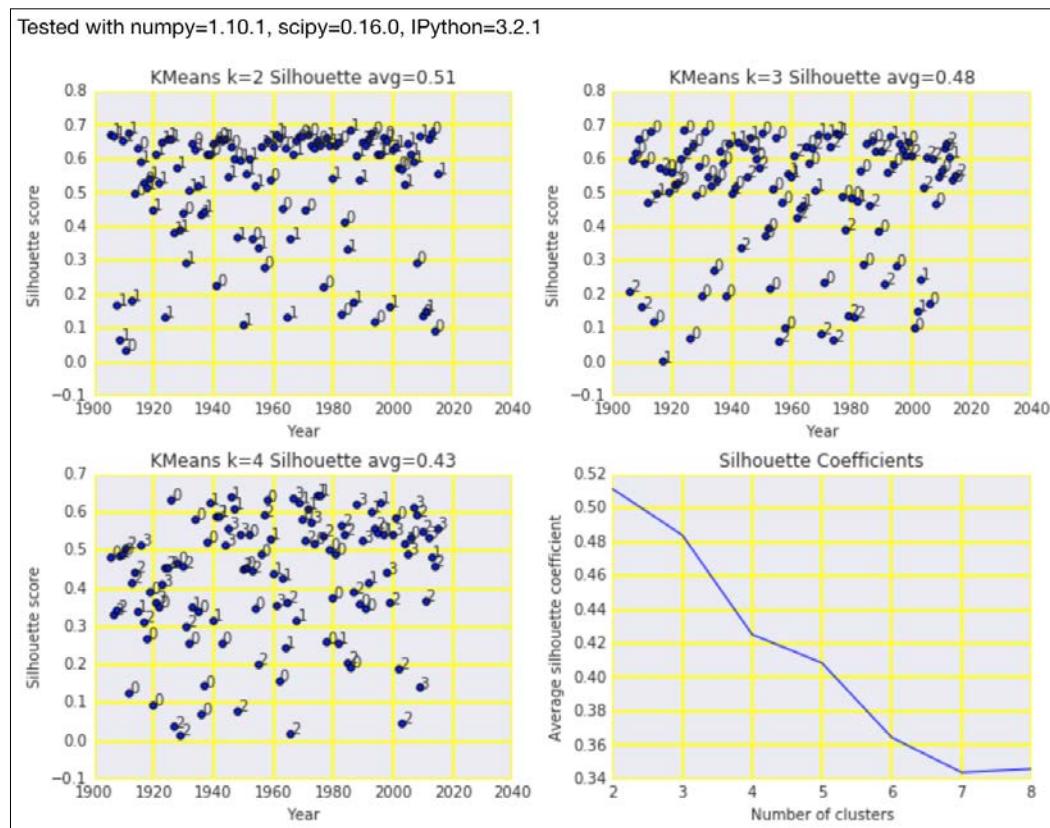
for i in rng:
    kmeans = KMeans(n_clusters=i, random_state=37)
    labels = kmeans.fit_predict(X)
    avg = silhouette_score(X, labels)
    avgs.append(avg)

    if i < 5:
        if i > 2:
            sp.next_ax()

    plot_samples(sp.ax, years, labels, i, avg)

sp.next_ax().plot(rng, avgs)
sp.label()
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code is in the `evaluating_clusters.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about the silhouette coefficient at https://en.wikipedia.org/wiki/Silhouette_%28clustering%29 (retrieved November 2015)
- ▶ The `silhouette_score()` function documented at http://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html (retrieved November 2015)

Comparing results with a dummy classifier

The scikit-learn `DummyClassifier` class implements several strategies for random guessing, which can serve as a baseline for classifiers. The strategies are as follows:

- ▶ `stratified`: This uses the training set class distribution
- ▶ `most_frequent`: This predicts the most frequent class
- ▶ `prior`: This is available in scikit-learn 0.17 and predicts by maximizing the class prior
- ▶ `uniform`: This uses an uniform distribution to randomly sample classes
- ▶ `constant`: This predicts a user-specified class

As you can see, some strategies of the `DummyClassifier` class always predict the same class. This can lead to warnings from some scikit-learn metrics functions. We will perform the same analysis as we did in the *Computing precision, recall, and F1 score* recipe, but with dummy classifiers added.

How to do it...

1. The imports are as follows:

```
import numpy as np
from sklearn import metrics
import ch10util
from sklearn.dummy import DummyClassifier
from IPython.display import HTML
import dautil as dl
```

2. Load the data as follows:

```
y_test = np.load('rain_y_test.npy')
X_train = np.load('rain_X_train.npy')
X_test = np.load('rain_X_test.npy')
y_train = np.load('rain_y_train.npy')
```

3. Create the dummy classifiers and predict with them:

```
stratified = DummyClassifier(random_state=28)
frequent = DummyClassifier(strategy='most_frequent',
                           random_state=28)
prior = DummyClassifier(strategy='prior', random_state=29)
uniform = DummyClassifier(strategy='uniform',
                           random_state=29)
```

```
preds = ch10util.rain_preds()

for clf in [stratified, frequent, prior, uniform]:
    clf.fit(X_train, y_train)
    preds.append(clf.predict(X_test))
```

4. Calculate metrics with the predictions as follows:

```
accuracies = [metrics.accuracy_score(y_test, p)
              for p in preds]
precisions = [metrics.precision_score(y_test, p)
              for p in preds]
recalls = [metrics.recall_score(y_test, p)
              for p in preds]
f1s = [metrics.f1_score(y_test, p)
              for p in preds]
```

5. Plot the metrics for the dummy and regular classifiers:

```
labels = ch10util.rain_labels()
labels.extend(['stratified', 'frequent',
               'prior', 'uniform'])

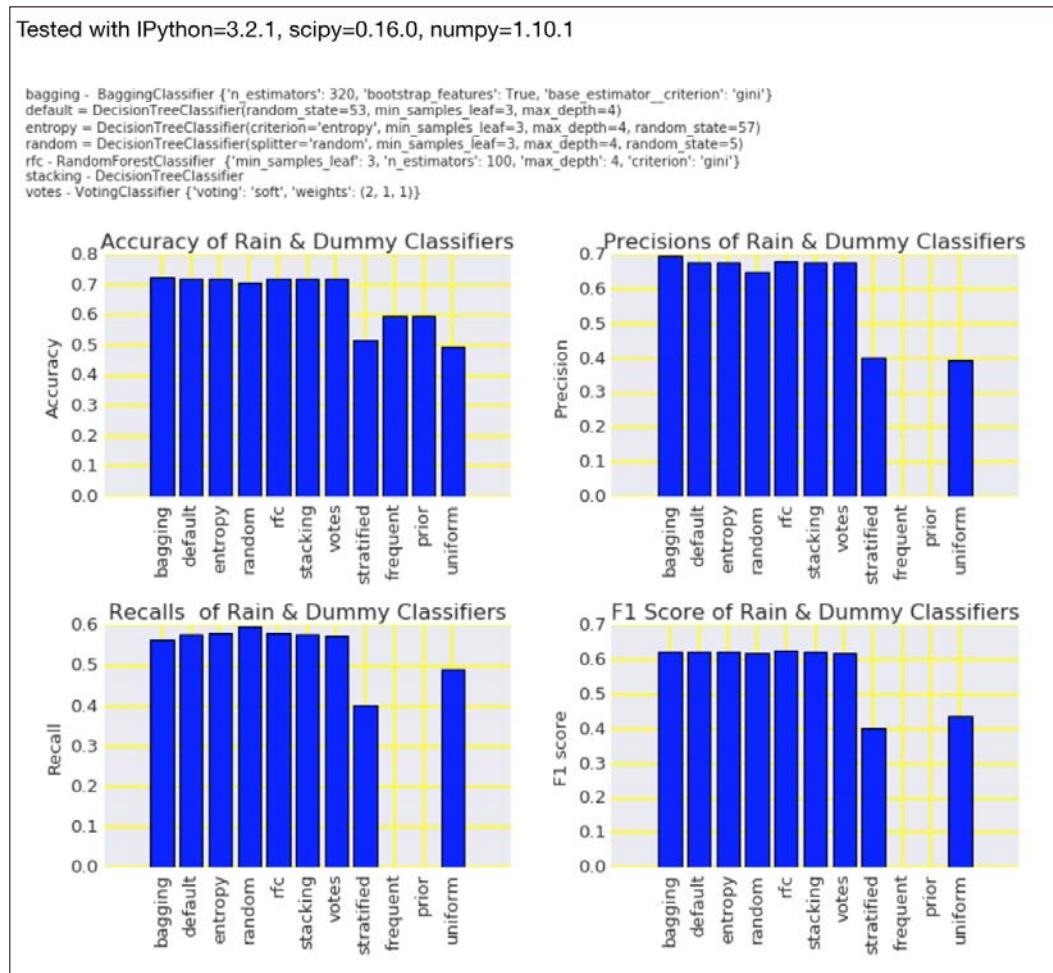
sp = dl.plotting.Subplotter(2, 2, context)
ch10util.plot_bars(sp.ax, accuracies, labels, rotate=True)
sp.label()

ch10util.plot_bars(sp.next_ax(), precisions, labels, rotate=True)
sp.label()

ch10util.plot_bars(sp.next_ax(), recalls, labels, rotate=True)
sp.label()

ch10util.plot_bars(sp.next_ax(), f1s, labels, rotate=True)
sp.label()
sp.fig.text(0, 1, ch10util.classifiers(), fontsize=10)
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code is in the `dummy_clf.ipynb` file in this book's code bundle.

See also

- ▶ The `DummyClassifier` class documented at <http://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html> (retrieved November 2015)

Determining MAPE and MPE

The **Mean Percentage Error (MPE)** and **Mean Absolute Percentage Error (MAPE)** express forecasting errors as ratios, and they are, therefore, dimensionless and easy to interpret. As you can see in the following equations, the disadvantage of MPE and MAPE is that we run the risk of dividing by zero:

$$(10.9) \quad MPE = \frac{100\%}{n} \sum_{t=1}^n \frac{a_t - f_t}{a_t}$$

$$(10.10) \quad MAPE = \frac{1}{n} \sum_{t=1}^n \left| \frac{A_t - F_t}{A_t} \right|$$

It is perfectly valid for the target variable to be equal to zero. For temperature, this happens to be the freezing point. Freezing often occurs in winter, so we either have to ignore those observations or add a constant large enough to avoid dividing by zero values. In the following section, it becomes clear that simply ignoring observations leads to strange bootstrap distributions.

How to do it...

1. The imports are as follows:

```
import ch10util
import dautil as dl
from IPython.display import HTML
```

2. Plot the bootstrapped metrics as follows:

```
sp = dl.plotting.Subplotter(3, 2, context)
ch10util.plot_bootstrap('boosting',
                        dl.stats.mape, sp.ax)
sp.label()

ch10util.plot_bootstrap('boosting',
                        dl.stats.mpe, sp.next_ax())
sp.label()

ch10util.plot_bootstrap('etr',
                        dl.stats.mape, sp.next_ax())
sp.label()

ch10util.plot_bootstrap('etr',
```

```

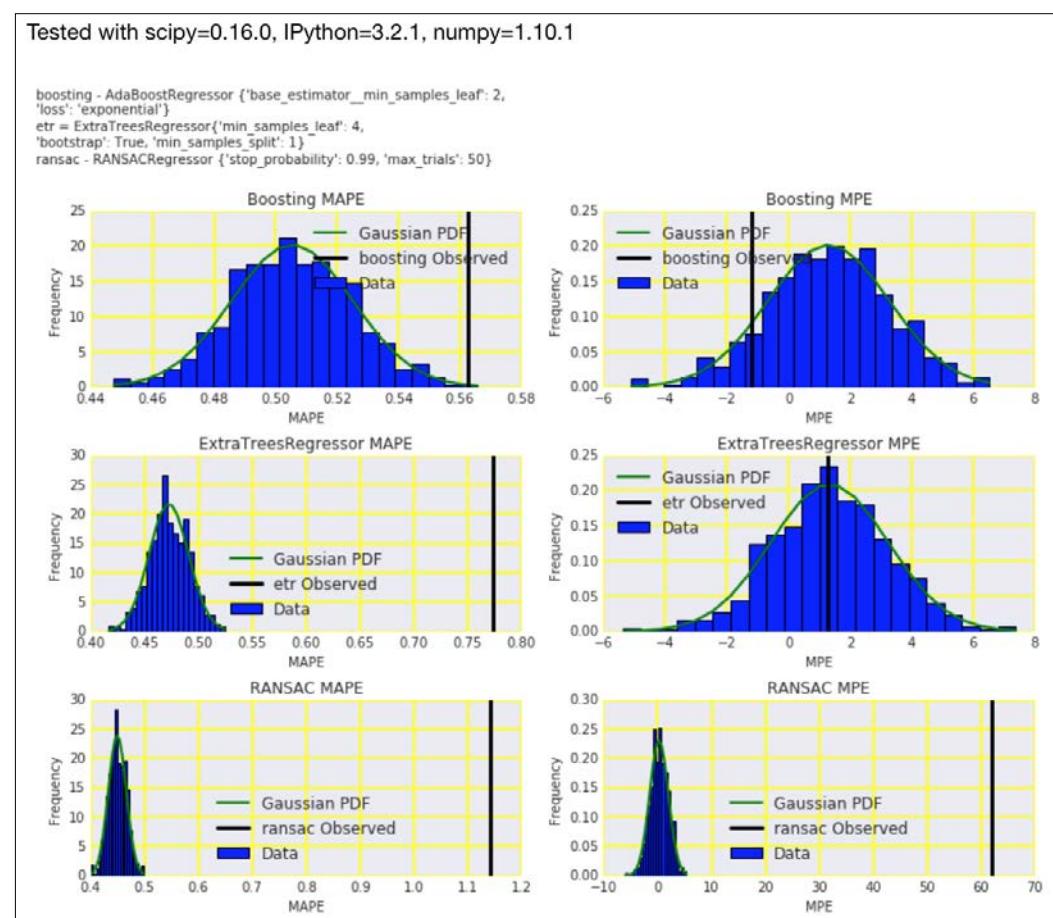
    dl.stats.mpe, sp.next_ax())
sp.label()

ch10util.plot_bootstrap('ransac',
                        dl.stats.mape, sp.next_ax())
sp.label()

ch10util.plot_bootstrap('ransac',
                        dl.stats.mpe, sp.next_ax())
sp.label()
sp.fig.text(0, 1, ch10util.regressors())
HTML(sp.exit())

```

Refer to the following screenshot for the end result:



The code is in the `mape_mpe.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about the MPE at https://en.wikipedia.org/wiki/Mean_percentage_error (retrieved November 2015)
- ▶ The Wikipedia page about the MAPE at https://en.wikipedia.org/wiki/Mean_absolute_percentage_error (retrieved November 2015)

Comparing with a dummy regressor

The scikit-learn DummyRegressor class implements several strategies for random guessing, which can serve as baseline for regressors. The strategies are as follows:

- ▶ `mean`: This predicts the mean of the training set.
- ▶ `median`: This predicts the median of the training set.
- ▶ `quantile`: This predicts a specified quantile of the training set when provided with the `quantile` parameter. We will apply this strategy by specifying the first and third quartile.
- ▶ `constant`: This predicts a constant value that is provided by the user.

We will compare the dummy regressors with the regressors from Chapter 9, *Ensemble Learning and Dimensionality Reduction*, using R-squared, MSE, MedAE, and MPE.

How to do it...

1. The imports are as follows:

```
import numpy as np
from sklearn.dummy import DummyRegressor
import ch10util
from sklearn import metrics
import dautil as dl
from IPython.display import HTML
```

2. Load the temperature data as follows:

```
y_test = np.load('temp_y_test.npy')
X_train = np.load('temp_X_train.npy')
X_test = np.load('temp_X_test.npy')
y_train = np.load('temp_y_train.npy')
```

3. Create dummy regressors using the available strategies and predict temperature with them:

```
mean = DummyRegressor()
median = DummyRegressor(strategy='median')
q1 = DummyRegressor(strategy='quantile', quantile=0.25)
q3 = DummyRegressor(strategy='quantile', quantile=0.75)

preds = ch10util.temp_preds()

for reg in [mean, median, q1, q3]:
    reg.fit(X_train, y_train)
    preds.append(reg.predict(X_test))
```

4. Calculate R-squared, MSE, median absolute error, and mean percentage error for the regular and dummy regressors:

```
r2s = [metrics.r2_score(p, y_test) for p in preds]
mses = [metrics.mean_squared_error(p, y_test)
        for p in preds]
maes = [metrics.median_absolute_error(p, y_test)
        for p in preds]
mpes = [dl.stats.mpe(y_test, p) for p in preds]

labels = ch10util.temp_labels()
labels.extend(['mean', 'median', 'q1', 'q3'])
```

5. Plot the metrics as follows:

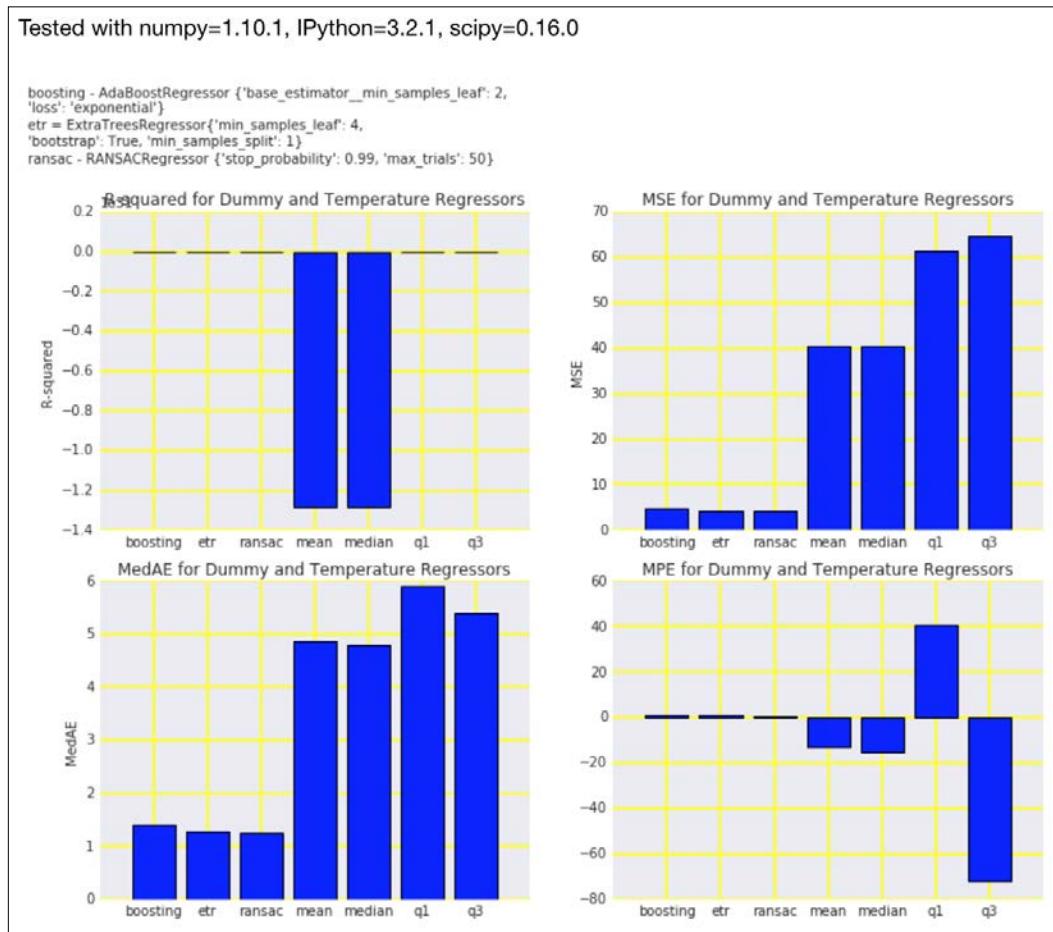
```
sp = dl.plotting.Subplotter(2, 2, context)
ch10util.plot_bars(sp.ax, r2s, labels)
sp.label()

ch10util.plot_bars(sp.next_ax(), mses, labels)
sp.label()

ch10util.plot_bars(sp.next_ax(), maes, labels)
sp.label()

ch10util.plot_bars(sp.next_ax(), mpes, labels)
sp.label()
sp.fig.text(0, 1, ch10util.regressors())
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code is in the `dummy_reg.ipynb` file in this book's code bundle.

See also

- ▶ The `DummyRegressor` class documented at <http://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyRegressor.html> (retrieved November 2015)
- ▶ The *Computing MSE and median absolute error* recipe in this chapter
- ▶ The *Determining MAPE and MPE* recipe in this chapter

Calculating the mean absolute error and the residual sum of squares

The **mean absolute error (MeanAE)** and **residual sum of squares (RSS)** are regression metrics given by the following equations:

$$(10.11) \quad MeanAE = \frac{1}{n} \sum_{i=1}^n |f_i - y_i| = \frac{1}{n} \sum_{i=1}^n |e_i|$$

$$(10.12) \quad RSS = \sum_{i=1}^n (y_i - f(x_i))^2$$

The mean absolute error (10.11) is similar to the MSE and MedAE, but it differs in one step of the calculation. The common feature of these metrics is that they ignore the sign of the error and are analogous to variance. MeanAE values are larger than or ideally equal to zero.

The RSS (10.12) is similar to the MSE, except we don't divide by the number of residuals. For this reason, you get larger values with the RSS. However, an ideal fit gives you a zero RSS.

How to do it...

1. The imports are as follows:

```
import ch10util
import dautil as dl
from sklearn import metrics
from IPython.display import HTML
```

2. Plot the bootstrapped metrics as follows:

```
sp = dl.plotting.Subplotter(3, 2, context)
ch10util.plot_bootstrap('boosting',
                        metrics.mean_absolute_error, sp.ax)
sp.label()

ch10util.plot_bootstrap('boosting',
                        dl.stats.rss, sp.next_ax())
sp.label()

ch10util.plot_bootstrap('etr',
                        metrics.mean_absolute_error, sp.next_ax())
sp.label()

ch10util.plot_bootstrap('etr',
                        dl.stats.rss, sp.next_ax())
```

```

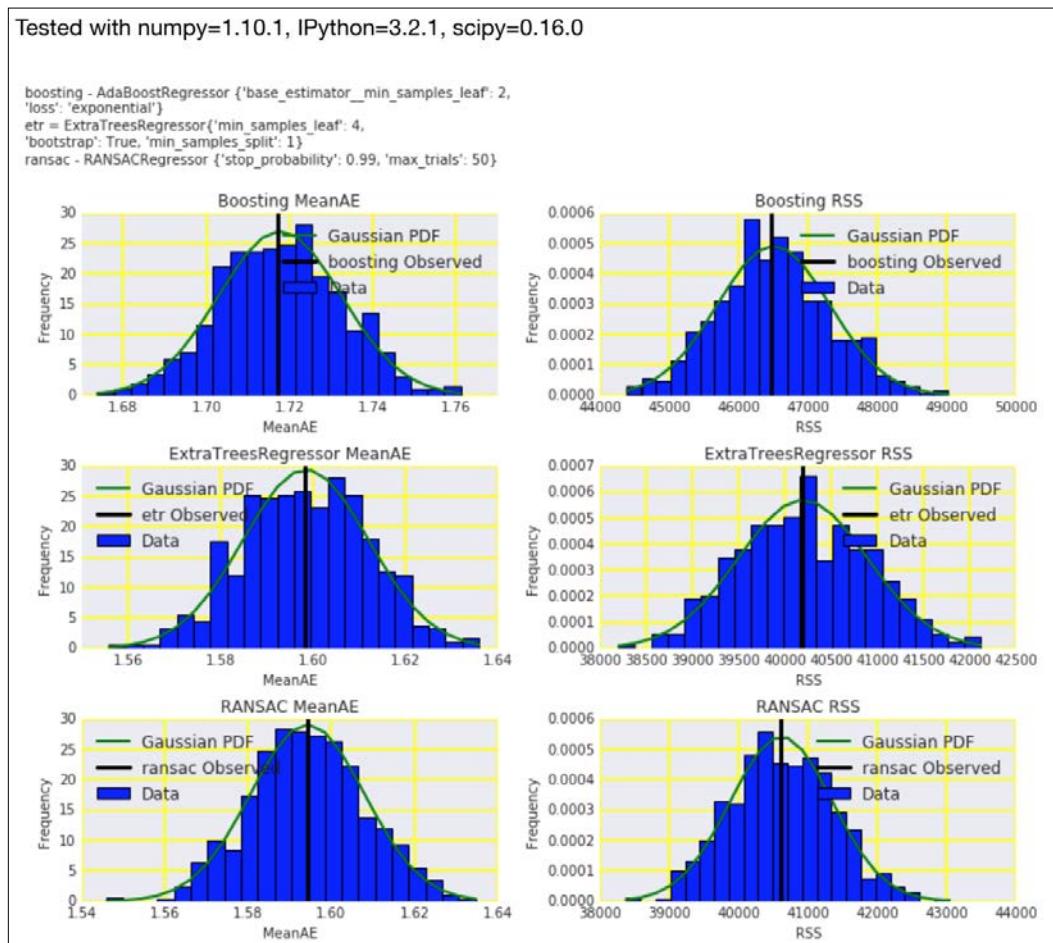
sp.label()

ch10util.plot_bootstrap('ransac',
                        metrics.mean_absolute_error, sp.next_ax())
sp.label()

ch10util.plot_bootstrap('ransac',
                        dl.stats.rss, sp.next_ax())
sp.label()
sp.fig.text(0, 1, ch10util.regressors())
HTML(sp.exit())

```

Refer to the following screenshot for the end result:



The code is in the `mae_rss.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about the MeanAE at https://en.wikipedia.org/wiki/Mean_absolute_error (retrieved November 2015)
- ▶ The `mean_absolute_error()` function documented at http://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_absolute_error.html (retrieved November 2015)
- ▶ The Wikipedia page about the RSS at https://en.wikipedia.org/wiki/Residual_sum_of_squares (retrieved November 2015)

Examining the kappa of classification

Cohen's kappa measures the agreement between target and predicted class similar to accuracy, but it also takes into account random chance of getting the predictions. Cohen's kappa is given by the following equation:

$$(10.13) \quad K = \frac{p_o - p_e}{1 - p_e} = 1 - \frac{1 - p_o}{1 - p_e}$$

In this equation, p_o is the relative observed agreement and p_e is the random chance of agreement derived from the data. Kappa varies between negative values and one with the following rough categorization from Landis and Koch:

- ▶ Poor agreement: $\text{kappa} < 0$
- ▶ Slight agreement: $\text{kappa} = 0$ to 0.2
- ▶ Fair agreement: $\text{kappa} = 0.21$ to 0.4
- ▶ Moderate agreement: $\text{kappa} = 0.41$ to 0.6
- ▶ Good agreement: $\text{kappa} = 0.61$ to 0.8
- ▶ Very good agreement: $\text{kappa} = 0.81$ to 1.0

I know of two other schemes to grade kappa, so these numbers are not set in stone. I think we can agree not to accept kappa less than 0.2. The most appropriate use case is, of course, to rank models. There are other variations of Cohen's kappa, but as of November 2015, they were not implemented in scikit-learn. scikit-learn 0.17 has added support for Cohen's kappa via the `cohen_kappa_score()` function.

How to do it...

- The imports are as follows:

```
import dutil as dl
from sklearn import metrics
import numpy as np
import ch10util
from IPython.display import HTML
```

- Compute accuracy, precision, recall, F1-score, and kappa for the rain predictors:

```
y_test = np.load('rain_y_test.npy')
accuracies = [metrics.accuracy_score(y_test, preds)
              for preds in ch10util.rain_preds()]
precisions = [metrics.precision_score(y_test, preds)
               for preds in ch10util.rain_preds()]
recalls = [metrics.recall_score(y_test, preds)
            for preds in ch10util.rain_preds()]
f1s = [metrics.f1_score(y_test, preds)
       for preds in ch10util.rain_preds()]
kappas = [metrics.cohen_kappa_score(y_test, preds)
           for preds in ch10util.rain_preds()]
```

- Scatter plot the metrics against kappa as follows:

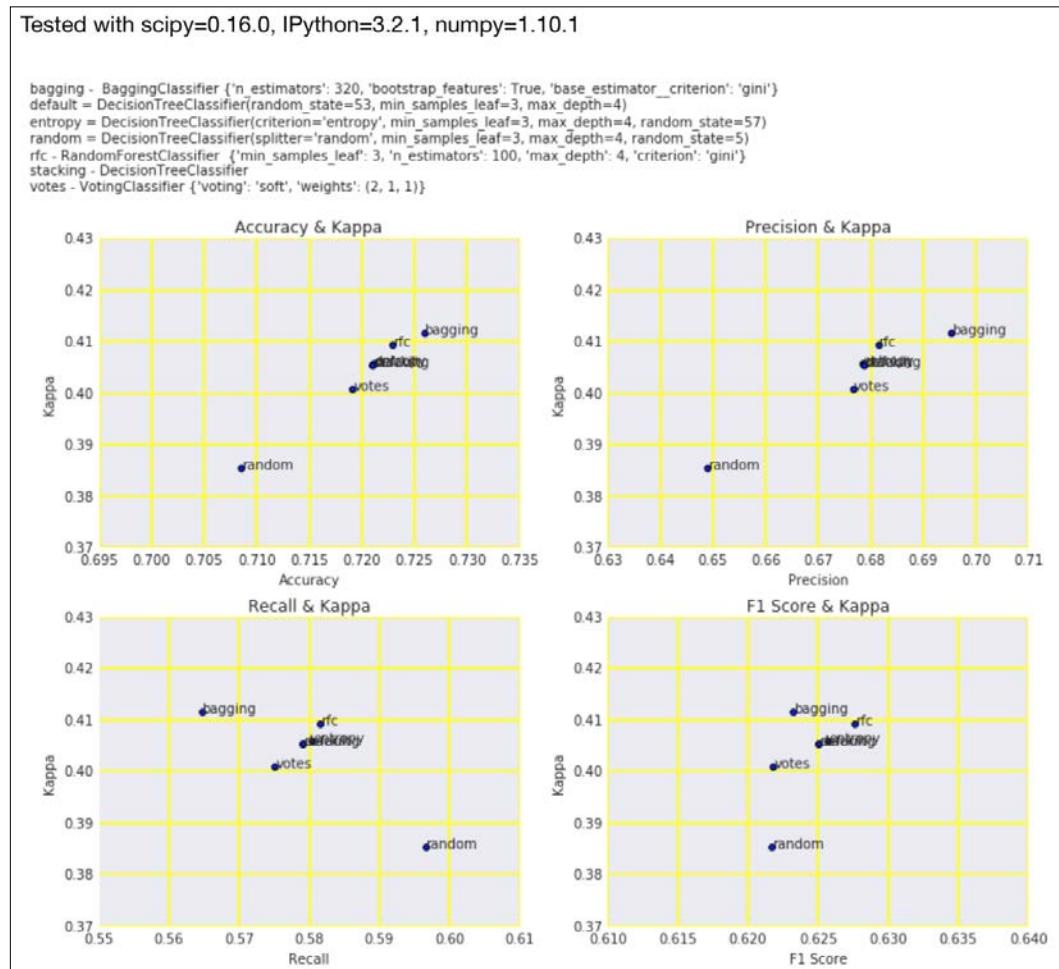
```
sp = dl.plotting.Subplotter(2, 2, context)
dl.plotting.plot_text(sp.ax, accuracies, kappas,
                      ch10util.rain_labels(), add_scatter=True)
sp.label()

dl.plotting.plot_text(sp.next_ax(), precisions, kappas,
                      ch10util.rain_labels(), add_scatter=True)
sp.label()

dl.plotting.plot_text(sp.next_ax(), recalls, kappas,
                      ch10util.rain_labels(), add_scatter=True)
sp.label()

dl.plotting.plot_text(sp.next_ax(), f1s, kappas,
                      ch10util.rain_labels(), add_scatter=True)
sp.label()
sp.fig.text(0, 1, ch10util.classifiers())
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code is in the `kappa.ipynb` file in this book's code bundle.

How it works

From the first two plots, we can conclude that the bagging classifier has the highest accuracy, precision, and kappa. All the classifiers have a kappa above 0.2, so they are at least somewhat acceptable.

See also

- ▶ The Wikipedia page about Cohen's kappa at https://en.wikipedia.org/wiki/Cohen%27s_kappa (retrieved November 2015)

Taking a look at the Matthews correlation coefficient

The **Matthews correlation coefficient (MCC)** or **phi coefficient** is an evaluation metric for binary classification invented by Brian Matthews in 1975. The MCC is a correlation coefficient for target and predictions and varies between -1 and 1 (best agreement). MCC is a very good way to summarize the confusion matrix (refer to the *Getting classification straight with the confusion matrix recipe*) as it uses all four numbers in it. The MCC is given by the following equation:

$$(10.14) \quad MCC = \frac{T_p \times T_n - F_p \times F_n}{\sqrt{(T_p + F_p)(T_p + F_n)(T_n + F_p)(T_n + F_n)}}$$

How to do it...

1. The imports are as follows:

```
import dutil as dl
from sklearn import metrics
import numpy as np
import ch10util
from IPython.display import HTML
```

2. Calculate accuracies, precisions, recalls, F1-scores, and Matthews correlation coefficients for the rain predictors:

```
y_test = np.load('rain_y_test.npy')
accuracies = [metrics.accuracy_score(y_test, preds)
              for preds in ch10util.rain_preds()]
precisions = [metrics.precision_score(y_test, preds)
               for preds in ch10util.rain_preds()]
recalls = [metrics.recall_score(y_test, preds)
            for preds in ch10util.rain_preds()]
f1s = [metrics.f1_score(y_test, preds)
       for preds in ch10util.rain_preds()]
mc = [metrics.matthews_corrcoef(y_test, preds)
      for preds in ch10util.rain_preds()]
```

3. Plot the metrics as follows:

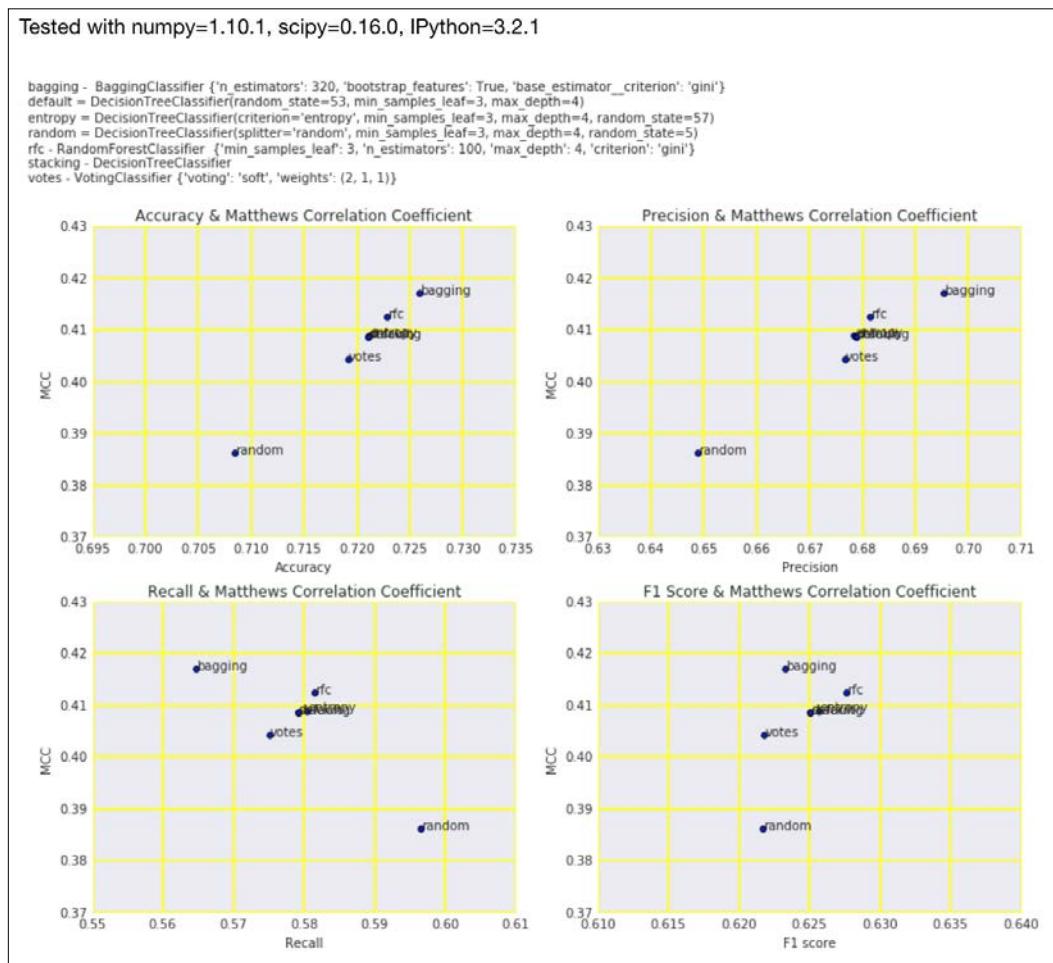
```
sp = dl.plotting.Subplotter(2, 2, context)
dl.plotting.plot_text(sp.ax, accuracies, mc,
                      ch10util.rain_labels(), add_scatter=True)
sp.label()

dl.plotting.plot_text(sp.next_ax(), precisions, mc,
                      ch10util.rain_labels(), add_scatter=True)
sp.label()

dl.plotting.plot_text(sp.next_ax(), recalls, mc,
                      ch10util.rain_labels(), add_scatter=True)
sp.label()

dl.plotting.plot_text(sp.next_ax(), f1s, mc,
                      ch10util.rain_labels(), add_scatter=True)
sp.label()
sp.fig.text(0, 1, ch10util.classifiers())
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code is in the `matthews_correlation.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about the MCC at https://en.wikipedia.org/wiki/Matthews_correlation_coefficient (retrieved November 2015)
- ▶ The `matthews_corrcoef()` function documented at http://scikit-learn.org/stable/modules/generated/sklearn.metrics.matthews_corrcoef.html (retrieved November 2015)

11

Analyzing Images

In this chapter, we will cover the following recipes:

- ▶ Setting up OpenCV
- ▶ Applying Scale-Invariant Feature Transform (SIFT)
- ▶ Detecting features with SURF
- ▶ Quantizing colors
- ▶ Denoising images
- ▶ Extracting patches from an image
- ▶ Detecting faces with Haar cascades
- ▶ Searching for bright stars
- ▶ Extracting metadata from images
- ▶ Extracting texture features from images
- ▶ Applying hierarchical clustering on images
- ▶ Segmenting images with spectral clustering

Introduction

Image processing is a very large field of study. The techniques used for image processing can often (with small changes) be applied to video analysis as well. We can view image processing as a special type of signal processing. Signal processing is covered in *Chapter 6, Signal Processing and Timeseries*. However, images pose special challenges, such as high dimensionality (we can define each image pixel to be a feature) and spatial dependence (pixel location matters).

The human visual system is very advanced compared to what computers can do. We are able to recognize objects, facial expressions, and object motion. Apparently, this has to do with predators and with their tendency to consume human flesh. Instead of trying to understand how human vision works, we will concentrate on finding features in images and clustering image pixels (segmenting) in this chapter.

In this chapter, we use the OpenCV library quite a lot, and since it is a fairly large library, I decided to create a special Docker container for this chapter only. As you probably know already, I made a Docker image called `pydabck`. Well, the Docker container for this chapter is named `pydabck11`.

Setting up OpenCV

OpenCV (Open Source Computer Vision) is a library for computer vision created in 2000, and is currently maintained by Itseez. OpenCV is written in C++, but it also has bindings for Python and other programming languages. OpenCV supports many operating systems and GPUs. There is not enough space in this chapter to cover all the features of OpenCV. Even a single book is probably not enough—for Pythonistas, I recommend *OpenCV Computer Vision with Python* by Joseph Howse.

Some of the third-party patented algorithms in the OpenCV 2.x.x package, such as SIFT and SURF (refer to the relevant recipes in this chapter), have been moved to a special GitHub repository. You still can use them, but you need to explicitly include them in the installation process.

The OpenCV build process has many options. If you are unsure which options are the best for you, read the OpenCV documentation or use the appropriate package manager for your operating system. In general, you should not use too many options. Although you have the flexibility to turn off certain modules, other modules may depend on them, which could lead to a cascade of errors.

Getting ready

If you are on Windows or Fedora, read the corresponding tutorials at http://docs.opencv.org/3.0.0/da/df6/tutorial_py_table_of_contents_setup.html (retrieved December 2015). For the Ubuntu-based Docker container, I needed to install some prerequisites with the following commands:

```
$ apt-get update  
$ apt-get install -y cmake make git g++
```

How to do it...

The following instructions serve as an example and make some assumptions about your setup. For instance, it assumes that you are using Anaconda with Python 3. For convenience, I organized all the instructions in a single shell script for the Ubuntu-based Docker container; however, if you prefer, you can also type each line separately in a terminal.

1. Download the code of the core OpenCV project (if you don't have Git, you can also download the code from the GitHub website):

```
$ cd /opt  
$ git clone https://github.com/Itseez/opencv.git  
$ cd opencv  
$ git checkout tags/3.0.0
```

2. Download the code of the (third-party) contributions to OpenCV (if you don't have Git, you can also download the code from the GitHub website):

```
$ cd /opt  
$ git clone https://github.com/Itseez/opencv_contrib  
$ cd opencv_contrib  
$ git checkout tags/3.0.0  
$ cd /opt/opencv
```

3. Make a build directory and navigate to it:

```
$ mkdir build  
$ cd build
```

4. This step shows some of the build options available to you (you don't have to use all these options):

```
$ ANACONDA=~/anaconda  
$ cmake -D CMAKE_BUILD_TYPE=RELEASE \  
       -D BUILD_PERF_TESTS=OFF \  
       -D BUILD_opencv_core=ON \  
       -D BUILD_opencv_python2=OFF \  
       -D BUILD_opencv_python3=ON \  
       -D BUILD_opencv_cuda=OFF \  
       -D BUILD_opencv_java=OFF \  
       -D BUILD_opencv_video=ON \  
       -D BUILD_opencv_videoio=ON \  
       -D BUILD_opencv_world=OFF \  
       -D BUILD_opencv_viz=ON \  
       -D WITH_CUBLAS=OFF \  
       -D CMAKE_CXX_STANDARD=11
```

```

-D WITH_CUDA=OFF \
-D WITH_CUFFT=OFF \
-D WITH_FFMPEG=OFF \
-D PYTHON3_EXECUTABLE=${ANACONDA}/bin/python3 \
-D PYTHON3_LIBRARY=${ANACONDA}/lib/libpython3.4m.so \
-D PYTHON3_INCLUDE_DIR=${ANACONDA}/include/python3.4m \
-D PYTHON3_NUMPY_INCLUDE_DIRS=${ANACONDA}/lib/python3.4/site-
packages/numpy/core/include \
-D PYTHON3_PACKAGES_PATH=${ANACONDA}/lib/python3.4/site-
packages \
-D BUILD_opencv_latentsvm=OFF \
-D BUILD_opencv_xphoto=OFF \
-D BUILD_opencv_xfeatures2d=ON \
-D OPENCV_EXTRA_MODULES_PATH=/opt/opencv_contrib/modules \
/opt/opencv

```

5. Run the make command (with 8 cores) and install as follows:

```

$ make -j8
$ sudo make install

```

How it works

The previous instructions assumed that you are installing OpenCV for the first time. If you are upgrading from OpenCV 2.x.x, you will have to take extra precautions. Also, I assumed that you don't want certain options and are using Anaconda with Python 3. The following table explains some of the build options we used:

Option	Description
BUILD_opencv_python2	Support for Python 2
BUILD_opencv_python3	Support for Python 3
BUILD_opencv_java	Support for the OpenCV Java bindings
PYTHON3_EXECUTABLE	The location of the Python 3 executable
PYTHON3_LIBRARY	The location of the Python 3 library
PYTHON3_INCLUDE_DIR	The location of the Python 3 include directory
PYTHON3_NUMPY_INCLUDE_DIRS	The location of the NumPy include directories
PYTHON3_PACKAGES_PATH	The location of the Python 3 packages
BUILD_opencv_xfeatures2d	Support for certain third-party algorithms, such as SIFT and SURF
OPENCV_EXTRA_MODULES_PATH	The location of the code of the OpenCV third-party contributions

There's more

If you don't have enough space, for instance in a Docker container, then you can clean up with the following commands:

```
$ rm -rf /opt/opencv
$ rm -rf /opt/opencv_contrib
```

Applying Scale-Invariant Feature Transform (SIFT)

The SIFT algorithm (1999) finds features in images or videos and is patented by the University of British Columbia. Typically, we can use the features for classification or clustering. SIFT is invariant with respect to translation, scaling, and rotation.

The algorithm's steps are as follows:

1. Blur the image at different scales using a Gaussian blur filter.
2. An **octave** corresponds to doubling the standard deviation of the filter. Group the blurred images by octave and difference them.
3. Find the local extrema across the scale for the differenced images.
4. Compare each pixel related to local extrema to the neighboring pixels in the same scale and neighboring scales.
5. Select the largest or smallest value from the comparison.
6. Reject points with low contrast.
7. Interpolate candidate key points (image features) to get the position on the original image.

Getting ready

Follow the instructions in the *Setting up OpenCV* recipe.

How to do it...

1. The imports are as follows:

```
import cv2
import matplotlib.pyplot as plt
import dutil as dl
from scipy.misc import face
```

2. Plot the original image as follows:

```
img = face()
plt.title('Original')
dl.plotting.img_show(plt.gca(), img)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

3. Plot the grayscaled image as follows:

```
plt.figure()
plt.title('Gray')
dl.plotting.img_show(plt.gca(), gray, cmap=plt.cm.gray)
```

4. Plot the image with keypoints (blue) as follows:

```
sift = cv2.xfeatures2d.SIFT_create()
(kps, descs) = sift.detectAndCompute(gray, None)
img2 = cv2.drawKeypoints(gray, kps, None, (0, 0, 255))

plt.figure()
plt.title('With Keypoints')
dl.plotting.img_show(plt.gca(), img2)
```

Refer to the following screenshot for the end result:



The program is in the `applying_sift.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about SIFT at https://en.wikipedia.org/wiki/Scale-invariant_feature_transform (retrieved December 2015)
- ▶ The SIFT algorithm documented at http://docs.opencv.org/3.0.0/da/df5/tutorial_py_sift_intro.html (retrieved December 2015)

Detecting features with SURF

Speeded Up Robust Features (SURF) is a patented algorithm similar to and inspired by SIFT (refer to the *Applying Scale-Invariant Feature Transform* recipe). SURF was introduced in 2006 and uses Haar wavelets (refer to the *Applying the discrete wavelet transform* recipe). The greatest advantage of SURF is that it is faster than SIFT.

Take a look at the following equations:

$$(11.1) \quad S(x, y) = \sum_{i=0}^x \sum_{j=0}^y I(i, j)$$

$$(11.2) \quad H(p, \sigma) = \begin{pmatrix} L_{xx}(p, \sigma) & L_{xy}(p, \sigma) \\ L_{xy}(p, \sigma) & L_{yy}(p, \sigma) \end{pmatrix}$$

$$(11.3) \quad \sigma_{approx} = \text{Current filter size} * \left(\frac{\text{Base Filter Scale}}{\text{Base Filter Size}} \right)$$

The algorithm steps are as follows:

1. Transform the image if necessary to get the grayscale equivalent.
2. Calculate the **integral image** at different scales, which is the sum of the pixels above and to the left of a pixel, as shown in equation (11.1). The integral image replaces the Gaussian filter in SIFT.
3. Define the **Hessian matrix** (11.2) containing second-order derivatives of the grayscale image as function of pixel location p and scale σ (11.3).
4. **Determinants** are values related to square matrices. The determinant of the Hessian matrix corresponds to a local change in a point. Select points with the largest determinant.
5. The scale σ is defined by 11.3, and just as with SIFT, we can define scale octaves. SURF works by varying the size of the filter kernel, while SIFT varies the image size. Interpolate the maximums from the previous step in the scale and image space.

-
6. Apply the Haar wavelet transform to a circle around key points.
 7. Use a sliding window to sum responses.
 8. Determine orientation from the response sums.

Getting ready

Follow the instructions in the *Setting up OpenCV* recipe.

How to do it...

1. The imports are as follows:

```
import cv2
import matplotlib.pyplot as plt
import dautil as dl
```

2. Plot the original image as follows:

```
img = cv2.imread('covers.jpg')
plt.title('Original')
dl.plotting.img_show(plt.gca(), img)
```

3. Plot the grayscaled image as follows:

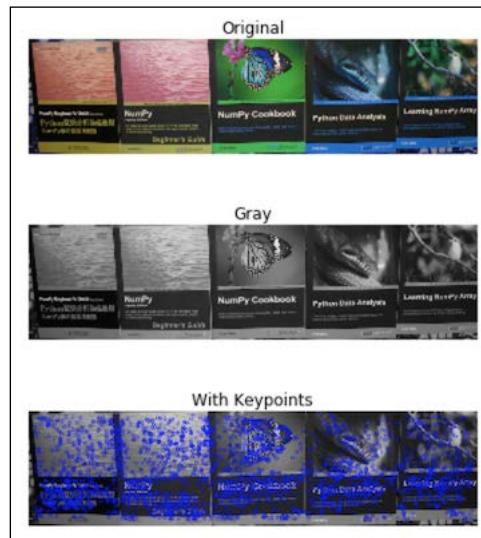
```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

plt.figure()
plt.title('Gray')
dl.plotting.img_show(plt.gca(), gray, cmap=plt.cm.gray)
surf = cv2.xfeatures2d.SURF_create()
(kps, desc) = surf.detectAndCompute(gray, None)
img2 = cv2.drawKeypoints(gray, kps, None, (0, 0, 255))
```

4. Plot the image with keypoints (blue) as follows:

```
plt.figure()
plt.title('With Keypoints')
dl.plotting.img_show(plt.gca(), img2)
```

Refer to the following screenshot for the end result:



The code is in the `applying_surf.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about SURF at https://en.wikipedia.org/wiki/Speeded_up_robust_features (retrieved December 2015)
- ▶ The Wikipedia page about the integral image at https://en.wikipedia.org/wiki/Summed_area_table (retrieved December 2015)
- ▶ The Wikipedia page about the Hessian matrix at https://en.wikipedia.org/wiki/Hessian_matrix (retrieved December 2015)
- ▶ The Wikipedia page about the determinant at <https://en.wikipedia.org/wiki/Determinant> (retrieved December 2015)
- ▶ The SURF algorithm documented at http://docs.opencv.org/3.0.0/d7/dd2/tutorial_py_surf_intro.html (retrieved December 2015)

Quantizing colors

In ancient times, computer games were practically monochromatic. Many years later, the Internet allowed us to download images, but the Web was slow, so compact images with few colors were preferred. We can conclude that restricting the number of colors is traditional. Color is a dimension of images, so we can speak of dimensionality reduction if we remove colors from an image. The actual process is called **color quantization**.

Usually, we represent **RGB** (**red**, **green**, and **blue**) values in three-dimensional space for each pixel and then cluster the points. For each cluster, we are left with a corresponding average color. In this recipe, we will use k-means clustering (refer to the *Clustering streaming data with Spark* recipe), although this is not necessarily the best algorithm.

Getting ready

Follow the instructions in the *Setting up OpenCV* recipe.

How to do it...

The code is in the `quantizing_colors.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import dautil as dl
from scipy.misc import face
```

2. Plot the original image as follows:

```
sp = dl.plotting.Subplotter(2, 2, context)
img = face()
dl.plotting.img_show(sp.ax, img)
sp.label()
Z = img.reshape((-1, 3))

Z = np.float32(Z)
```

3. Apply k-means clustering and plot the result:

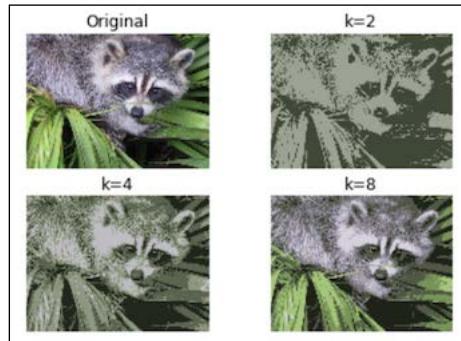
```
criteria = (cv2.TERM_CRITERIA_MAX_ITER, 7, 1.0)

for k in [2, 4, 8]:
    _, label, center = cv2.kmeans(Z, k, None, criteria, 7,
                                    cv2.KMEANS_RANDOM_CENTERS)

    center = np.uint8(center)
    res = center[label.flatten()]
    res2 = res.reshape((img.shape))

    dl.plotting.img_show(sp.next_ax(), res2)
    sp.label()
```

Refer to the following screenshot for the end result:



See also

- ▶ The Wikipedia page about color quantization at https://en.wikipedia.org/wiki/Color_quantization (retrieved December 2015)
- ▶ The `kmeans()` function documented at http://docs.opencv.org/3.0.0/d5/d38/group__core__cluster.html#ga9a34dc06c6ec9460e90860f15bcd2f88 (retrieved December 2015)

Denoising images

Noise is a common phenomenon in data and also in images. Of course, noise is undesirable, as it does not add any value to our analysis. We typically assume that noise is normally distributed around zero. We consider a pixel value to be the sum of the true value and noise (if any). We also assume that the noise values are independent, that is, the noise value of one pixel is independent of another pixel.

One simple idea is to average pixels in a small window, since we suppose the expected value of noise to be zero. This is the general idea behind blurring. We can take this idea a step further and define multiple windows around a pixel, and we can then average similar patches.

OpenCV has several denoising functions and usually we need to specify the strength of the filter, the size of the search window, and the size of the template window for similarity checks. You should be careful not to set the filter strength too high because that may make the image not only cleaner, but also a bit blurred.

Getting ready

Follow the instructions in the *Setting up OpenCV* recipe.

How to do it...

1. The imports are as follows:

```
import cv2
import matplotlib.pyplot as plt
from sklearn.datasets import load_sample_image
import numpy as np
import dautil as dl
```

2. Plot the original image as follows:

```
img = load_sample_image('china.jpg')
dl.plotting.img_show(plt.gca(), img)
plt.title('Original')
Z = img.reshape((-1, 3))
```

3. Add noise to the image and plot the noisy image:

```
np.random.seed(59)
noise = np.random.random(Z.shape) < 0.99

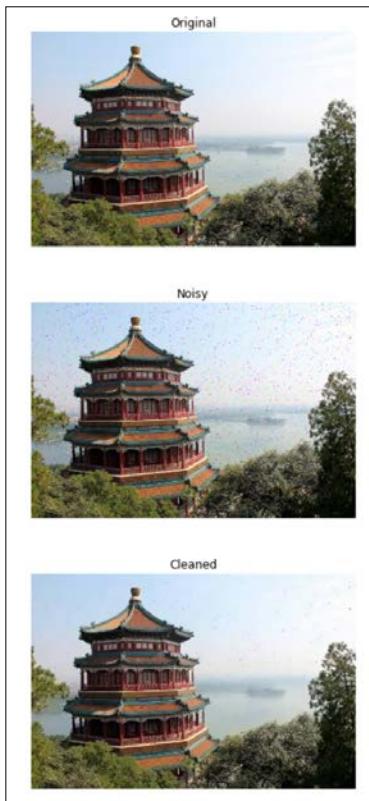
noisy = (Z * noise).reshape((img.shape))

plt.figure()
plt.title('Noisy')
dl.plotting.img_show(plt.gca(), noisy)
```

4. Clean the image and display it:

```
cleaned = cv2.fastNlMeansDenoisingColored(noisy, None, 10, 10, 7,
21)
plt.figure()
plt.title('Cleaned')
dl.plotting.img_show(plt.gca(), cleaned)
```

Refer to the following screenshot for the end result:



The code is in the denoising_images.ipynb file in this book's code bundle.

See also

- ▶ The `fastNlMeansDenoisingColored()` function documented at http://docs.opencv.org/3.0.0/d1/d79/group_photo_denoise.html#ga21abc1c8b0e15f78cd3eff672cb6c476 (retrieved December 2015)

Extracting patches from an image

Image segmentation is a procedure that splits an image into multiple segments. The segments have similar color or intensity. The segments also usually have a meaning in the context of medicine, traffic, astronomy, or something else.

The easiest way to segment images is with a threshold value, which produces two segments (if values are equal to the threshold, we put them in one of the two segments). **Otsu's thresholding** method minimizes the weighted variance of the two segments (refer to the following equation):

$$(11.4) \quad \sigma_w^2(t) = \omega_w^2(t) \sigma_1^2(t) + \omega_2(t) \sigma_2^2(t)$$

If we segment images, it is a good idea to remove noise or foreign artifacts. With **dilation** (see the See also section) we can find parts of the image that belong to the background and the foreground. However, dilation leaves us with unidentified pixels.

Getting ready

Follow the instructions in *Setting up OpenCV*.

How to do it...

1. The imports are as follows:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
from sklearn.datasets import load_sample_image
import dautil as dl
from IPython.display import HTML
```

2. Plot the original image as follows:

```
sp = dl.plotting.Subplotter(2, 2, context)
img = load_sample_image('flower.jpg')
dl.plotting.img_show(sp.ax, img)
sp.label()
```

3. Plot the Otsu threshold image as follows:

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
_, thresh = cv2.threshold(gray, 0, 255,
                         cv2.THRESH_OTSU)

dl.plotting.img_show(sp.next_ax(), thresh)
sp.label()
```

4. Plot the image with foreground and background distracted as follows:

```

kernel = np.ones((3, 3), np.uint8)
opening = cv2.morphologyEx(thresh, cv2.MORPH_OPEN,
                           kernel, iterations=2)

bg = cv2.dilate(opening, kernel, iterations=3)

dist_transform = cv2.distanceTransform(opening, cv2.DIST_L2, 5)
_, fg = cv2.threshold(dist_transform, 0.7 * dist_transform.max(),
                      255, 0)

fg = np.uint8(fg)
rest = cv2.subtract(bg, fg)

dl.plotting.img_show(sp.next_ax(), rest)
sp.label()

5. Plot the image with markers as follows:

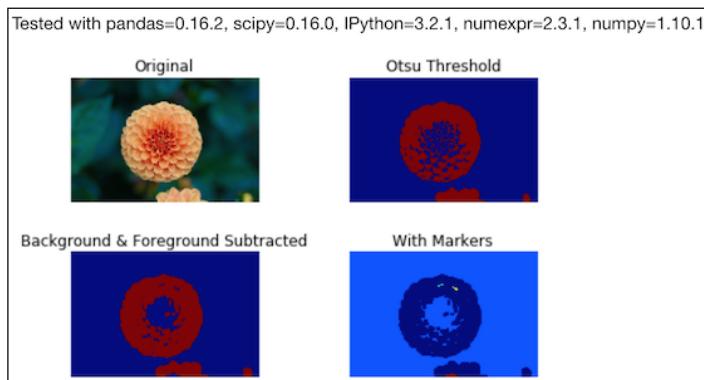
_, markers = cv2.connectedComponents(fg)
markers += 1
markers[rest == 255] = 0

dl.plotting.img_show(sp.next_ax(), markers)
sp.label()

HTML(sp.exit())

```

Refer to the following screenshot for the end result:



The code is in the `extracting_patches.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about image segmentation at https://en.wikipedia.org/wiki/Image_segmentation (retrieved December 2015)
- ▶ The Wikipedia page about Otsu's method at https://en.wikipedia.org/wiki/Otsu%27s_method (retrieved December 2015)
- ▶ The Wikipedia page about dilation at https://en.wikipedia.org/wiki/Dilation_%28morphology%29 (retrieved December 2015)

Detecting faces with Haar cascades

Faces are an identifying feature of human anatomy. Strictly speaking, many animals also have faces, but that is less relevant for most practical applications. **Face detection** tries to find (rectangular) areas in an image that represent faces. Face detection is a type of **object detection**, because faces are a type of object.

Most face detection algorithms are good at detecting clean front-facing faces because most training images fall in that category. Tilted faces, bright lights, or noisy images may cause problems for face detection. It is possible to deduce age, gender, or ethnicity (for instance, the presence of epicanthic folds) from a face, which of course is useful for marketing.

A possible application could be analyzing profile pictures on social media sites. OpenCV uses a **Haar feature-based cascade classifiers system** to detect faces. The system is also named the **Viola-Jones object detection framework** after its inventors who proposed it in 2001.

The algorithm has the following steps:

1. Haar feature selection: Haar features are similar to Haar wavelets (as covered in the *Applying the discrete wavelet transform* recipe in *Chapter 6, Signal Processing and Timeseries*).
2. Creating an integral image (refer to the *Detecting features with SURF* recipe).
3. Adaboost training (refer to the *Boosting for better learning* recipe in *Chapter 9, Ensemble Learning and Dimensionality Reduction*).
4. Cascading classifiers.

When we look at face images, we can create heuristics related to brightness.

For instance, the nose region is brighter than regions directly to its left and right. Therefore, we can define a white rectangle covering the nose and black rectangles covering the neighboring areas. Of course the Viola-Jones system doesn't know exactly where the nose is, but by defining windows of varying size and seeking corresponding white and black rectangles, there is a chance of matching a nose. The actual Haar features are defined as the sum of brightness in a black rectangle and the sum of brightness in a neighboring rectangle. For a 24 x 24 window, we have more than 160 thousand features (roughly 24 to the fourth power).

The training set consists of a huge collection of positive (with faces) images and negative (no faces) images. Only about 0.01% of the windows (in the order of 24 by 24 pixels) actually contain faces. The cascade of classifiers progressively filters out negative image areas stage by stage. In each progressive stage, the classifiers use progressively more features on less image windows. The idea is to spend the most time on image patches that contain faces. The original paper by Viola and Jones had 38 stages with 1, 10, 25, 25, and 50 features in the first five stages. On average, 10 features per image window were evaluated.

In OpenCV, you can train a cascade classifier yourself, as described in http://docs.opencv.org/3.0.0/dc/d88/tutorial_traincascade.html (retrieved December 2015). However, OpenCV has pre-trained classifiers for faces, eyes, and other features. The configuration for these classifiers is stored as XML files, which can be found in the folder where you installed OpenCV (on my machine, /usr/local/share/OpenCV/haarcascades/).

Getting ready

Follow the instructions in *Setting up OpenCV*.

How to do it...

1. The imports are as follows:

```
import cv2
from scipy.misc import lena
import matplotlib.pyplot as plt
import numpy as np
import dautil as dl
import os
from IPython.display import HTML
```

2. Define the following function to plot the image with a detected face (if detected):

```
def plot_with_rect(ax, img):
    img2 = img.copy()

    for x, y, w, h in face_cascade.detectMultiScale(img2, 1.3, 5):
        cv2.rectangle(img2, (x, y), (x + w, y + h), (255, 0, 0),
2)

    dl.plotting.img_show(ax, img2, cmap=plt.cm.gray)
```

-
3. Download the XML configuration file and create a classifier:

```
# dir = '/usr/local/share/opencv/haarcascades/'  
base = 'https://raw.githubusercontent.com/Itseez/opencv/master/  
data/'  
url = base + 'haarcascades/haarcascade_frontalface_default.xml'  
path = os.path.join(dl.data.get_data_dir(),  
                    'haarcascade_frontalface_default.xml')  
  
if not dl.conf.file_exists(path):  
    dl.data.download(url, path)  
  
face_cascade = cv2.CascadeClassifier(path)
```

4. Plot the original image with a detected face:

```
sp = dl.plotting.Subplotter(2, 2, context)  
img = lena().astype(np.uint8)  
plot_with_rect(sp.ax, img)  
sp.label()
```

5. Plot the slightly rotated image (detection fails):

```
rows, cols = img.shape  
mat = cv2.getRotationMatrix2D((cols/2, rows/2), 21, 1)  
rot = cv2.warpAffine(img, mat, (cols, rows))  
plot_with_rect(sp.next_ax(), rot)  
sp.label()
```

6. Plot the image with noise added (detection fails):

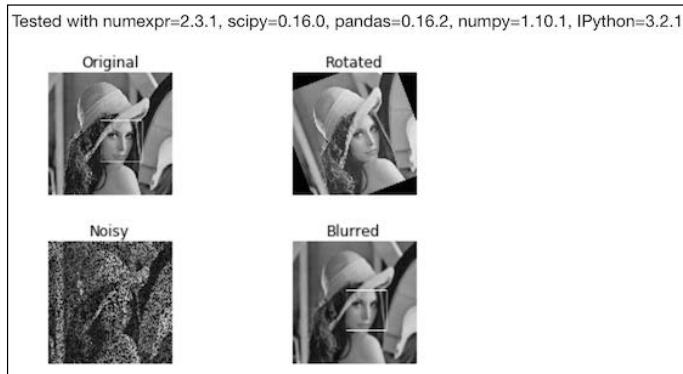
```
np.random.seed(36)  
noisy = img * (np.random.random(img.shape) < 0.6)  
plot_with_rect(sp.next_ax(), noisy)  
sp.label()
```

7. Plot the blurred image with a detected face:

```
blur = cv2.blur(img, (9, 9))  
plot_with_rect(sp.next_ax(), blur)  
sp.label()
```

```
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code is in the `detecting_faces.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about face detection at https://en.wikipedia.org/wiki/Face_detection (retrieved December 2015)
- ▶ The Wikipedia page about the Viola-Jones framework at https://en.wikipedia.org/wiki/Viola%20%26%20Jones_object_detection_framework (retrieved December 2015)

Searching for bright stars

Many stars are visible at night, even without using a telescope or any other optical device. Stars are, in general, larger than planet Earth, but in certain stages of their evolution, they can be smaller. Due to the large distance, they appear as tiny dots. Often, these dots consist of two (a binary system) or more stars. Not all stars emit visible light and not all starlight can reach us.

There are many approaches that we can take to find bright stars in a starry sky image. In this recipe, we will look for local maximums of brightness, which are also above a threshold. To determine brightness, we will convert the image to the HSV color space. In this color space, the three dimensions are hue, saturation, and value (brightness). The OpenCV `split()` function image values in a color space into the constituent values, for example, hue, saturation, and brightness. This is a relatively slow operation. To find maximums, we can apply the SciPy `argrelmax()` function.

Getting ready

Follow the instructions in the *Setting up OpenCV* recipe.

How to do it...

1. The imports are as follows:

```
import dautil as dl
import os
import cv2
import matplotlib.pyplot as plt
from scipy.signal import argrelmax
import numpy as np
from IPython.display import HTML
```

2. Define the following function to scan the horizontal or vertical axis for local brightness peaks:

```
def scan_axis(v, axis):
    argmax = argrelmax(v, order=int(np.sqrt(v.shape[axis])),
                        axis=axis)

    return set([(i[0], i[1]) for i in np.column_stack(argmax)])
```

3. Download the image to analyze:

```
dir = dl.data.get_data_dir()
path = os.path.join(dir, 'night-927168_640.jpg')
base = 'https://pixabay.com/static/uploads/
photo/2015/09/06/10/19/'
url = base + 'night-927168_640.jpg'

if not dl.conf.file_exists(path):
    dl.data.download(url, path)
```

-
4. Extract the brightness values from the image:

```
img = cv2.imread(path)
hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

h, s, v = cv2.split(hsv)

# Transform for normalization
v = v.astype(np.uint16) ** 2
```

5. Plot a histogram of the brightness values:

```
sp = dl.plotting.Subplotter(2, 2, context)
sp.ax.hist(v.ravel(), normed=True)
sp.label()
```

6. Plot a histogram of the brightness values for axis 0:

```
dl.plotting.hist_norm_pdf(sp.next_ax(), v.mean(axis=0))
sp.label()
```

7. Plot a histogram of the brightness values for axis 1:

```
dl.plotting.hist_norm_pdf(sp.next_ax(), v.mean(axis=1))
sp.label()
```

8. Plot the image with points we believe to contain bright stars:

```
points = scan_axis(v, 0).intersection(scan_axis(v, 1))

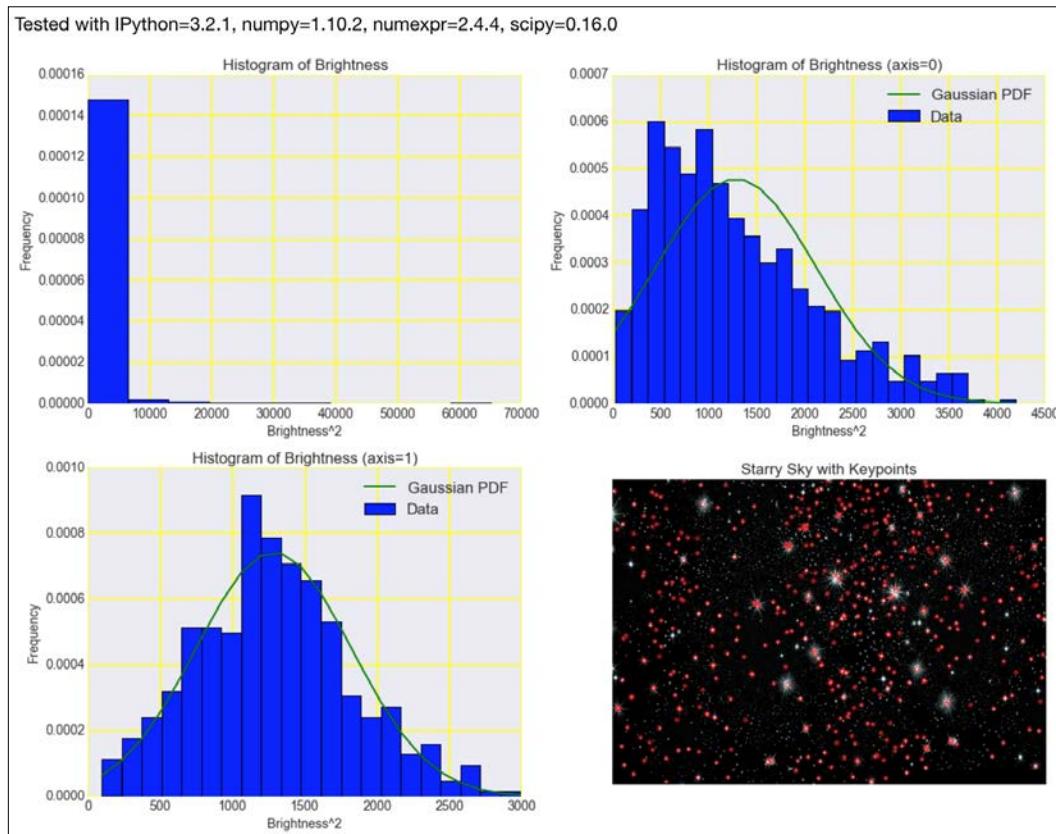
limit = np.percentile(np.unique(v.ravel()), 95)

kp = [cv2.KeyPoint(p[1], p[0], 1) for p in points
      if v[p[0], p[1]] > limit]
with_kp = cv2.drawKeypoints(img, kp, None, (255, 0, 0))

dl.plotting.img_show(sp.next_ax(), with_kp)
sp.label()

HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code is in the `searching_stars.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about HSL and HSV at https://en.wikipedia.org/wiki/HSL_and_HSV (retrieved December 2015)
- ▶ The `argrelmax()` function documented at <https://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.signal.argrelmax.html> (retrieved December 2015)
- ▶ The `split()` function documented at http://docs.opencv.org/3.0-rc1/d2/de8/group__core__array.html#ga0547c7fed86152d7e9d0096029c8518a (retrieved December 2015)

Extracting metadata from images

Digital photos often contain extra textual metadata, for example, timestamps, exposure information, and geolocations. Some of this metadata is editable by the camera owner. In the context of marketing, for instance, it can be useful to extract the metadata from profile (or other) images on social media websites. Purportedly, whistle blower Edward Snowden claimed that the American NSA is collecting EXIF metadata from global online data.

Getting ready

In this recipe, we will use ExifRead to extract the EXIF metadata.

Install ExifRead as follows:

```
$ pip install ExifRead
```

I tested the code with ExifRead 2.1.2.

How to do it...

1. The imports are as follows:

```
import exifread
import pprint
```

2. Open the image as follows:

```
f = open('covers.jpg', 'rb')
```

3. Print the tags and keys as follows:

```
# Return Exif tags
tags = exifread.process_file(f)
print(tags.keys())
pprint.pprint(tags)
f.close()
```

Refer to the following end result:

```
dict_keys(['EXIF Flash', 'Image Make', 'EXIF Contrast',
'EXIF DateTimeOriginal', 'Image ResolutionUnit', 'EXIF
ComponentsConfiguration', 'EXIF ISOSpeedRatings', 'Image
ExifOffset', 'Image ImageDescription', 'EXIF MaxApertureValue',
'EXIF ExposureBiasValue', 'Image YResolution', 'Image
Orientation', 'EXIF DateTimeDigitized', 'EXIF MeteringMode',
'EXIF Sharpness', 'EXIF WhiteBalance', 'EXIF ExposureTime',
'Image Model', 'EXIF SceneCaptureType', 'Image Software', 'EXIF
SceneType', 'EXIF SubjectDistanceRange', 'EXIF LightSource', 'EXIF
```

```
FocalLengthIn35mmFilm', 'Image XResolution', 'Image DateTime',
'EXIF FileSource', 'EXIF ExposureProgram', 'EXIF FocalLength',
'EXIF FNumber', 'EXIF Saturation', 'EXIF ExifImageWidth', 'EXIF
ExposureMode', 'EXIF DigitalZoomRatio', 'EXIF FlashPixVersion',
'EXIF ExifVersion', 'EXIF ColorSpace', 'EXIF CustomRendered', 'EXIF
GainControl', 'EXIF CompressedBitsPerPixel', 'EXIF ExifImageLength'])
{'EXIF ColorSpace': (0xA001) Short=sRGB @ 406,
'EXIF ComponentsConfiguration': (0x9101) Undefined=YCbCr @ 298,
'EXIF CompressedBitsPerPixel': (0x9102) Ratio=2 @ 650,
'EXIF Contrast': (0xA408) Short=Normal @ 550,
'EXIF CustomRendered': (0xA401) Short=Normal @ 466,
'EXIF DateTimeDigitized': (0x9004) ASCII=0000:00:00 00:00:00 @ 630,
'EXIF DateTimeOriginal': (0x9003) ASCII=0000:00:00 00:00:00 @ 610,
'EXIF DigitalZoomRatio': (0xA404) Ratio=0 @ 682,
'EXIF ExifImageLength': (0xA003) Long=240 @ 430,
'EXIF ExifImageWidth': (0xA002) Long=940 @ 418,
'EXIF ExifVersion': (0x9000) Undefined=0220 @ 262,
'EXIF ExposureBiasValue': (0x9204) Signed Ratio=0 @ 658,
'EXIF ExposureMode': (0xA402) Short=Auto Exposure @ 478,
'EXIF ExposureProgram': (0x8822) Short=Program Normal @ 238,
'EXIF ExposureTime': (0x829A) Ratio=10/601 @ 594,
'EXIF FNumber': (0x829D) Ratio=14/5 @ 602,
'EXIF FileSource': (0xA300) Undefined=Digital Camera @ 442,
'EXIF Flash': (0x9209) Short=Flash fired, auto mode @ 370,
'EXIF FlashPixVersion': (0xA000) Undefined=0100 @ 394,
'EXIF FocalLength': (0x920A) Ratio=39/5 @ 674,
'EXIF FocalLengthIn35mmFilm': (0xA405) Short=38 @ 514,
'EXIF GainControl': (0xA407) Short=None @ 538,
'EXIF ISOSpeedRatings': (0x8827) Short=50 @ 250,
'EXIF LightSource': (0x9208) Short=Unknown @ 358,
'EXIF MaxApertureValue': (0x9205) Ratio=3 @ 666,
'EXIF MeteringMode': (0x9207) Short=Pattern @ 346,
'EXIF Saturation': (0xA409) Short=Normal @ 562,
'EXIF SceneCaptureType': (0xA406) Short=Standard @ 526,
'EXIF SceneType': (0xA301) Undefined=Directly Photographed @ 454,
'EXIF Sharpness': (0xA40A) Short=Normal @ 574,
'EXIF SubjectDistanceRange': (0xA40C) Short=0 @ 586,
'EXIF WhiteBalance': (0xA403) Short=Auto @ 490,
'Image DateTime': (0x0132) ASCII=0000:00:00 00:00:00 @ 184,
'Image ExifOffset': (0x8769) Long=204 @ 126,
'Image ImageDescription': (0x010E) ASCII= @ 134,
'Image Make': (0x010F) ASCII=NIKON @ 146,
'Image Model': (0x0110) ASCII=E7900 @ 152,
'Image Orientation': (0x0112) Short=Horizontal (normal) @ 54,
'Image ResolutionUnit': (0x0128) Short=Pixels/Inch @ 90,
```

```
'Image Software': (0x0131) ASCII=E7900v1.1 @ 174,
'Image XResolution': (0x011A) Ratio=300 @ 158,
'Image YResolution': (0x011B) Ratio=300 @ 166}
```

The code is in the `img_metadata.py` file in this book's code bundle.

See also

- ▶ The Wikipedia page about EXIF at https://en.wikipedia.org/wiki/Exchangeable_image_file_format (retrieved December 2015)
- ▶ The documentation for ExifRead at <https://github.com/ianare/exif-py> (retrieved December 2015)

Extracting texture features from images

Texture is the spatial and visual quality of an image. In this recipe, we will take a look at **Haralick texture features**. These features are based on the **co-occurrence matrix** (11.5) defined as follows:

$$(11.5) \quad C_{\Delta x, \Delta y}(i, j) = \sum_{p=1}^n \sum_{q=1}^m \begin{cases} 1, & \text{if } I(p, q) = i \text{ and } I(p + \Delta x, q + \Delta y) = j \\ 0, & \text{Otherwise} \end{cases}$$

$$(11.6) \quad \begin{aligned} \text{Angular 2nd Moment} &= \sum_j \sum_i p[i, j]^2 \\ \text{Contrast} &= \sum_{n=0}^{Ng-1} n^2 \left\{ \sum_{i=1}^{Ng} \sum_{j=1}^{Ng} p[i, j] \right\}, \text{ where } |i - j| = n \\ \text{Correlation} &= \frac{\sum_{i=1}^{Ng} \sum_{j=1}^{Ng} (ij) p[i, j] - \mu_x \mu_y}{\sigma_x \sigma_y} \\ \text{Entropy} &= - \sum_i \sum_j p[i, j] \log(p[i, j]) \end{aligned}$$

In equation 11.5, i and j are intensities, while p and q are positions. The Haralick features are 13 metrics derived from the co-occurrence matrix, some of them given in equation 11.6. For a more complete list, refer to http://murphylab.web.cmu.edu/publications/boland/boland_node26.html (retrieved December 2015).

We will calculate the Haralick features with the mahotas API and apply them to the handwritten digits dataset of scikit-learn.

Getting ready

Install mahotas as follows:

```
$ pip install mahotas
```

I tested the code with mahotas 1.4.0.

How to do it...

1. The imports are as follows:

```
import mahotas as mh
import numpy as np
from sklearn.datasets import load_digits
import matplotlib.pyplot as plt
from tpot import TPOT
from sklearn.cross_validation import train_test_split
import dautil as dl
```

2. Load the scikit-learn digits data as follows:

```
digits = load_digits()
X = digits.data.copy()
```

3. Create Haralick features and add them:

```
for i, img in enumerate(digits.images):
    np.append(X[i], mh.features.haralick(
        img.astype(np.uint8)).ravel())
```

4. Fit and score a model with TPOT (or my fork, as discussed in *Chapter 9, Ensemble Learning and Dimensionality Reduction*):

```
X_train, X_test, y_train, y_test = train_test_split(
    X, digits.target, train_size=0.75)

tpot = TPOT(generations=6, population_size=101,
            random_state=46, verbosity=2)
tpot.fit(X_train, y_train)

print('Score {:.2f}'.format(tpot.score(X_train, y_train, X_test,
y_test)))
```

5. Plot the first original image as follows:

```
dl.plotting.img_show(plt.gca(), digits.images[0])
plt.title('Original Image')
```

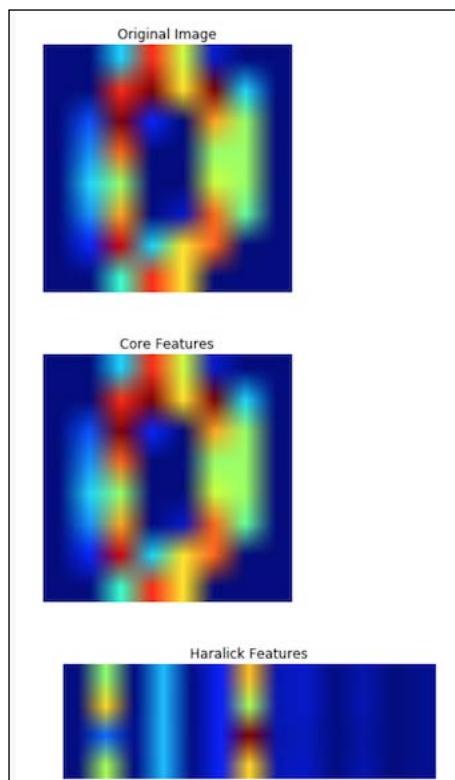
6. Plot the core features for that image:

```
plt.figure()  
dl.plotting.img_show(plt.gca(), digits.data[0].reshape((8, 8)))  
plt.title('Core Features')
```

7. Plot the Haralick features for that image too:

```
plt.figure()  
dl.plotting.img_show(plt.gca(), mh.features.haralick(  
    digits.images[0].astype(np.uint8)))  
plt.title('Haralick Features')
```

Refer to the following screenshot for the end result:



The code is in the `extracting_texture.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about image texture at https://en.wikipedia.org/wiki/Image_texture (retrieved December 2015)
- ▶ The Wikipedia page about the co-occurrence matrix at https://en.wikipedia.org/wiki/Co-occurrence_matrix (retrieved December 2015)

Applying hierarchical clustering on images

We encountered the concept of hierarchical clustering in *Chapter 9, Ensemble Learning and Dimensionality Reduction*. In this recipe, we will segment an image by hierarchically clustering it. We will apply **agglomerative clustering** $O(n^3)$, which is a type of hierarchical clustering.

In agglomerative clustering, each item is assigned its own cluster at initialization. Later, these clusters merge (agglomerate) and move up the hierarchy as needed. Obviously, we only merge clusters that are similar by some measure.

After initialization, we find the pair that are closest by some distance metric and merge them. The merged cluster is a higher-level cluster consisting of lower-level clusters. After that, we again find the closest pair and merge them, and so on. During this process, clusters can have any number of items. We stop clustering after we reach a certain number of clusters, or when the clusters are too far apart.

How to do it...

1. The imports are as follows:

```
import numpy as np
from scipy.misc import ascent
import matplotlib.pyplot as plt
from sklearn.feature_extraction.image import grid_to_graph
from sklearn.cluster import AgglomerativeClustering
import dautil as dl
```

2. Load an image and load it into an array:

```
img = ascent()
X = np.reshape(img, (-1, 1))
```

3. Cluster the image with the number of cluster set to 9 (a guess):

```
connectivity = grid_to_graph(*img.shape)
NCLUSTERS = 9
ac = AgglomerativeClustering(n_clusters=NCLUSTERS,
                             connectivity=connectivity)
```

```

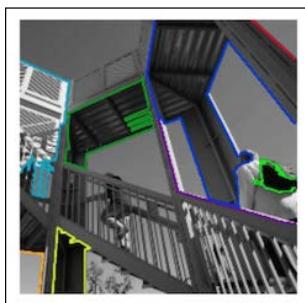
ac.fit(X)
label = np.reshape(ac.labels_, img.shape)

4. Plot the image with cluster segments superimposed:
for l in range(NCLUSTERS):
    plt.contour(label == l, contours=1,
                colors=[plt.cm.spectral(l/float(NCLUSTERS)), ])

dl.plotting.img_show(plt.gca(), img, cmap=plt.cm.gray)

```

Refer to the following screenshot for the end result:



The code is in the `clustering_hierarchy.ipynb` file in this book's code bundle.

See also

- ▶ The `AgglomerativeClustering` class documented at <http://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html> (retrieved December 2015)
- ▶ The Wikipedia page about hierarchical clustering at https://en.wikipedia.org/wiki/Hierarchical_clustering (retrieved December 2015)

Segmenting images with spectral clustering

Spectral clustering is a clustering technique that can be used to segment images. The scikit-learn `spectral_clustering()` function implements the normalized graph cuts spectral clustering algorithm. This algorithm represents an image as a graph of units. "Graph" here is the same mathematical concept as in *Chapter 8, Text Mining and Social Network Analysis*. The algorithm tries to partition the image, while minimizing segment size and the ratio of intensity gradient along cuts.

How to do it...

1. The imports are as follows:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.feature_extraction.image import img_to_graph
from sklearn.cluster import spectral_clustering
from sklearn.datasets import load_digits
```

2. Load the digits data set as follows:

```
digits = load_digits()
img = digits.images[0].astype(float)
mask = img.astype(bool)
```

3. Create a graph from the image:

```
graph = img_to_graph(img, mask=mask)
graph.data = np.exp(-graph.data/graph.data.std())
```

4. Apply spectral clustering to get three clusters:

```
labels = spectral_clustering(graph, n_clusters=3)
label_im = -np.ones(mask.shape)
label_im[mask] = labels
```

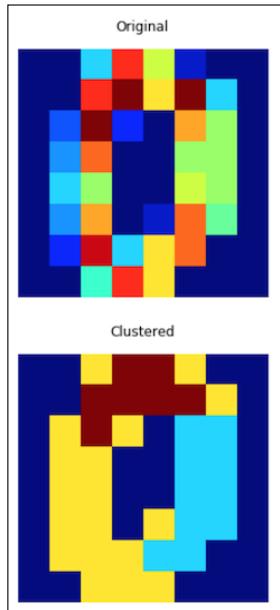
5. Plot the original image as follows:

```
plt.matshow(img, False)
plt.gca().axis('off')
plt.title('Original')
```

6. Plot the image with the three clusters as follows:

```
plt.figure()
plt.matshow(label_im, False)
plt.gca().axis('off')
plt.title('Clustered')
```

Refer to the following screenshot for the end result:



The code is in the `clustering_spectral.ipynb` file in this book's code bundle.

See also

- ▶ The Wikipedia page about spectral clustering at https://en.wikipedia.org/wiki/Spectral_clustering (retrieved December 2015)
- ▶ The `spectral_clustering()` function documented at http://scikit-learn.org/stable/modules/generated/sklearn.cluster.spectral_clustering.html (retrieved December 2015)

12

Parallelism and Performance

In this chapter, we will cover the following recipes:

- ▶ Just-in-time compiling with Numba
- ▶ Speeding up numerical expressions with Numexpr
- ▶ Running multiple threads with the `threading` module
- ▶ Launching multiple tasks with the `concurrent.futures` module
- ▶ Accessing resources asynchronously with the `asyncio` module
- ▶ Distributed processing with `execnet`
- ▶ Profiling memory usage
- ▶ Calculating the mean, variance, skewness, and kurtosis on the fly
- ▶ Caching with a least recently used cache
- ▶ Caching HTTP requests
- ▶ Streaming counting with the Count-min sketch
- ▶ Harnessing the power of the GPU with OpenCL

Introduction

The ENIAC, built between 1943 and 1946, filled a large room with eighteen thousand tubes and had a 20-bit memory. We have come a long way since then. The growth has been exponential as also predicted by Moore's law. Whether we are dealing with a self-fulfilling prophecy or a fundamental phenomenon is, of course, hard to say. Purportedly, the growth is starting to decelerate.

Given our current knowledge of technology, thermodynamics, and quantum mechanics, we can set hard limits for Moore's law. However, our assumptions may be wrong; for instance, scientists and engineers may come up with fundamentally better techniques to build chips. (One such development is quantum computing, which is currently far from widespread.) The biggest hurdle is heat dissipation, which is commonly measured in units of kT , with k the Boltzmann constant (about 10-23 J/K) and T in Kelvin (freezing point is 273.15 K). The heat dissipation per bit for a chip is at least kT (10-20 J at 350 K). Semi-conductors in the 1990s consumed at least a hundred thousand kT . A computational system undergoes changes in energy levels during operation. The smallest tolerable difference in energy is roughly 100 kT . Even if we somehow manage to avoid this limit, we will soon be operating close to atomic levels, which for quantum mechanical reasons is not practical (information about particles is fundamentally limited), unless we are talking about a quantum computer. Currently, the consensus is that we will reach the limit within decades. Another consideration is the complex wiring of chips. Complex wiring lowers the life expectancy of chips considerably.

This chapter is about software performance; however, there are other more important software aspects, such as maintainability, robustness, and usability. Betting on Moore's law is risky and not practical, since we have other possibilities to improve performance. The first option is to do the work in parallel as much as possible using multiple machines, cores on a single machine, GPUs, or other specialized hardware such as FPGAs. For instance, I am testing the code on an eight-core machine. As a student, I was lucky enough to get involved in a project with the goal of creating a grid. The grid was supposed to bring together university computers into a single computational environment. In a later phase, there were plans to connect other computers too, a bit like the SETI project. (As you know, many office computers are idle during weekends and at night, so why not make them work too?)

Currently, of course, there are various commercial cloud systems, such as those provided by Amazon and Google. I will not discuss those because I feel that these are more specialized topics, although I did cover some Python-specific cloud systems in *Python Data Analysis*.

The second method to improve performance is to apply caching, thereby avoiding unnecessary function calls. I covered the joblib library, which has a caching feature, in *Chapter 9, Ensemble Learning and Dimensionality Reduction*. Python 3 has brought us new features for parallelism and caching.

The third method is getting close to the metal. As you know, Python is a high-level programming language with a virtual machine and interpreter. Python has an extra layer, which a language unlike what C has. When I was a student, we were taught that C is a high-level language, with assembler and machine code as the lower levels. As far as I know, these days, practically nobody codes in assembler. Via Cython (covered in *Python Data Analysis*) and similar software, we can compile our code to obtain performance on a par with C and C++. Compiling is a hassle and is problematic because it reduces portability due to platform dependence. A common solution is to automate compiling with shell scripts and make files. Numba and other similar projects make life even easier with just-in-time compiling, although with some limitations.

Just-in-time compiling with Numba

The Numba software performs just-in-time compiling using special function decorators. The compilation produces native machine code automatically. The generated code can run on CPUs and GPUs. The main use case for Numba is math-heavy code that uses NumPy arrays.

We can compile the code with the `@numba.jit` decorator with optional function signature (for instance, `int32(int32)`). The types correspond with similar NumPy types. Numba operates in the `nopython` and `object` modes. The `nopython` mode is faster but more restricted. We can also release the **Global Interpreter Lock (GIL)** with the `nogil` option. You can cache the compilation results by requesting a file cache with the `cache` argument.

The `@vectorize` decorator converts functions with scalar arguments into NumPy ufuncs. Vectorization gives extra advantages, such as automatic broadcasting, and can be used on a single core, multiple cores in parallel, or a GPU.

Getting ready

Install Numba with the following command:

```
$ pip/conda install numba
```

I tested the code with Numba 0.22.1.

How to do it...

1. The imports are as follows:

```
from numba import vectorize
from numba import jit
import numpy as np
```

2. Define the following function to use the `@vectorize` decorator:

```
@vectorize
def vectorize_version(x, y, z):
    return x ** 2 + y ** 2 + z ** 2
```

3. Define the following function to use the `@jit` decorator:

```
@jit(nopython=True)
def jit_version(x, y, z):
    return x ** 2 + y ** 2 + z ** 2
```

4. Define some random arrays as follows:

```
np.random.seed(36)
x = np.random.random(1000)
y = np.random.random(1000)
z = np.random.random(1000)
```

5. Measure the time it takes to sum the squares of the arrays:

```
%timeit x ** 2 + y ** 2 + z ** 2
%timeit vectorize_version(x, y, z)
%timeit jit_version(x, y, z)
jit_version.inspect_types()
```

Refer to the following screenshot for the end result:

```
The slowest run took 10.80 times longer than the fastest. This could mean that an intermediate result is being cached
100000 loops, best of 3: 8.56 µs per loop
The slowest run took 44363.89 times longer than the fastest. This could mean that an intermediate result is being cached
100000 loops, best of 3: 2.87 µs per loop
The slowest run took 97089.67 times longer than the fastest. This could mean that an intermediate result is being cached
1000000 loops, best of 3: 1.82 µs per loop
jit_version (array(float64, 1d, C), array(float64, 1d, C), array(float64, 1d, C))
-----
# File: <ipython-input-2-85851b86f297>
# --- LINE 5 ---

@jit(nopython=True)

# --- LINE 6 ---

def jit_version(x, y, z):

    # --- LINE 7 ---
    # label 0
    #     x = arg(0, name=x)  :: array(float64, 1d, C)
    #     y = arg(1, name=y)  :: array(float64, 1d, C)
    #     z = arg(2, name=z)  :: array(float64, 1d, C)
    #     $const0.2 = const(int, 2)  :: int64
    #     $const0.5 = const(int, 2)  :: int64
```

The code is in the `compiling_numba.ipynb` file in this book's code bundle.

How it works

The best time measured is 1.82 microseconds on my machine, which is significantly faster than the measured time for normal Python code. At the end of the screenshot, we see the result of the compilation, with the last part omitted because it is too long and difficult to read. We get warnings, which are most likely caused by CPU caching. I left them on purpose, but you may be able to get rid of them using much larger arrays that don't fit in the cache.

See also

- ▶ The Numba website at <http://numba.pydata.org/> (retrieved January 2016)

Speeding up numerical expressions with Numexpr

Numexpr is a software package for the evaluation of numerical array expressions, which is also installed when you install pandas, and you may have seen it announced in the watermark of other recipes (tested with Numexpr 2.3.1). Numexpr tries to speed up calculations by avoiding the creation of temporary variables because reading the variables can be a potential bottleneck. The largest speedups are expected for arrays that can't fit in the CPU cache.

Numexpr splits large arrays into chunks, which fit in the cache, and it also uses multiple cores in parallel when possible. It has an `evaluate()` function, which accepts simple expressions and evaluates them (refer to the documentation for the complete list of supported features).

How to do it...

1. The imports are as follows:

```
import numexpr as ne
import numpy as np
```

2. Generate random arrays, which should be too large to hold in a cache:

```
a = np.random.rand(1e6)
b = np.random.rand(1e6)
```

3. Evaluate a simple arithmetic expression and measure execution time:

```
%timeit 2 * a ** 3 + 3 * b ** 9
%timeit ne.evaluate("2 * a ** 3 +3 * b ** 9 ")
```

Refer to the following screenshot for the end result:

10 loops, best of 3: 71.3 ms per loop
100 loops, best of 3: 3.05 ms per loop

The code is in the `speeding_numexpr.ipynb` file in this book's code bundle.

How it works

We generated random data that should not fit in a cache to avoid caching effects and because that is the best use case for Numexpr. The size of the cache differs from one machine to another, so if necessary use a larger or smaller size for the arrays. In the example, we put a string containing a simple arithmetic expression, although we could have used a slightly more complex expression. For more details, refer to the documentation. I tested the code with a machine that has eight cores. The speedup is larger than a factor of eight, so it's clearly due to Numexpr.

See also

- ▶ The Numexpr website at <https://pypi.python.org/pypi/numexpr> (retrieved January 2016)

Running multiple threads with the threading module

A computer process is an instance of a running program. Processes are actually heavyweight, so we may prefer threads, which are lighter. In fact, threads are often just subunits of a process. Processes are separated from each other, while threads can share instructions and data.

Operating systems typically assign one thread to each core (if there are more than one), or switch between threads periodically; this is called **time slicing**. Threads as processes can have different priorities and the operating system has daemon threads running in the background with very low priority.

It's easier to switch between threads than between processes; however, because threads share information, they are more dangerous to use. For instance, if multiple threads are able to increment a counter at the same time, this will make the code nondeterministic and potentially incorrect. One way to minimize risks is to make sure that only one thread can access a shared variable or shared function at a time. This strategy is implemented in Python as the GIL.

How to do it...

1. The imports are as follows:

```
import dautil as dl
import ch12util
from functools import partial
from queue import Queue
```

```
from threading import Thread
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import skew
from IPython.display import HTML
```

```
STATS = []
```

2. Define the following function to resample:

```
def resample(arr):
    sample = ch12util.bootstrap(arr)
    STATS.append((sample.mean(), sample.std(), skew(sample)))
```

3. Define the following class to bootstrap:

```
class Bootstrapper(Thread):
    def __init__(self, queue, data):
        Thread.__init__(self)
        self.queue = queue
        self.data = data
        self.log = dl.log_api.conf_logger(__name__)

    def run(self):
        while True:
            index = self.queue.get()

            if index % 10 == 0:
                self.log.debug('Bootstrap {}'.format(
                    index))

            resample(self.data)
            self.queue.task_done()
```

4. Define the following function to perform serial resampling:

```
def serial(arr, n):
    for i in range(n):
        resample(arr)
```

-
5. Define the following function to perform parallel resampling:

```
def threaded(arr, n):
    queue = Queue()

    for x in range(8):
        worker = Bootstrapper(queue, arr)
        worker.daemon = True
        worker.start()

    for i in range(n):
        queue.put(i)

    queue.join()
```

6. Plot distributions of moments and execution times:

```
sp = dl.plotting.Subplotter(2, 2, context)
temp = dl.data.Weather.load() ['TEMP'].dropna().values
np.random.seed(26)
threaded_times = ch12util.time_many(partial(threaded, temp))
serial_times = ch12util.time_many(partial(serial, temp))

ch12util.plot_times(sp.ax, serial_times, threaded_times)

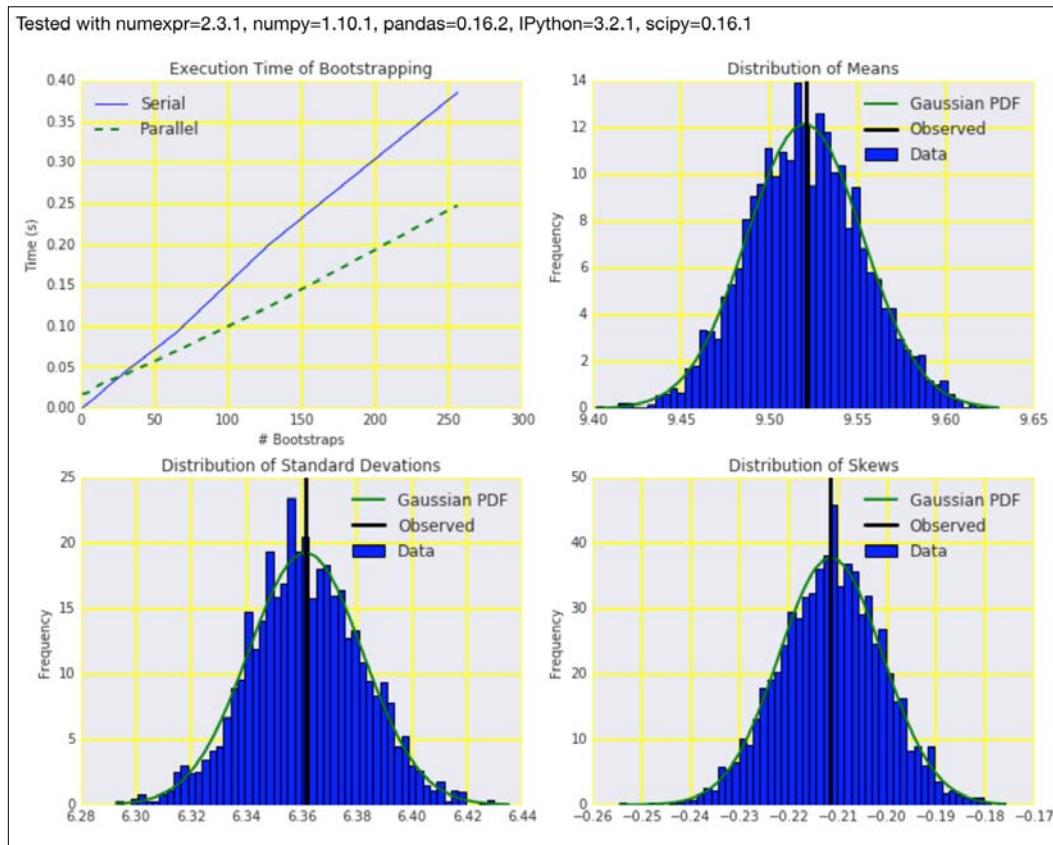
stats_arr = np.array(STATS)
ch12util.plot_distro(sp.next_ax(), stats_arr.T[0], temp.mean())
sp.label()

ch12util.plot_distro(sp.next_ax(), stats_arr.T[1], temp.std())
sp.label()

ch12util.plot_distro(sp.next_ax(), stats_arr.T[2], skew(temp))
sp.label()

HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code is in the `running_threads.ipynb` file in this book's code bundle.

See also

- ▶ The documentation for Python threading at <https://docs.python.org/3/library/threading.html> (retrieved January 2016)

Launching multiple tasks with the concurrent.futures module

The `concurrent.futures` module is a Python module with which we can execute callables asynchronously. If you are familiar with Java and go through the module, you will notice some similarities with the equivalent Java API, such as class names and architecture. According to the Python documentation, this is not a coincidence.

A task in this context is an autonomous unit of work. For instance, printing a document can be considered a task, but usually we consider much smaller tasks, such as adding two numbers.

How to do it...

1. The imports are as follows:

```
import dautil as dl
import ch12util
from functools import partial
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import skew
import concurrent.futures
from IPython.display import HTML

STATS = []
```

2. Define the following function to resample:

```
def resample(arr):
    sample = ch12util.bootstrap(arr)
    STATS.append((sample.mean(), sample.std(), skew(sample)))
```

3. Define the following class to bootstrap:

```
class Bootstrapper():
    def __init__(self, data):
        self.data = data
        self.log = dl.log_api.conf_logger(__name__)

    def run(self, index):
        if index % 10 == 0:
            self.log.debug('Bootstrap {}'.format(
                index))

        resample(self.data)
```

-
4. Define the following function to perform serial resampling:

```
def serial(arr, n):
    for i in range(n):
        resample(arr)
```

5. Define the following function to perform parallel resampling:

```
def parallel(arr, n):
    executor = concurrent.futures.ThreadPoolExecutor(max_
workers=8)
    bootstrapper = Bootstrapper(arr)

    for x in executor.map(bootstrapper.run, range(n)):
        pass

    executor.shutdown()
```

6. Plot distributions of moments and execution times:

```
rain = dl.data.Weather.load() ['RAIN'].dropna().values
np.random.seed(33)
parallel_times = ch12util.time_many(partial(parallel, rain))
serial_times = ch12util.time_many(partial(serial, rain))

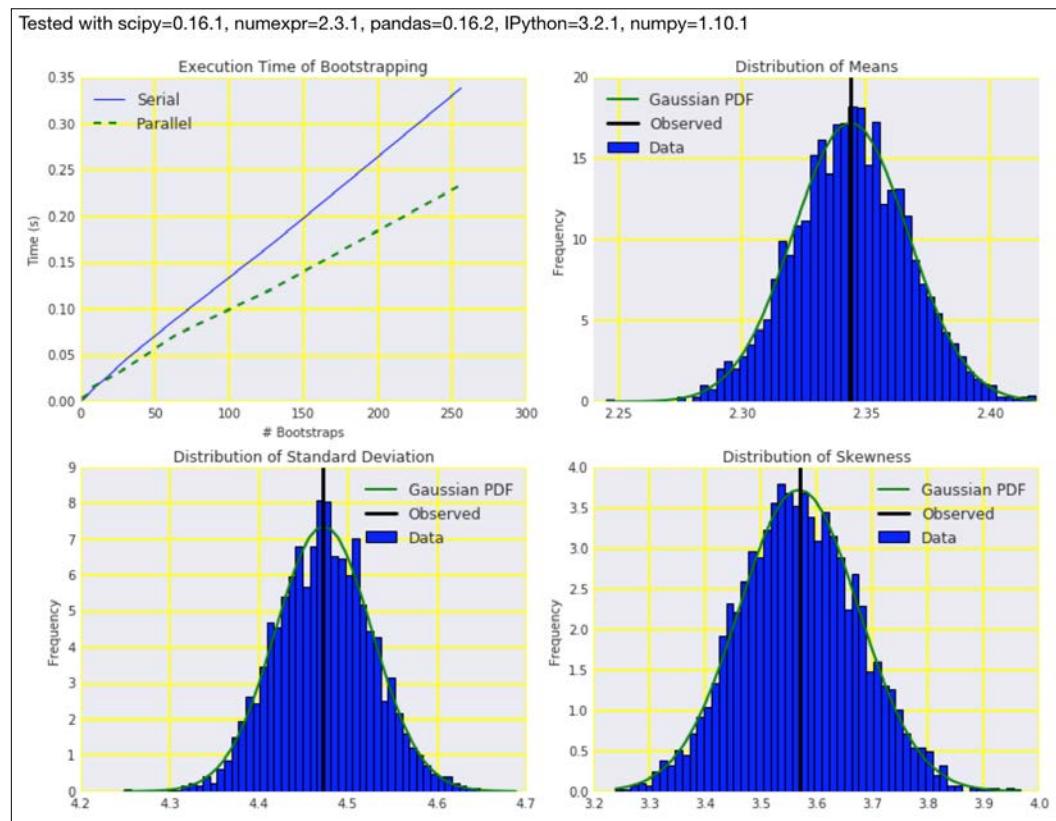
sp = dl.plotting.Subplotter(2, 2, context)
ch12util.plot_times(sp.ax, serial_times, parallel_times)

STATS = np.array(STATS)
ch12util.plot_distro(sp.next_ax(), STATS.T[0], rain.mean())
sp.label()

ch12util.plot_distro(sp.next_ax(), STATS.T[1], rain.std())
sp.label()

ch12util.plot_distro(sp.next_ax(), STATS.T[2], skew(rain))
sp.label()
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



The code is in the `launching_futures.ipynb` file in this book's code bundle.

See also

- ▶ The documentation for the `concurrent.futures` module at <https://docs.python.org/3/library/concurrent.futures.html> (retrieved January 2016)

Accessing resources asynchronously with the `asyncio` module

It is a basic fact of life that I/O (for example, file or database access) is slow. I/O is not only slow, but also unpredictable. In a common scenario, we wait for data (from a web service or sensors) and write the data to the filesystem or a database. In such a situation, we can find ourselves to be I/O bound—spending more time waiting for the data than actually processing it. We can poll for data periodically or act on event triggers (either check your watch or set an alarm). GUIs usually have special threads that wait for user input in an infinite loop.

The Python `asyncio` module for asynchronous I/O uses the concept of **coroutines** with a related function decorator. A brief example of this module was also given in the *Scraping the web* recipe of *Chapter 5, Web Mining, Databases, and Big Data*. Subroutines can be thought of as a special case of coroutines. A subroutine has a start and exit point, either through an early exit with a `return` statement or by reaching the end of the subroutine definition. In contrast, a coroutine can yield with the `yield from` statement by calling another coroutine and then resuming execution from that exit point. The coroutine is letting another coroutine take over, as it were, and is going back to sleep until it is activated again.

Subroutines can be placed on a single stack. However, coroutines require multiple stacks, which makes understanding the code and potential exceptions more complex.

How to do it...

The code is in the `accessing_asyncio.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import dautil as dl
import ch12util
from functools import partial
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import skew
import asyncio
import time
from IPython.display import HTML

STATS = []
```

-
2. Define the following function to resample:

```
def resample(arr):
    sample = ch12util.bootstrap(arr)
    STATS.append((sample.mean(), sample.std(), skew(sample)))
```

3. Define the following class to bootstrap:

```
class Bootstrapper():
    def __init__(self, data, queue):
        self.data = data
        self.log = dl.log_api.conf_logger(__name__)
        self.queue = queue

    @asyncio.coroutine
    def run(self):
        while not self.queue.empty():
            index = yield from self.queue.get()

            if index % 10 == 0:
                self.log.debug('Bootstrap {}'.format(
                    index))

            resample(self.data)
            # simulates slow IO
            yield from asyncio.sleep(0.01)
```

4. Define the following function to perform serial resampling:

```
def serial(arr, n):
    for i in range(n):
        resample(arr)
        # simulates slow IO
        time.sleep(0.01)
```

5. Define the following function to perform parallel resampling:

```
def parallel(arr, n):
    q = asyncio.Queue()

    for i in range(n):
        q.put_nowait(i)

    bootstrapper = Bootstrapper(arr, q)
    policy = asyncio.get_event_loop_policy()
    policy.set_event_loop(policy.new_event_loop())
    loop = asyncio.get_event_loop()
```

```
tasks = [asyncio.async(bootstrapper.run())
         for i in range(n)]

loop.run_until_complete(asyncio.wait(tasks))
loop.close()
```

6. Plot distributions of moments and execution times:

```
pressure = dl.data.Weather.load()['PRESSURE'].dropna().values
np.random.seed(33)
parallel_times = ch12util.time_many(partial(parallel, pressure))
serial_times = ch12util.time_many(partial(serial, pressure))

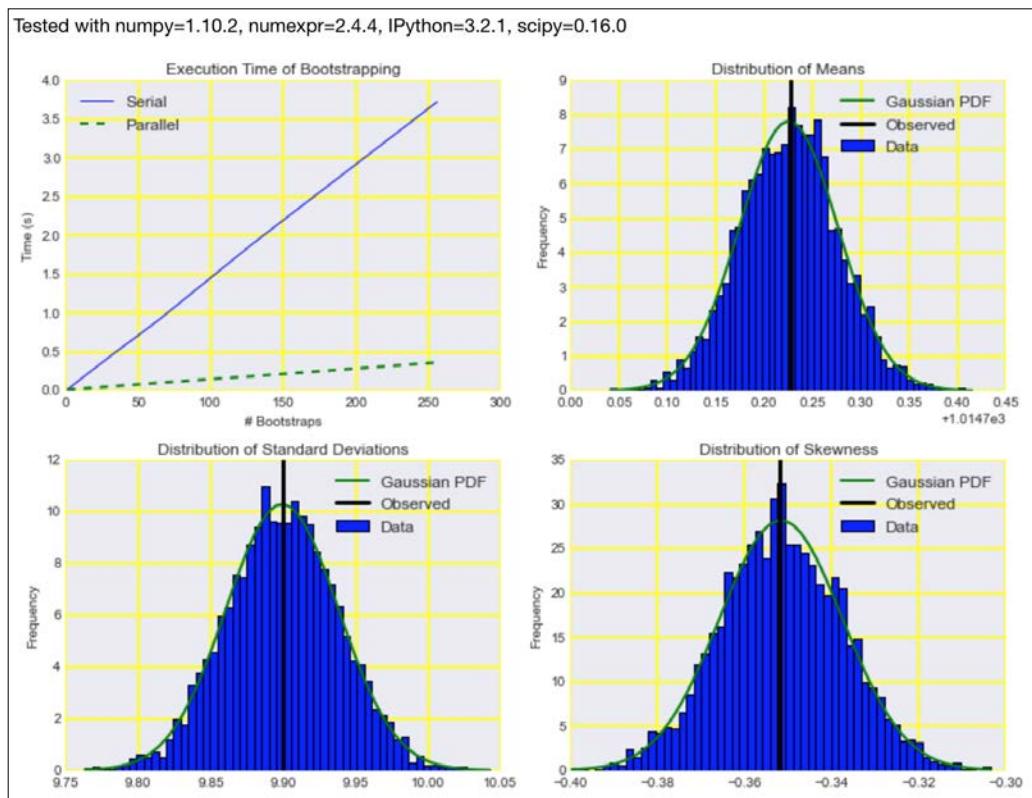
dl.options.mimic_seaborn()
ch12util.plot_times(plt.gca(), serial_times, parallel_times)

sp = dl.plotting.Subplotter(2, 2, context)
ch12util.plot_distro(sp.next_ax(), STATS.T[0], pressure.mean())
sp.label()

ch12util.plot_distro(sp.next_ax(), STATS.T[1], pressure.std())
sp.label()

ch12util.plot_distro(sp.next_ax(), STATS.T[2], skew(pressure))
sp.label()
HTML(sp.exit())
```

Refer to the following screenshot for the end result:



See also

- ▶ The documentation for the `asyncio` module at <https://docs.python.org/3/library/asyncio.html> (retrieved January 2016)
- ▶ The related Wikipedia page at <https://en.wikipedia.org/wiki/Coroutine> (retrieved January 2016)

Distributed processing with execnet

The `execnet` module has a share-nothing model and uses **channels** for communication. Channels in this context are software abstractions used to send and receive messages between (distributed) computer processes. `execnet` is most useful for combining heterogeneous computing environments with different Python interpreters and installed software. The environments can have different operating systems and Python implementations (CPython, Jython, PyPy, or others).

In the **shared nothing architecture**, computing nodes don't share memory or files. The architecture is therefore totally decentralized with completely independent nodes. The obvious advantage is that we are not dependent on any one node.

Getting ready

Install execnet with the following command:

```
$ pip/conda install execnet
```

I tested the code with execnet 1.3.0.

How to do it...

1. The imports are as follows:

```
import dautil as dl
import ch12util
from functools import partial
import matplotlib.pyplot as plt
import numpy as np
import execnet
```

```
STATS = []
```

2. Define the following helper function:

```
def run(channel, data=[]):
    while not channel.isclosed():
        index = channel.receive()

        if index % 10 == 0:
            print('Bootstrap {}'.format(
                index))

        total = 0

        for x in data:
            total += x

        channel.send((total - data[index])/(len(data) - 1))
```

-
3. Define the following function to perform serial resampling:

```
def serial(arr, n):
    for i in range(n):
        total = 0

        for x in arr:
            total += x

    STATS.append((total - arr[i])/(len(arr) - 1))
```

4. Define the following function to perform parallel resampling:

```
def parallel(arr, n):
    gw = execnet.makegateway()
    channel = gw.remote_exec(run, data=arr.tolist())

    for i in range(n):
        channel.send(i)
        STATS.append(channel.receive())

    gw.exit()
```

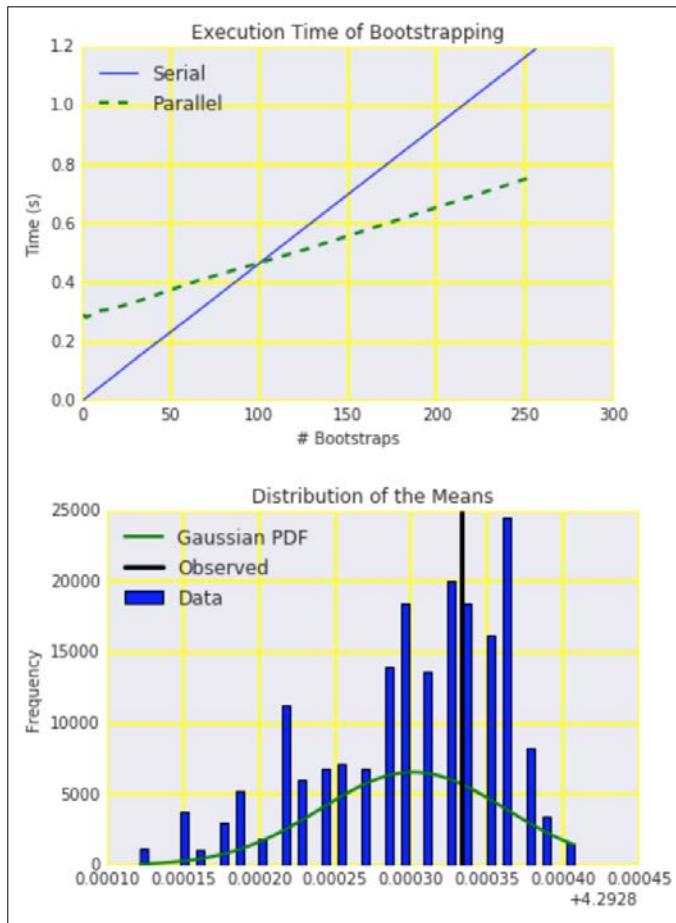
5. Plot distributions of means and execution times:

```
ws = dl.data.Weather.load() ['WIND_SPEED'].dropna().values
np.random.seed(33)
parallel_times = ch12util.time_many(partial(parallel, ws))
serial_times = ch12util.time_many(partial(serial, ws))

%matplotlib inline
dl.options.mimic_seaborn()
ch12util.plot_times(plt.gca(), serial_times, parallel_times)
plt.legend(loc='best')

plt.figure()
STATS = np.array(STATS)
ch12util.plot_distro(plt.gca(), STATS, ws.mean())
plt.title('Distribution of the Means')
plt.legend(loc='best')
```

Refer to the following screenshot for the end result:



The code is in the `distributing_execnet.ipynb` file in this book's code bundle.

See also

- ▶ The execnet website at <http://codespeak.net/execnet/> (retrieved January 2016)
- ▶ The related Wikipedia page at https://en.wikipedia.org/wiki/Shared_nothing_architecture (retrieved January 2016)

Profiling memory usage

In *Python Data Analysis*, we used various profiling tools. These tools mostly had to do with measuring execution times. However, memory is also important, especially if we don't have enough of it. **Memory leaks** are a common issue with computer programs that we can find by performing memory profiling. Leaks occur when we don't release memory that is not needed. Problems also may occur when we use data types that require more memory than we need, for instance, NumPy `float64` arrays when integer arrays will do.

The Python `memory_profiler` module can profile memory usage of code line by line. Once you install it, you can also use the module in an IPython notebook via various magic commands. The module works by communicating with the operating system. On Windows, you will require the Python `psutil` package for communication.

Getting ready

Install `memory_profiler` with the following command:

```
$ pip install memory-profiler
```

I tested the code with `memory_profiler` 0.39.

Create a script to profile (refer to the `mem_test.py` file in this book's code bundle):

```
import numpy as np

def test_me():
    a = np.random.random((999, 99))
    b = np.random.random((99, 99))
    a.ravel()
    b.tolist()
```

How to do it...

1. The imports are as follows:

```
import dautil as dl
from mem_test import test_me
```

2. Load the IPython extension as follows:

```
%load_ext memory_profiler
```

3. Profile the test script line-by-line with the following command:

```
%mprun -f test_me test_me()
```

Refer to the following screenshot for the end result:

Line #	Mem usage	Increment	Line Contents
<hr/>			
4	108.5 MiB	0.0 MiB	def test_me():
5	109.3 MiB	0.8 MiB	a = np.random.random((999, 99))
6	109.3 MiB	0.0 MiB	b = np.random.random((99, 99))
7	109.3 MiB	0.0 MiB	a.ravel()
8	109.5 MiB	0.2 MiB	b.tolist()

The code is in the `profiling_memory.ipynb` file in this book's code bundle.

See also

- ▶ The memory_profiler website at https://pypi.python.org/pypi/memory_profiler (retrieved January 2016)

Calculating the mean, variance, skewness, and kurtosis on the fly

Mean, variance, skewness, and kurtosis are important quantities in statistics. Some of the calculations involve sums of squares, which for large values may lead to overflow. To avoid loss of precision, we have to realize that variance is invariant under shift by a certain constant number.

When we have enough space in memory, we can directly calculate the moments, taking into account numerical issues if necessary. However, we may want to not keep the data in memory because there is a lot of it, or because it is more convenient to calculate the moments on the fly.

An online and numerically stable algorithm to calculate the variance has been provided by Terriberry (Terriberry, Timothy B. (2007), *Computing Higher-Order Moments Online*). We will compare this algorithm, although it is not the best one, to the implementation in the `LiveStats` module. If you are interested in improved algorithms, take a look at the Wikipedia page listed in the See also section.

Take a look at the following equations:

$$(12.1) \quad \delta = x - m$$

$$(12.2) \quad m' = m + \frac{\delta}{n}$$

$$(12.3) \quad M_2' = M_2 + \delta^2 \frac{n-1}{n}$$

$$(12.4) \quad M_3' = M_3 + \delta^3 \frac{(n-1)(n-2)}{n^2} - \frac{3\delta M_2}{n}$$

$$(12.5) \quad M_4' = M_4 + \frac{\delta^4 (n-1)(n^2 - 3n + 3)}{n^3} + \frac{6\delta^2 M_2}{n^2} - \frac{4\delta M_3}{n}$$

$$(12.6) \quad g_1 = \frac{\sqrt{nM_3}}{M_2^{3/2}}$$

$$(12.7) \quad g_2 = \frac{nM_4}{M_2^2} - 3$$

Skewness is given by 12.6 and kurtosis is given by 12.7.

Getting ready

Install LiveStats with the following command:

```
$ pip install LiveStats
```

I tested the code with LiveStats 1.0.

How to do it...

1. The imports are as follows:

```
from livestats import livestats
from math import sqrt
import dautil as dl
import numpy as np
from scipy.stats import skew
from scipy.stats import kurtosis
import matplotlib.pyplot as plt
```

2. Define the following function to implement the equations for the moments calculation:

```
# From https://en.wikipedia.org/wiki/
# Algorithms_for_calculating_variance
def online_kurtosis(data):
    n = 0
    mean = 0
    M2 = 0
    M3 = 0
    M4 = 0
    stats = []

    for x in data:
        n1 = n
        n = n + 1
        delta = x - mean
        delta_n = delta / n
        delta_n2 = delta_n ** 2
        term1 = delta * delta_n * n1
        mean = mean + delta_n
        M4 = M4 + term1 * delta_n2 * (n**2 - 3*n + 3) + \
            6 * delta_n2 * M2 - 4 * delta_n * M3
        M3 = M3 + term1 * delta_n * (n - 2) - 3 * delta_n * M2
        M2 = M2 + term1
        s = sqrt(n) * M3 / sqrt(M2 ** 3)
        k = (n*M4) / (M2**2) - 3
        stats.append((mean, sqrt(M2/(n - 1)), s, k))

    return np.array(stats)
```

3. Initialize and load data as follows:

```
test = livestats.LiveStats([0.25, 0.5, 0.75])

data = dl.data.Weather.load() ['TEMP'] .\
    resample('M') .dropna() .values
```

4. Calculate the various statistics with LiveStats, the algorithm mentioned in the previous section, and compare with the results when we apply NumPy functions to all the data at once:

```
ls = []
truth = []

test.add(data[0])
```

```
for i in range(1, len(data)):
    test.add(data[i])
    q1, q2, q3 = test.quantiles()

    ls.append((test.mean(), sqrt(test.variance()),
               test.skewness(), test.kurtosis(), q1[1], q2[1],
               q3[1]))
    slice = data[:i]
    truth.append((slice.mean(), slice.std(),
                  skew(slice), kurtosis(slice),
                  np.percentile(slice, 25), np.median(slice),
                  np.percentile(slice, 75)))

ls = np.array(ls)
truth = np.array(truth)
ok = online_kurtosis(data)
```

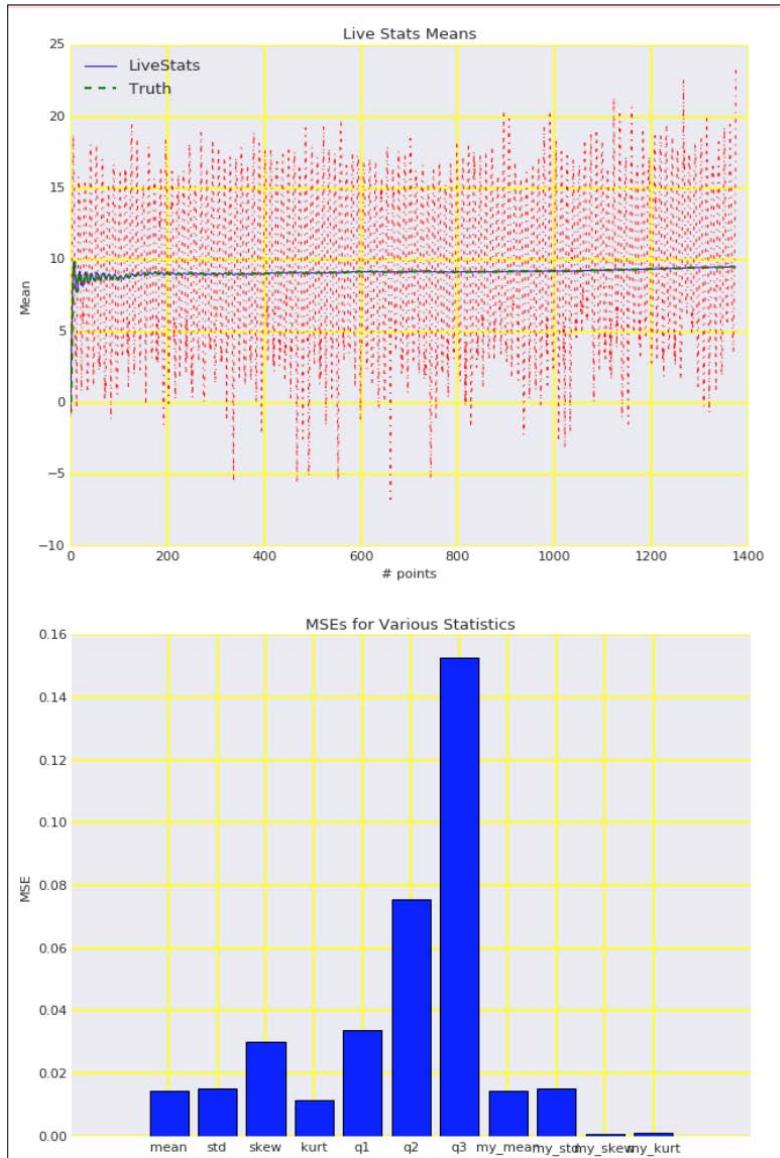
5. Plot the results as follows:

```
dl.options.mimic_seaborn()
cp = dl.plotting.CyclePlotter(plt.gca())
cp.plot(ls.T[0], label='LiveStats')
cp.plot(truth.T[0], label='Truth')
cp.plot(data)
plt.title('Live Stats Means')
plt.xlabel('# points')
plt.ylabel('Mean')
plt.legend(loc='best')

plt.figure()

mses = [dl.stats.mse(truth.T[i], ls.T[i])
        for i in range(7)]
mses.extend([dl.stats.mse(truth.T[i], ok[1:].T[i])
             for i in range(4)])
dl.plotting.bar(plt.gca(),
                ['mean', 'std', 'skew', 'kurt',
                 'q1', 'q2', 'q3',
                 'my_mean', 'my_std', 'my_skew', 'my_kurt'], mses)
plt.title('MSEs for Various Statistics')
plt.ylabel('MSE')
```

Refer to the following screenshot for the end result:



The code is in the `calculating_moments.ipynb` file in this book's code bundle.

See also

- ▶ The LiveStats website at <https://bitbucket.org/scassidy/livestats> (retrieved January 2016)
- ▶ The related Wikipedia page at https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance (retrieved January 2016)

Caching with a least recently used cache

Caching involves storing results, usually from a function call, in memory or on disk. If done correctly, caching helps by reducing the number of function calls. In general, we want to keep the cache small for space reasons. If we are able to find items in cache, we talk about hits; otherwise, we have misses. Obviously, we want to have as many hits as possible and as few misses as possible. This means that we want to maximize the hits-misses ratio.

Many caching algorithms exist, of which we will cover the **least recently used (LRU)** algorithm. This algorithm keeps track of when a cache item was used. If the cache is about to exceed its maximum specified size, LRU gets rid of the least recently used item. The reasoning is that these items are possibly older and, therefore, not as relevant any more. There are several variations of LRU. Other algorithms do the opposite—removing the most recent item, the least frequently used item, or a random item.

The standard Python library has an implementation of LRU, but there is also a specialized Python library with some parts implemented in C and it is therefore potentially faster. We will compare the two implementations using the NLTK `lemmatize()` method (refer to the *Stemming, lemmatizing, filtering and TF-IDF scores* recipe in *Chapter 8, Text Mining and Social Network Analysis*).

Getting ready

Install fastcache as follows:

```
$ pip/conda install fastcache
```

I tested the code with fastcache 1.0.2.

How to do it...

1. The imports are as follows:

```
from fastcache import clru_cache
from functools import lru_cache
from nltk.corpus import brown
from nltk.stem import WordNetLemmatizer
```

```
import daultil as dl
import numpy as np
from IPython.display import HTML
```

2. Define the following function to cache:

```
def lemmatize(word, lemmatizer):
    return lemmatizer.lemmatize(word.lower())
```

3. Define the following function to measure the effects of caching:

```
def measure(impl, words, lemmatizer):
    cache = dl.perf.LRUcache(impl, lemmatize)
    times = []
    hm = []

    for i in range(5, 12):
        cache.maxsize = 2 ** i
        cache.cache()
        with dl.perf.StopWatch() as sw:
            _ = [cache.cached(w, lemmatizer) for w in words]

        hm.append(cache.hits_miss())
        times.append(sw.elapsed)
        cache.clear()

    return (times, hm)
```

4. Initialize a list of words and an NLTK WordNetLemmatizer object:

```
words = [w for w in brown.words()]
lemmatizer = WordNetLemmatizer()
```

5. Measure the execution time as follows:

```
with dl.perf.StopWatch() as sw:
    _ = [lemmatizer.lemmatize(w.lower()) for w in words]

plain = sw.elapsed

times, hm = measure(clru_cache, words, lemmatizer)
```

6. Plot the results for different cache sizes:

```
sp = dl.plotting.Subplotter(2, 2, context)
sp.ax.plot(2 ** np.arange(5, 12), times)
sp.ax.axhline(plain, lw=2, label='Uncached')
sp.label()
```

```

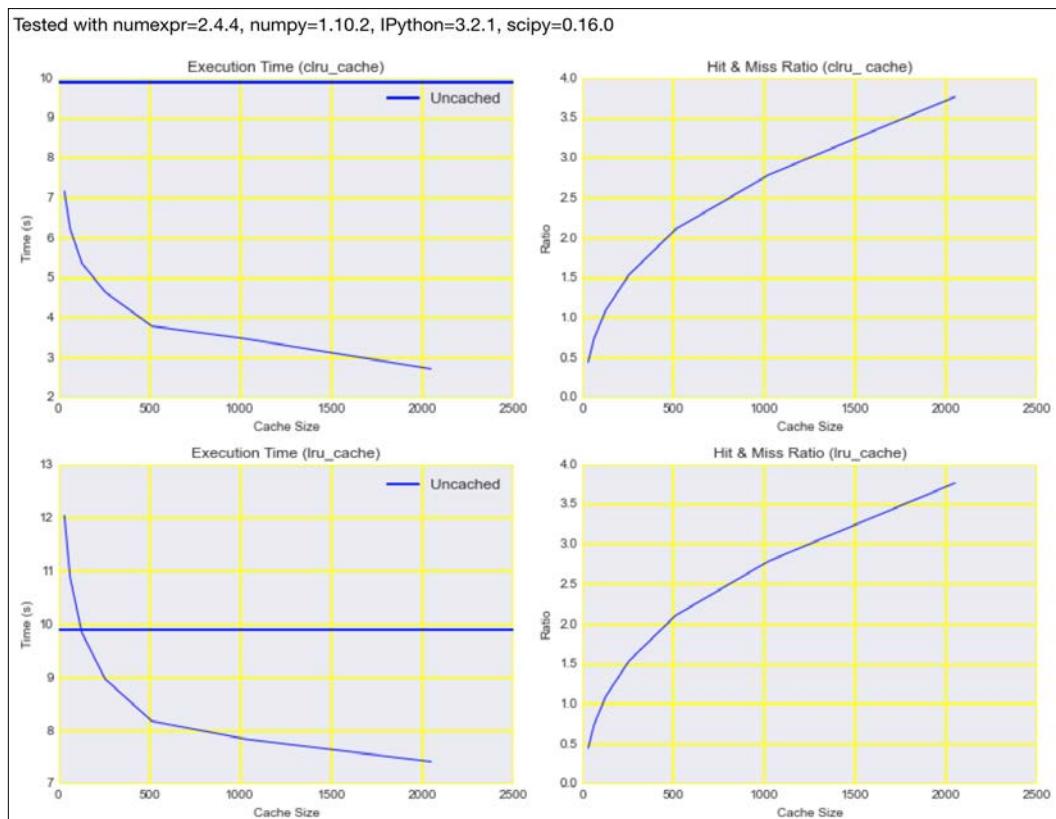
sp.next_ax().plot(2 ** np.arange(5, 12), hm)
sp.label()

times, hm = measure(lru_cache, words, lemmatizer)
sp.next_ax().plot(2 ** np.arange(5, 12), times)
sp.ax.axhline(plain, lw=2, label='Uncached')
sp.label()

sp.next_ax().plot(2 ** np.arange(5, 12), hm)
sp.label()
HTML(sp.exit())

```

Refer to the following screenshot for the end result:



The code is in the `caching_lru.ipynb` file in this book's code bundle.

See also

- ▶ The related Wikipedia page at http://en.wikipedia.org/wiki/Cache_algorithms (retrieved January 2016)
- ▶ The fastcache website at <https://pypi.python.org/pypi/fastcache> (retrieved January 2016)
- ▶ The functools.lru_cache documentation at https://docs.python.org/3/library/functools.html#functools.lru_cache (retrieved January 2016)

Caching HTTP requests

Sometimes data is made available via a web service over HTTP. The advantage is that we don't have to care that much about the technologies that the sending party is using. This is comparable to the way e-mail, for instance, works. However, we have to explicitly request information via an HTTP GET (often) or HTTP POST (uppercase by convention) method. Whenever we request a web page or download a file, we usually perform a GET request. The web server on the other side has to process the request. If there are many requests, we can potentially slow down the server, so organizations often take measures to prevent this. It may mean that further requests from you will be blocked.

Avoiding issuing the same request multiple times is advantageous for efficiency reasons too. Web browsers solve this with a cache, and we can do the same with the `requests-cache` package. The cache is stored in a SQLite database by default.

A common use case that we will not cover is that of periodically retrieving information with HTTP. Obviously, we don't want to retrieve content if nothing has changed. The HTTP protocol provides efficient mechanisms to determine whether content was modified. A web server, however, is not required to report content changes.

Getting ready

Install `requests-cache` with the following command:

```
$ pip install --upgrade requests-cache
```

I tested the code with `requests-cache` 0.4.10.

How to do it...

1. The imports are as follows:

```
import requests
import requests_cache
```

2. Install the cache (this creates a SQLite database by default):

```
requests_cache.install_cache()
```

3. Request a website that builds the cache:

```
%time requests.get('http://google.com')
```

4. Request the same website that should now come from the local cache:

```
%time requests.get('http://google.com')
```

5. Clear the cache as follows:

```
requests_cache.clear()
```

6. Request the website yet again (the cache should be empty now):

```
%time requests.get('http://google.com')
```

Refer to the following screenshot for the end result:

```
CPU times: user 14 ms, sys: 5.51 ms, total: 19.5 ms
Wall time: 42.1 ms
CPU times: user 3.56 ms, sys: 1 ms, total: 4.57 ms
Wall time: 5.29 ms
CPU times: user 14.4 ms, sys: 4.28 ms, total: 18.7 ms
Wall time: 952 ms
```

The code is in the `caching_requests.ipynb` file in this book's code bundle.

See also

- ▶ The related Wikipedia page at https://en.wikipedia.org/wiki/HTTP_ETag (retrieved January 2016)
- ▶ The requests-cache website at <https://pypi.python.org/pypi/requests-cache> (retrieved January 2016)

Streaming counting with the Count-min sketch

Streaming or online algorithms are useful as they don't require as much memory and processing power as other algorithms. This chapter has a recipe involving the calculation of statistical moments online (refer to *Calculating the mean, variance, skewness, and kurtosis on the fly*).

Also, in the *Clustering streaming data with Spark* recipe of Chapter 5, *Web Mining, Databases, and Big Data*, I covered another streaming algorithm.

Streaming algorithms are often approximate for fundamental reasons or because of roundoff errors. You should, therefore, try to use other algorithms if possible. Of course in many situations approximate results are good enough. For instance, it doesn't matter whether a user has 500 or 501 connections on a social media website. If you just send thousands of invitations, you will get there sooner or later.

Sketching is something you probably know from drawing. In that context, sketching means outlining rough contours of objects without any details. A similar concept exists in the world of streaming algorithms.

In this recipe, I cover the *Count-min sketch* (2003) by Graham Cormode and S. Muthu Muthukrishnan, which is useful in the context of ranking. For example, we may want to know the most viewed articles on a news website, trending topics, the ads with the most clicks, or the users with the most connections. The naive approach requires keeping counts for each item in a dictionary or a table. Dictionaries use hashing functions to calculate identifying integers, which serve as keys. For theoretical reasons, we can have collisions—this means that two or more items have the same key. The Count-min sketch is a two-dimensional tabular data structure that is small on purpose, and it uses hashing functions for each row. It is prone to collisions, leading to overcounting.

When an event occurs, for instance someone views an ad, we do the following:

1. For each row in the sketch, we apply the related hashing function using, for instance, the ad identifier to get a column index.
2. Increment the value corresponding with the row and column.

Each event is clearly mapped to each row in the sketch. When we request the count, we follow the opposite path to obtain multiple counts. The lowest count gives an estimate for the count of this item.

The idea behind this setup is that frequent items are likely to dominate less common items. The probability of a popular item having collisions with unpopular items is larger than of collisions between popular items.

How to do it...

1. The imports are as follows:

```
from nltk.corpus import brown
from nltk.corpus import stopwords
import dautil as dl
from collections import defaultdict
import matplotlib.pyplot as plt
import numpy as np
from IPython.display import HTML
```

2. Store the words of the NLTK Brown and stop words corpora in lists:

```
words_dict = dl.collect.IdDict()
dd = defaultdict(int)
fid = brown.fileids(categories='news')[0]
words = brown.words(fid)
sw = set(stopwords.words('english'))
```

3. Count the occurrence of each stopword:

```
for w in words:
    if w in sw:
        dd[w] += 1
```

4. Plot the distribution of count errors for various parameters of the Count-min sketch:

```
sp = dl.plotting.Subplotter(2, 2, context)
actual = np.array([dd[w] for w in sw])
errors = []

for i in range(1, 4):
    cm = dl.perf.CountMinSketch(depth=5 * 2 ** i,
                                  width=20 * 2 ** i)

    for w in words:
        cm.add(words_dict.add_or_get(w.lower()))

    estimates = np.array([cm.estimate_count(words_dict.add_or_
get(w))
                           for w in sw])
    diff = estimates - actual
    errors.append(diff)

    if i > 1:
        sp.next_ax()
```

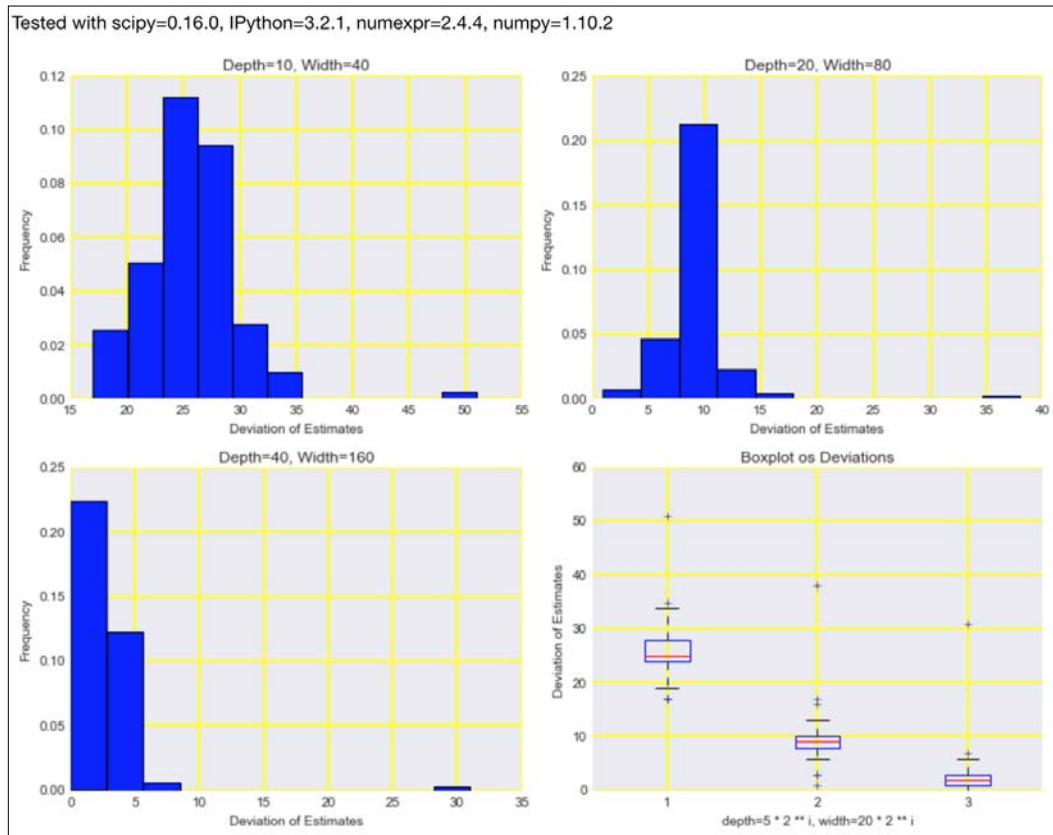
```

sp.ax.hist(diff, normed=True,
           bins=dl.stats.sqrt_bins(actual))
sp.label()

sp.next_ax().boxplot(errors)
sp.label()
HTML(sp.exit())

```

Refer to the following screenshot for the end result:



The code is in the `stream_demo.py` file in this book's code bundle.

See also

- ▶ The related Wikipedia page at https://en.wikipedia.org/wiki/Count%20min_sketch (retrieved January 2016)

Harnessing the power of the GPU with OpenCL

Open Computing Language (OpenCL), initially developed by Apple Inc., is an open technology standard for programs, which can run on a variety of devices, including CPUs and GPUs that are available on commodity hardware, such as the machine I am using for this recipe. Since 2009, OpenCL has been maintained by the Khronos Compute Working Group. Many hardware vendors, including the one I am partial to, have an implementation of OpenCL.

OpenCL is a language resembling C (actually, there are multiple C dialects or versions) with functions called **ernels**. Kernels can run in parallel on multiple processing elements. The hardware vendor gives the definition of the processing element. OpenCL programs are compiled at runtime for the purpose of portability.

Portability is the greatest advantage of OpenCL over similar technologies such as CUDA, which is an NVIDIA product. Another advantage is the ability to share work between CPUs, GPUs, and other devices. It has been suggested to use machine learning for optimal division of labor.

Pythonistas can write OpenCL programs with the PyOpenCL package. PyOpenCL adds extra features, such as object cleanup and conversion of errors, to Python exceptions. A number of other libraries use and in some ways enhance PyOpenCL (refer to the PyOpenCL documentation).

Getting ready

Install pyopencl with the following command:

```
$ pip install pyopencl
```

I tested the code with PyOpenCL 2015.2.3. For more information, please refer to <https://wiki.tiker.net/OpenCLHowTo>.

How to do it...

The code is in the `opencl_demo.ipynb` file in this book's code bundle:

1. The imports are as follows:

```
import pyopencl as cl
from pyopencl import array
import numpy as np
```

2. Define the following function to accept a NumPy array and perform a simple computation:

```
def np_se(a, b):
    return (a - b) ** 2
```

3. Define the following function to do the same calculation as in the previous step using OpenCL:

```
def gpu_se(a, b, platform, device, context, program):
```

4. Create a queue with profiling enabled (only for demonstration) and buffers to shuffle data around:

```
queue = cl.CommandQueue(context,
                        properties=cl.command_queue_
properties.
PROFILING_ENABLE)

mem_flags = cl.mem_flags
a_buf = cl.Buffer(context,
                  mem_flags.READ_ONLY | mem_flags.COPY_HOST_
PTR,
hostbuf=a)
b_buf = cl.Buffer(context,
                  mem_flags.READ_ONLY | mem_flags.COPY_HOST_
PTR, hostbuf=b)
error = np.empty_like(a)
destination_buf = cl.Buffer(context,
                            mem_flags.WRITE_ONLY,
error.nbytes)
```

5. Execute the OpenCL program and profile the code:

```
exec_evt = program.mean_squared_error(queue, error.shape,
None,
a_buf, b_buf,
destination_buf)
exec_evt.wait()
elapsed = 1e-9*(exec_evt.profile.end - exec_evt.profile.start)

print("Execution time of OpenCL: %g s" % elapsed)

cl.enqueue_copy(queue,
               error, destination_buf)

return error
```

-
6. Generate random data as follows:

```
np.random.seed(51)
a = np.random.rand(4096).astype(np.float32)
b = np.random.rand(4096).astype(np.float32)
```

7. Access CPU and GPUs. This part is hardware dependent, so you may have to change these lines:

```
platform = cl.get_platforms()[0]
device = platform.get_devices()[2]
context = cl.Context([device])
```

8. Define a kernel with the OpenCL language:

```
program = cl.Program(context, """
__kernel void mean_squared_error(__global const float *a,
__global const float *b, __global float *result)
{
    int gid = get_global_id(0);
    float temp = a[gid] - b[gid];
    result[gid] = temp * temp;
}
""").build()
```

9. Calculate squared errors with NumPy and OpenCL (GPU) and measure execution times:

```
gpu_error = gpu_se(a, b, platform, device, context, program)

np_error = np_se(a, b)
print('GPU error', np.mean(gpu_error))
print('NumPy error', np.mean(np_error))
%time np_se(a, b)
```

Refer to the following screenshot for the end result:

```
Execution time of OpenCL: 1.0528e-05 s
GPU error 0.168414
NumPy error 0.168414
CPU times: user 16 µs, sys: 5 µs, total: 21 µs
Wall time: 25 µs
```

See also

- ▶ The PyOpenCL website at <http://documentacion.de/pyopencl/> (retrieved January 2016)

A

Glossary

This appendix is a brief glossary of technical concepts used throughout *Python Data Analysis* and this book.

American Standard Code for Information Interchange (ASCII) was the dominant encoding standard on the Internet until the end of 2007, with UTF-8 (8-bit Unicode) taking over. ASCII is limited to the English alphabet and has no support for other alphabets.

Analysis of variance (ANOVA) is a statistical data analysis method invented by statistician Ronald Fisher. This method partitions the data of a continuous variable using the values of one or more corresponding categorical variable to analyze variance. ANOVA is a form of linear modeling.

Anaconda is a free Python distribution for data analysis and scientific computing. It has its own package manager, **conda**.

The **Anscombe's quartet** is a classic example, which illustrates why visualizing data is important. The quartet consists of four datasets with similar statistical properties. Each dataset has a series of x values and dependent y values.

The **bag-of-words model**: A simplified model of text, in which text is represented by a bag (a set in which something can occur multiple times) of words. In this representation, the order of the words is ignored. Typically, word counts or the presence of certain words are used as features in this model.

Beta in finance is the slope of a linear regression model involving the returns of the asset and the returns of a benchmark, for instance, the S & P 500 index.

Caching involves storing results, usually from a function call, in memory or on disk. If done correctly, caching helps by reducing the number of function calls. In general, we want to keep the cache small for space reasons.

A **clique** is a subgraph that is complete. This is equivalent to the general concept of cliques, in which every person knows all the other people.

Clustering aims to partition data into groups called clusters. Clustering is unsupervised in the sense that the training data is not labeled. Some clustering algorithms require a guess for the number of clusters, while other algorithms don't.

Cohen's kappa measures agreement between the target and predicted class (in the context of classification)—similar to accuracy, but it also takes into account the random chance of getting the predictions. Kappa varies between negative values and one.

A **complete graph** is a graph in which every pair of nodes is connected by a unique connection.

The **confusion matrix** is a table usually used to summarize the results of classification. The two dimensions of the table are the predicted class and the target class.

Contingency table: A table containing counts for all combinations of the two categorical variables.

The **cosine similarity** is a common distance metric to measure the similarity of two documents. For this metric, we need to compute the inner product of two feature vectors. The cosine similarity of vectors corresponds to the cosine of the angle between vectors, hence the name.

Cross-correlation measures the correlation between two signals using a sliding inner product. We can use cross-correlation to measure the time delay between two signals.

The **Data Science Toolbox (DST)** is a virtual environment based on Ubuntu for data analysis using Python and R. Since DST is a virtual environment, we can install it on various operating systems.

The **discrete cosine transform (DCT)** is a transform similar to the Fourier transform, but it tries to represent a signal by a sum of cosine terms only.

The **efficient-market hypothesis (EMH)** stipulates that you can't, on average, "beat the market" by picking better stocks or timing the market. According to the EMH, all information about the market is immediately available to every market participant in one form or another and is immediately reflected in asset prices.

Eigenvalues are scalar solutions to the equation $Ax = ax$, where A is a two-dimensional matrix and x is a one-dimensional vector.

Eigenvectors are vectors corresponding to eigenvalues.

Exponential smoothing is a low-pass filter, which aims to remove noise.

Face detection tries to find (rectangular) areas in an image that represent faces.

Fast Fourier transform (FFT): A fast algorithm to compute Fourier transforms. FFT is $O(N \log N)$, which is a huge improvement on older algorithms.

Filtering is a type of signal processing technique, involving the removal or suppression of part of the signal. Many filter types exist, including the median and Wiener filters.

Fourier analysis is based on the **Fourier series**, named after the mathematician Joseph Fourier. The Fourier series is a mathematical method to represent functions as an infinite series of sine and cosine terms. The functions in question can be real or complex valued.

Genetic algorithms are based on the biological theory of evolution. This type of algorithm is useful for searching and optimization.

GPUs (graphical processor units) are specialized circuits used to display graphics efficiently. Recently, GPUs have been used to perform massively parallel computations (for instance, to train neural networks).

Hadoop Distributed File System (HDFS) is the storage component of the Hadoop framework for big data. HDFS is a distributed filesystem, which spreads data on multiple systems, and is inspired by Google File System, used by Google for its search engine.

A **hive plot** is a visualization technique for plotting network graphs. In hive plots, we draw edges as curved lines. We group nodes by some property and display them on radial axes.

Influence plots take into account residuals, influence, and leverage for individual data points, similar to bubble plots. The size of the residuals is plotted on the vertical axis and can indicate that a data point is an outlier.

Jackknifing is a deterministic algorithm to estimate confidence intervals. It falls under the family of resampling algorithms. Usually, we generate new datasets under the jackknifing algorithm by deleting one value (we can also delete two or more values).

JSON (JavaScript Object Notation) is a data format. In this format, data is written down using JavaScript notation. JSON is more succinct than other data formats, such as XML.

K-fold cross-validation is a form of cross-validation involving k (a small integer number) random data partitions called **folds**. In k iterations, each fold is used once for validation, and the rest of the data is used for training. The results of the iterations can be combined at the end.

Linear discriminant analysis (LDA) is an algorithm that looks for a linear combination of features in order to distinguish between classes. It can be used for classification or dimensionality reduction by projecting to a lower-dimensional subspace.

Learning curve: A way to visualize the behavior of a learning algorithm. It is a plot of training and test scores for a range of training data sizes.

Logarithmic plots (or log plots) are plots that use a logarithmic scale. This type of plot is useful when the data varies a lot, because they display orders of magnitude.

Logistic regression is a type of a classification algorithm. This algorithm can be used to predict probabilities associated with a class or an event occurring. Logistic regression is based on the **logistic function**, which has output values in the range from zero to one, just like in probabilities. The logistic function can therefore be used to transform arbitrary values into probabilities.

The **Lomb-Scargle periodogram** is a frequency spectrum estimation method that fits sines to data, and it is frequently used with unevenly sampled data. The method is named after Nicholas R. Lomb and Jeffrey D. Scargle.

The **Matthews correlation coefficient (MCC)** or **phi coefficient** is an evaluation metric for binary classification invented by Brian Matthews in 1975. The MCC is a correlation coefficient for target and predictions and varies between -1 and 1 (best agreement).

Memory leaks are a common issue of computer programs, which we can find by performing memory profiling. Leaks occur when we don't release memory that is not needed.

Moore's law is the observation that the number of transistors in a modern computer chip doubles every 2 years. This trend has continued since Moore's law was formulated, around 1970. There is also a second Moore's law, which is also known as Rock's law. This law states that the cost of R&D and manufacturing of integrated circuits increases exponentially.

Named-entity recognition (NER) tries to detect names of persons, organizations, locations, and others in text. Some NER systems are almost as good as humans, but it is not an easy task. Named entities usually start with upper case, such as Ivan. We should therefore not change the case of words when applying NER.

Object-relational mapping (ORM): A software architecture pattern for translation between database schemas and object-oriented programming languages.

Open Computing Language (OpenCL), initially developed by Apple Inc., is an open technology standard for programs, which can run on a variety of devices, including CPUs and GPUs that are available on commodity hardware.

OpenCV (Open Source Computer Vision) is a library for computer vision created in 2000 and currently maintained by Itseez. OpenCV is written in C++, but it also has bindings to Python and other programming languages.

Opinion mining or **sentiment analysis** is a research field with the goal of efficiently finding and evaluating opinions and sentiment in text.

Principal component analysis (PCA), invented by Karl Pearson in 1901, is an algorithm that transforms data into uncorrelated orthogonal features called principal components. The **principal components** are the eigenvectors of the covariance matrix.

The **Poisson distribution** is named after the French mathematician Poisson, who published it in 1837. The Poisson distribution is a discrete distribution usually associated with counts for a fixed interval of time or space.

Robust regression is designed to deal better with outliers in data than ordinary regression. This type of regression uses special robust estimators.

Scatter plot: A two-dimensional plot showing the relationship between two variables in a Cartesian coordinate system. The values of one variable are represented on one axis, and the other variable is represented by the other axis. We can quickly visualize correlation this way.

In the **shared-nothing architecture**, computing nodes don't share memory or files. The architecture is therefore totally decentralized, with completely independent nodes. The obvious advantage is that we are not dependent on any one node. The first commercial shared-nothing databases were created in the 1980s.

Signal processing is a field of engineering and applied mathematics that deals with the analysis of analog and digital signals corresponding to variables that vary with time.

Structured Query Language (SQL) is a specialized language for relational database querying and manipulation. This includes creating, inserting rows in, and deleting tables.

Short-time Fourier transform (STFT): The STFT splits a signal in the time domain into equal parts and then applies the FFT to each segment.

Stop words: Common words with low information value. Stop words are usually removed before analyzing text. Although filtering stop words is common practice, there is no standard definition of stop words.

The **Spearman rank correlation** uses ranks to correlate two variables with the Pearson correlation. Ranks are the positions of values in sorted order. Items with equal values get a rank, which is the average of their positions. For instance, if we have two items of equal value assigned positions 2 and 3, the rank is 2.5 for both items.

Spectral clustering is a clustering technique that can be used to segment images.

The **star schema** is a database pattern that facilitates reporting. Star schemas are appropriate for the processing of events such as website visits, ad clicks, or financial transactions. Event information (metrics such as temperature or purchase amount) is stored in fact tables linked to much smaller-dimension tables. Star schemas are denormalized, which places the responsibility of integrity checks on the application code. For this reason, we should only write to the database in a controlled manner.

Term frequency-inverse document frequency (tf-idf) is a metric measuring the importance of a word in a corpus. It is composed of a term frequency number and an inverse document frequency number. The term frequency counts the number of times a word occurs in a document. The inverse document frequency counts the number of documents in which the word occurs and takes the inverse of the number.

Time series: An ordered list of data points, starting with the oldest measurements. Usually, each data point has a related timestamp.

Violin plots combine box plots and kernel-density plots or histograms in one type of plot.

Winsorising is a technique to deal with outliers and is named after Charles Winsor. In effect, Winsorising clips outliers to given percentiles in a symmetric fashion.

B

Function Reference

This appendix is a short reference of functions not meant as exhaustive documentation, but as an extra aid in case you are temporarily unable to look up the documentation. These functions are organized by package for various libraries.

IPython

The following displays a Python object in all frontends:

```
IPython.core.display.display(*objs, **kwargs)
```

The following renders HTML content:

```
IPython.display.HTML(TextDisplayObject)
```

The following displays interactive widgets connected to a function. The first parameter is expected to be a function:

```
IPython.html.widgets.interaction.interact (__interact_f=None,  
**kwargs)
```

The following arguments to this function are widget abbreviations passed in as keyword arguments, which build a group of interactive widgets tied to `__interact_f` and places the group in a container:

```
IPython.html.widgets.interaction.interactive (__interact_f, **kwargs)
```

Matplotlib

The following method is used to get or set axis properties. For example, `axis('off')` turns off the axis lines and labels:

```
matplotlib.pyplot.axis(*v, **kwargs)
```

The following argument creates a new figure:

```
matplotlib.pyplot.figure(num=None, figsize=None, dpi=None,
facecolor=None, edgecolor=None, frameon=True, FigureClass=<class
'matplotlib.figure.Figure'>, **kwargs)
```

The following argument turns the plot grids on or off:

```
matplotlib.pyplot.grid(b=None, which='major', axis='both', **kwargs)
```

The following argument plots a histogram:

```
matplotlib.pyplot.hist(x, bins=10, range=None, normed=False,
weights=None, cumulative=False, bottom=None, histtype='bar',
align='mid', orientation='vertical', rwidth=None, log=False,
color=None, label=None, stacked=False, hold=None, **kwargs)
```

The following displays an image for array-like data:

```
matplotlib.pyplot.imshow(X, cmap=None, norm=None, aspect=None,
interpolation=None, alpha=None, vmin=None, vmax=None, origin=None,
extent=None, shape=None, filternorm=1, filterrad=4.0, imlim=None,
resample=None, url=None, hold=None, **kwargs)
```

The following shows a legend at an optionally specified location (for instance, `plt.legend(loc='best')`):

```
matplotlib.pyplot.legend(*args, **kwargs)
```

The following argument creates a two-dimensional plot with single or multiple x, y pairs and corresponding optional format string:

```
matplotlib.pyplot.plot(*args, **kwargs)
```

The following creates a scatter plot of two arrays:

```
matplotlib.pyplot.scatter(x, y, s=20, c='b', marker='o', cmap=None,
norm=None, vmin=None, vmax=None, alpha=None, linewidths=None,
verts=None, hold=None, **kwargs)
```

The following argument displays a plot:

```
matplotlib.pyplot.show(*args, **kw)
```

The following argument creates subplots given the row number, column number, and index number of the plot. All these numbers start at one. For instance, `plt.subplot(221)` creates the first subplot in a two-by-two grid:

```
matplotlib.pyplot.subplot(*args, **kwargs)
```

The following argument puts a title on the plot:

```
matplotlib.pyplot.title(s, *args, **kwargs)
```

NumPy

The following creates a NumPy array with evenly spaced values within a specified range:

```
numpy.arange([start,] stop[, step,], dtype=None)
```

The following argument returns the indices that would sort the input array:

```
numpy.argsort(a, axis=-1, kind='quicksort', order=None)
```

The following creates a NumPy array from an array-like sequence, such as a Python list:

```
numpy.array(object, dtype=None, copy=True, order=None, subok=False,
ndmin=0)
```

The following argument calculates the dot product of two arrays:

```
numpy.dot(a, b, out=None)
```

The following argument returns the identity matrix:

```
numpy.eye(N, M=None, k=0, dtype=<type 'float'>)
```

The following argument loads NumPy arrays or pickled objects from `.npy`, `.npz` or pickles. A memory-mapped array is stored in the filesystem and doesn't have to be completely loaded in memory. This is especially useful for large arrays:

```
numpy.load(file, mmap_mode=None)
```

The following argument loads data from a text file into a NumPy array:

```
numpy.loadtxt(fname, dtype=<type 'float'>, comments='#',
delimiter=None, converters=None, skiprows=0, usecols=None,
unpack=False, ndmin=0)
```

The following calculates the arithmetic mean along the given axis:

```
numpy.mean(a, axis=None, dtype=None, out=None, keepdims=False)
```

The following argument calculates the median along the given axis:

```
numpy.median(a, axis=None, out=None, overwrite_input=False)
```

The following creates a NumPy array of specified shape and data type, containing ones:

```
numpy.ones(shape, dtype=None, order='C')
```

The following performs a least squares polynomial fit:

```
numpy.polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False)
```

The following changes the shape of a NumPy array:

```
numpy.reshape(a, newshape, order='C')
```

The following argument saves a NumPy array to a file in the NumPy .npy format:

```
numpy.save(file, arr)
```

The following argument saves a NumPy array to a text file:

```
numpy.savetxt(fname, X, fmt='%.18e', delimiter=' ', newline='\n',
header='', footer='', comments='# ')
```

The following argument sets printing options:

```
numpy.set_printoptions(precision=None, threshold=None, edgeitems=None,
linewidth=None, suppress=None, nanstr=None, infstr=None,
formatter=None)
```

The following argument returns the standard deviation along the given axis:

```
numpy.std(a, axis=None, dtype=None, out=None, ddof=0, keepdims=False)
```

The following selects array elements from input arrays based on a Boolean condition:

```
numpy.where(condition, [x, y])
```

The following creates a NumPy array of specified shape and data type, containing zeros:

```
numpy.zeros(shape, dtype=float, order='C')
```

pandas

The following creates a fixed frequency datetime index:

```
pandas.date_range(start=None, end=None, periods=None, freq='D',
tz=None, normalize=False, name=None, closed=None)
```

The following argument generate various summary statistics, ignoring NaN values:

```
pandas.DataFrame.describe(self, percentile_width=None,  
percentiles=None, include=None, exclude=None)
```

The following creates a DataFrame object from a dictionary of array-like objects or dictionaries:

```
pandas.DataFrame.from_dict(data, orient='columns', dtype=None)
```

The following argument finds NaN and None values:

```
pandas.isnull(obj)
```

The following argument merges DataFrame objects with a database-like join on columns or indices:

```
pandas.merge(left, right, how='inner', on=None, left_on=None,  
right_on=None, left_index=False, right_index=False, sort=False,  
suffixes=('_x', '_y'), copy=True)
```

The following creates a DataFrame object from a CSV file:

```
pandas.read_csv(filepath_or_buffer, sep=',', dialect=None,  
compression=None, doublequote=True, escapechar=None, quotechar='',  
quoting=0, skipinitialspace=False, lineterminator=None,  
header='infer', index_col=None, names=None, prefix=None,  
skiprows=None, skipfooter=None, skip_footer=0, na_values=None, na_  
fvalues=None, true_values=None, false_values=None, delimiter=None,  
converters=None, dtype=None, usecols=None, engine='c', delim_  
whitespace=False, as_recarray=False, na_filter=True, compact_  
ints=False, use_unsigned=False, low_memory=True, buffer_lines=None,  
warn_bad_lines=True, error_bad_lines=True, keep_default_na=True,  
thousands=Nment=None, decimal='.', parse_dates=False, keep_date_  
col=False, dayfirst=False, date_parser=None, memory_map=False,  
nrows=None, iterator=False, chunkszie=None, verbose=False,  
encoding=None, squeeze=False, mangle_dupe_cols=True, tupleize_  
cols=False, infer_datetime_format=False)
```

Scikit-learn

The following argument turns seed into a numpy.random.RandomState instance:

```
sklearn.utils.check_random_state(seed)
```

The following performs a grid search over given hyperparameter values for an estimator:

```
sklearn.grid_search.GridSearchCV(estimator, param_grid, scoring=None,  
fit_params=None, n_jobs=1, iid=True, refit=True, cv=None, verbose=0,  
pre_dispatch='2*n_jobs', error_score='raise')
```

The following argument splits arrays into random train and test sets:

```
sklearn.cross_validation.train_test_split(*arrays, **options)
```

The following returns the accuracy classification score:

```
sklearn.metrics.accuracy_score(y_true, y_pred, normalize=True, sample_weight=None)
```

SciPy

The following computes the relative maxima of data:

```
scipy.signal.argrelextrema(data, axis=0, order=1, mode='clip')
```

The following argument calculates the kurtosis of a dataset:

```
scipy.stats.kurtosis(a, axis=0, fisher=True, bias=True)
```

The following applies a median filter on an array:

```
scipy.signal.medfilt(volume, kernel_size=None)
```

The following argument calculates the skewness of a data set:

```
scipy.stats.skew(a, axis=0, bias=True)
```

Seaborn

The following argument plots a univariate distribution of observations:

```
seaborn.distplot(a, bins=None, hist=True, kde=True, rug=False,
fit=None, hist_kws=None, kde_kws=None, rug_kws=None, fit_kws=None,
color=None, vertical=False, norm_hist=False, axlabel=None, label=None,
ax=None)
```

The following argument plots tabular data as a color-encoded matrix:

```
seaborn.heatmap(data, vmin=None, vmax=None, cmap=None, center=None,
robust=False, annot=False, fmt='.2g', annot_kws=None, linewidths=0,
linecolor='white', cbar=True, cbar_kws=None, cbar_ax=None,
square=False, ax=None, xticklabels=True, yticklabels=True, mask=None,
**kwargs)
```

The following argument plots data and the corresponding linear regression model fit:

```
seaborn.regplot(x, y, data=None, x_estimator=None, x_bins=None, x_ci='ci', scatter=True, fit_reg=True, ci=95, n_boot=1000, units=None, order=1, logistic=False, lowess=False, robust=False, logx=False, x_partial=None, y_partial=None, truncate=False, dropna=True, x_jitter=None, y_jitter=None, label=None, color=None, marker='o', scatter_kws=None, line_kws=None, ax=None)
```

The following argument restores all matplotlib RC parameters to the default settings:

```
seaborn.reset_defaults()
```

The following argument restores all matplotlib RC parameters to the original settings:

```
seaborn.reset_orig()
```

The following argument plots the residuals of a linear regression:

```
seaborn.residplot(x, y, data=None, lowess=False, x_partial=None, y_partial=None, order=1, robust=False, dropna=True, label=None, color=None, scatter_kws=None, line_kws=None, ax=None)
```

The following argument sets aesthetic parameters:

```
seaborn.set(context='notebook', style='darkgrid', palette='deep', font='sans-serif', font_scale=1, color_codes=False, rc=None)
```

Statsmodels

The following argument downloads and returns the R dataset from the Internet:

```
statsmodels.api.datasets.get_rdataset(dataname, package='datasets', cache=False)
```

The following argument plots a Q-Q plot:

```
statsmodels.api.qqplot(data, dist, distargs=(), a=0, loc=0, scale=1, fit=False, line=None, ax=None)
```

The following argument creates an ANOVA table for one or more fitted linear models:

```
statsmodels.stats.anova.anova_lm()
```


C

Online Resources

The following is a short list of resources including presentations, links to documentation, freely available IPython Notebooks, and data.

IPython notebooks and open data

For more information on IPython notebooks and open data, you can refer to the following:

- ▶ Data science Python notebooks available at <https://github.com/donnemartin/data-science-ipython-notebooks> (retrieved January 2016)
- ▶ A collection of tutorials and examples for solving and understanding machine learning and pattern classification tasks available at https://github.com/rasbt/pattern_classification (retrieved January 2016)
- ▶ Awesome public datasets available at <https://github.com/caesar0301/awesome-public-datasets> (retrieved January 2016)
- ▶ UCI machine learning datasets available at <https://archive.ics.uci.edu/ml/datasets.html> (retrieved January 2016)
- ▶ Gallery of interesting IPython notebooks available at <https://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks> (retrieved January 2016)

Mathematics and statistics

- ▶ Linear algebra tutorials from Khan Academy available at <https://www.khanacademy.org/math/linear-algebra> (retrieved January 2016)
- ▶ Probability and statistics tutorials from Khan Academy at <https://www.khanacademy.org/math/probability> (retrieved January 2016)
- ▶ Coursera course on linear algebra, which uses Python, available at <https://www.coursera.org/course/matrix> (retrieved January 2016)
- ▶ *Introduction to probability* by Harvard University, available at <https://itunes.apple.com/us/course/statistics-110-probability/id502492375> (retrieved January 2016)
- ▶ The statistics wikibook at <https://en.wikibooks.org/wiki/Statistics> (retrieved January 2016)
- ▶ *Electronic Statistics Textbook*. Tulsa, OK: StatSoft. WEB: <http://www.statsoft.com/textbook/> (retrieved January 2016)

Presentations

- ▶ *Statistics for hackers* by Jake van der Plas, available at <https://speakerdeck.com/jakevdp/statistics-for-hackers> (retrieved January 2016)
- ▶ *Explore Data: Data Science + Visualization* by Roelof Pieters, available at <http://www.slideshare.net/roelofp/explore-data-data-science-visualization> (retrieved January 2016)
- ▶ *High Performance Python (1.5hr) Tutorial at EuroSciPy 2014* by Ian Ozsváld, available at <https://speakerdeck.com/ianozsvald/high-performance-python-1-dot-5hr-tutorial-at-euroscipy-2014> (retrieved January 2016)
- ▶ *Mastering Linked Data* by Valerio Maggio, available at <https://speakerdeck.com/valeriomaggio/mastering-linked-data-with-python-at-pydata-berlin-2014> (retrieved January 2016)
- ▶ *Fast Data Analytics with Spark and Python* by Benjamin Bengfort, available at <http://www.slideshare.net/BenjaminBengfort/fast-data-analytics-with-spark-and-python> (retrieved January 2016)
- ▶ *Social network analysis with Python* by Benjamin Bengfort, available at <http://www.slideshare.net/BenjaminBengfort/social-network-analysis-with-python> (retrieved January 2016)
- ▶ SciPy 2015 conference list of talks, available at <https://www.youtube.com/playlist?list=PLYx7XA2nY5Gcpabmu61kKcToLz0FapmHu> (retrieved January 2016)

- ▶ Statistical inference in Python, available at <https://sites.google.com/site/pyinference/home/scipy-2015> (retrieved January 2016)
- ▶ *Ibis: Scaling Python Analytics on Hadoop and Impala* by Wes McKinney, available at <http://www.slideshare.net/wesm/ibis-scaling-python-analytics-on-hadoop-and-impala> (retrieved January 2016)
- ▶ *PyData: The Next Generation* by Wes McKinney, available at <http://www.slideshare.net/wesm/pydata-the-next-generation> (retrieved January 2016)
- ▶ *Python as the Zen of Data Science* by Travis Oliphant, available at <http://www.slideshare.net/teoliphant/python-as-the-zen-of-data-science> (retrieved January 2016)
- ▶ *PyData Texas 2015 Keynote* by Peter Wang, available at <http://www.slideshare.net/misterwang/pydata-texas-2015-keynote> (retrieved January 2016)
- ▶ *What's new in scikit-learn 0.17* by Andreas Mueller, available at <http://www.slideshare.net/AndreasMueller7/whats-new-in-scikitlearn-017> (retrieved January 2016)
- ▶ *Tree models with Scikit-learn: Great models with little assumptions* by Gilles Loupe, available at <http://www.slideshare.net/glouppe/slides-46767187> (retrieved January 2016)
- ▶ *Mining Social Web APIs with IPython Notebook (Data Day Texas 2015)* by Matthew Russell, available at <http://www.slideshare.net/ptwobrussell/mining-social-web-ap-iswithipythonnotebookddtx2015> (retrieved January 2016)
- ▶ *Docker for data science* by Calvin Giles, available at <http://www.slideshare.net/CalvinGiles/docker-for-data-science> (retrieved January 2016)
- ▶ *10 more lessons learned from building Machine Learning systems* by Xavier Amatriain, available at <http://www.slideshare.net/xamat/10-more-lessons-learned-from-building-machine-learning-systems> (retrieved January 2016)
- ▶ *IPython & Project Jupyter: A language-independent architecture for open computing and data science* by Fernando Perez, available at <https://speakerdeck.com/fperez/ipython-and-project-jupyter-a-language-independent-architecture-for-open-computing-and-data-science> (retrieved January 2016)
- ▶ *Scikit-learn for easy machine learning: the vision, the tool, and the project* by Gael Varoquaux, available at <http://www.slideshare.net/GaelVaroquaux/slides-48793181> (retrieved January 2016)
- ▶ *Big Data, Predictive Modeling and tools* by Olivier Grisel, available at <https://speakerdeck.com/ogrisel/big-data-predictive-modeling-and-tools> (retrieved January 2016)

- ▶ *Data Science Python Ecosystem* by Christine Doig, available at <https://speakerdeck.com/chdoig/data-science-python-ecosystem> (retrieved January 2016)
- ▶ *New Trends in Storing Large Data Silos in Python* by Francesc Alted, available at <https://speakerdeck.com/francescalted/new-trends-in-storing-large-data-silos-in-python> (retrieved January 2016)
- ▶ *Distributed Computing on your Cluster with Anaconda - Webinar 2015* by Continuum Analytics, available at <http://www.slideshare.net/continuumio/distributed-computing-on-your-cluster-with-anaconda-webinar-2015> (retrieved January 2016)

D

Tips and Tricks for Command-Line and Miscellaneous Tools

In this book we used various tools, such as the IPython notebook and Unix shell commands. We have a short list of tips, which is not meant to be exhaustive. For working with databases, I recommend the DbVisualiser software available at <https://www.dbvis.com/> (retrieved January 2016). It supports all the major database products and operating systems. Also, I like to use text expanders in a desktop environment.

IPython notebooks

I explained a minimal workflow for notebooks. Also, I made simple IPython widgets, which were used throughout the book, so I will describe them here. To run the IPython notebook code, follow these steps:

1. Start the IPython notebook either with your GUI or with the following command:
`$ jupyter notebook`
2. Run the code either cell by cell or in one run.

I made a widget that sets some of the matplotlib properties. The settings are stored in the `dautil.json` file in the current folder. These files should also be part of the code bundle.

The other IPython widget helps with setting up subplots. It takes care of setting titles, legends, and labels. I consider these strings to be configuration and, therefore, store them in the `dautil.json` files too.

Command-line tools

Some of these tools have GUI alternatives that are not always mentioned. In my opinion, it is a good idea to learn about using command-line tools even if you decide afterwards that you prefer the GUI options. Linux is one of the many popular operating systems that support CLI. You can find good documentation about Linux tools at <http://tldp.org/> (retrieved January 2016). Most information on the website is generic and useful on other operating systems as well, such as OS X.

Navigation is often cumbersome in the CLI world. I find bashmarks a good tool to help you with that. You can find bashmarks at <https://github.com/huyng/bashmarks> (retrieved January 2016). The steps to install bashmarks are as follows:

1. Type the following in a terminal:

```
$ git clone git://github.com/huyng/bashmarks.git
```

2. Now, type this in the terminal:

```
$ cd bashmarks
```

3. Next, type the following:

```
$ make install
```

4. Source either in a configuration file or just the current session:

```
$ source ~/.local/bin/bashmarks.sh
```

The following table lists the bashmarks commands:

Command	Description
s <bookmark_name>	This saves the current directory as bookmark_name
g <bookmark_name>	This goes to the directory associated with bookmark_name
p <bookmark_name>	This prints the directory associated with bookmark_name
d <bookmark_name>	This deletes the bookmark
l	This lists all available bookmarks

The alias command

The alias command allows you to define a short mnemonic for a long command. For instance, we can define the following alias to start the IPython server when we type ipnb:

```
$ alias ipnb='ipython notebook'
```

We can define aliases for the current session only, but usually we define aliases in the `.bashrc` startup file (the dot in the file name means that it is a hidden file) found in the home directory. If you find yourself having many aliases, it may be useful to create a file containing all the aliases. You can then source this file from `.bashrc`.

Command-line history

The command-line history is a mechanism to minimize the number of keystrokes. You can read more about it at <http://www.tldp.org/LDP/GNU-Linux-Tools-Summary/html/x1712.htm> (retrieved January 2016).

To simply execute the last run command, type the following again:

```
$ !!
```

Depending on which shell mode (`vi` or `emacs`) you are in, you may prefer other ways to navigate the history. The up and down arrows on your keyboard should also let you navigate history.

A common use case is to search for a long command we executed in the past and run it again. We can search through history, as follows:

```
$ history|grep <search for something>
```

You can of course shorten this using the aliasing mechanism or with a desktop text expander. The search gives a list of commands with numbers ranked in chronological order. You can execute, for instance, the command numbered 328, as follows:

```
$ !328
```

If, for example, you wish to execute the last command that started with `python`, type the following:

```
$ !python
```

Reproducible sessions

Chapter 1, Laying the Foundation for Reproducible Data Analysis explained the value of reproducible analysis. In this context, we have the `script` command, which is a way to capture commands and the output of a session.

Docker tips

Docker is a great technology, but we have to be careful not to make our images too big and to remove image files when possible. The `docker-clean` script at <https://gist.github.com/michaelneale/1366325a7737c4cb80b0> (retrieved January 2016) helps reclaim space.

I found it useful to have an install script, which is just a regular shell script, and I added it to the `Dockerfile` as follows:

```
ADD install.sh /root/install.sh
```

Python creates `__pycache__` directories for the purpose of optimization (we can disable this option in various ways). These are not strictly needed and can be easily removed as follows:

```
find /opt/conda -name __pycache__ -depth -exec rm -rf {} \;
```

Anaconda puts a lot of files in its `pkgs` directory, which we can remove as follows:

```
rm -r /opt/conda/pkgs/*
```

Some people recommend removing test code; however, in certain rare cases, the non-test code depends on the test code. Also, it is useful to have the test code just in case.

There are some gotchas to be aware of when working with Docker. For instance, we have to set the `PATH` environment variable as follows:

```
export PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:${PATH}
```

For Python scripts, we also need to set the language settings as follows:

```
ENV LANG=C.UTF-8
```

It is generally recommended to specify the package version when you install software with `pip` or `conda`, such as like this:

```
$ conda install scipy=0.15.0
$ pip install scipy==0.15.0
```

When installing with `conda`, it is also recommended that you install multiple packages at once in order to avoid installing multiple versions of common dependencies:

```
$ conda install scipy=0.15.0 curl=7.26.0
```

My Docker setup for the main Docker repository consists of a Dockerfile script and an install script (`install.sh`). The contents of the Dockerfile are as follows:

```
FROM continuumio/miniconda3

ADD install.sh /root/install.sh
RUN sh -x /root/install.sh

ENV LANG=C.UTF-8
```

I execute the install script with the `-x` switch, which gives more verbose output.

The contents of `install.sh` are as follows:

```
export PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:$PATH
apt-get install -y libgfortran3
conda config --set always_yes True
conda install beautiful-soup bokeh=0.9.1 execnet=1.3.0 \
fastcache=1.0.2 \
joblib=0.8.4 jsonschema ipython=3.2.1 lxml mpmath=0.19 \
networkx=1.9.1 nltk=3.0.2 numba=0.22.1 numexpr=2.3.1 \
pandas=0.16.2 pyzmq scipy=0.16.0 seaborn=0.6.0 \
sqlalchemy=0.9.9 statsmodels=0.6.1 terminado=0.5 tornado
conda install matplotlib=1.5.0 numpy=1.10.1 scikit-learn=0.17
pip install dautil==0.0.1a29
pip install hiveplot==0.1.7.4
pip install landslide==1.1.3
pip install LiveStats==1.0
pip install mpld3==0.2
pip install pep8==1.6.2
pip install requests-cache==0.4.10
pip install tabulate==0.7.5

find /opt/conda -name __pycache__ -depth -exec rm -rf {} \;
rm -r /opt/conda/pkgs/*

mkdir -p /.cache/dautil/log
mkdir -p /.local/share/dautil
```


Module 3

Mastering Python Data Analysis

Become an expert at using Python for advanced statistical analysis of data using real-world examples

1

Tools of the Trade

This chapter gives you an overview of the tools available for data analysis in Python, with details concerning the Python packages and libraries that will be used in this book. A few installation tips are given, and the chapter concludes with a brief example. We will concentrate on how to read data files, select data, and produce simple plots, instead of delving into numerical data analysis.

Before you start

We assume that you have familiarity with Python and have already developed and run some scripts or used Python interactively, either in the shell or on another interface, such as the Jupyter Notebook (formerly known as the **IPython notebook**). Hence, we also assume that you have a working installation of Python. In this book, we assume that you have installed Python 3.4 or later.

We also assume that you have developed your own workflow with Python, based on needs and available environment. To follow the examples in this book, you are expected to have access to a working installation of Python 3.4 or later. There are two alternatives to get started, as outlined in the following list:

- Use a Python installation from scratch. This can be downloaded from <https://www.python.org>. This will require a separate installation for each of the required libraries.
- Install a prepackaged distribution containing libraries for scientific and data computing. Two popular distributions are Anaconda Scientific Python (<https://store.continuum.io/cshop/anaconda>) and Enthought distribution (<https://www.enthought.com>).



Even if you have a working Python installation, you might want to try one of the prepackaged distributions. They contain a well-rounded collection of packages and modules suitable for data analysis and scientific computing. If you choose this path, all the libraries in the next list are included by default.

We also assume that you have the libraries in the following list:

- **numpy** and **scipy**: These are available at <http://www.scipy.org>. These are the essential Python libraries for computational work. NumPy defines a fast and flexible array data structure, and SciPy has a large collection of functions for numerical computing. They are required by some of the libraries mentioned in the list.
- **matplotlib**: This is available at <http://matplotlib.org>. It is a library for interactive graphics built on top of NumPy. I recommend versions above 1.5, which is what is included in Anaconda Python by default.
- **pandas**: This is available at <http://pandas.pydata.org>. It is a Python data analysis library. It will be used extensively throughout the book.
- **pymc**: This is a library to make Bayesian models and fitting in Python accessible and straightforward. It is available at <http://pymc-devs.github.io/pymc/>. This package will mainly be used in Chapter 6, *Bayesian Methods*, of this book.
- **scikit-learn**: This is available at <http://scikit-learn.org>. It is a library for machine learning in Python. This package is used in Chapter 7, *Supervised and Unsupervised Learning*.
- **IPython**: This is available at <http://ipython.org>. It is a library providing enhanced tools for interactive computations in Python from the command line.
- **Jupyter**: This is available at <https://jupyter.org/>. It is the notebook interface working on top of IPython (and other programming languages). Originally part of the IPython project, the notebook interface is a web-based platform for computational and data science that allows easy integration of the tools that are used in this book.

Notice that each of the libraries in the preceding list may have several dependencies, which must also be separately installed. To test the availability of any of the packages, start a Python shell and run the corresponding `import` statement. For example, to test the availability of NumPy, run the following command:

```
import numpy
```

If NumPy is not installed in your system, this will produce an error message. An alternative approach that does not require starting a Python shell is to run the command line:

```
python -c 'import numpy'
```

We also assume that you have either a programmer's editor or Python IDE. There are several options, but at the basic level, any editor capable of working with unformatted text files will do.

Using the notebook interface

Most examples in this book will use the Jupyter Notebook interface. This is a browser-based interface that integrates computations, graphics, and other forms of media. Notebooks can be easily shared and published, for example, <http://nbviewer.ipython.org/> provides a simple publication path.

It is not, however, absolutely necessary to use the Jupyter interface to run the examples in this book. We strongly encourage, however, that you at least experiment with the notebook and its many features. The Jupyter Notebook interface makes it possible to mix formatted, descriptive text with code cells that evaluate at the same time. This feature makes it suitable for educational purposes, but it is also useful for personal use as it makes it easier to add comments and share partial progress before writing a full report. We will sometimes refer to a Jupyter Notebook as just *a notebook*.

To start the notebook interface, run the following command line from the shell or Anaconda command prompt:

```
jupyter notebook
```

The notebook server will be started in the directory where the command is issued. After a while, the notebook interface will appear in your default browser. Make sure that you are using a standards-compliant browser, such as Chrome, Firefox, Opera, or Safari. Once the Jupyter dashboard shows on the browser, click on the **New** button on the upper-right side of the page and select **Python 3**. After a few seconds, a new notebook will open in the browser. A useful place to learn about the notebook interface is <http://jupyter.org>.

Imports

There are some modules that we will need to load at the start of every project. Assuming that you are running a Jupyter Notebook, the required imports are as follows:

```
%matplotlib inline  
import matplotlib.pyplot as plt  
import numpy as np  
import pandas as pd
```

Enter all the preceding commands in a single notebook cell and press *Shift + Enter* to run the whole cell. A new cell will be created when there is none after the one you are running; however, if you want to create one yourself, the menu or keyboard shortcut *Ctrl +M +A/B* is handy (*A* for above, *B* for below the current cell). In Appendix, *More on Jupyter Notebook and matplotlib Styles*, we cover some of the keyboard shortcuts available and installable extensions (that is, plugins) for Jupyter Notebook.

The statement `%matplotlib inline` is an example of Jupyter Notebook magic and sets up the interface to display plots inline, that is, embedded in the notebook. This line is not needed (and causes an error) in scripts. Next, optionally, enter the following commands:

```
import os  
plt.style.use(os.path.join(os.getcwd(), 'mystyle.mplstyle'))
```

As before, run the cell by pressing *Shift +Enter*. This code has the effect of selecting matplotlib stylesheet `mystyle.mplstyle`. This is a custom style sheet that I created, which resides in the same folder as the notebook. It is a rather simple example of what can be done; you can modify it to your liking. As we gain experience in drawing figures throughout the book, I encourage you to play around with the settings in the file. There are also built-in styles that you can by typing `plt.style.available` in a new cell.

This is it! We are all set to start the fun part!

An example using the Pandas library

The purpose of this example is to check whether everything is working in your installation and give a flavor of what is to come. We concentrate on the Pandas library, which is the main tool used in Python data analysis.

We will use the MovieTweetings 50K movie ratings dataset, which can be downloaded from <https://github.com/sidooms/MovieTweetings>. The data is from the study MovieTweetings: a Movie Rating Dataset Collected From Twitter – by Dooms, De Pessemier and Martens presented during Workshop on Crowdsourcing and Human Computation for Recommender Systems, CrowdRec at RecSys (2013). The dataset is spread in several text files, but we will only use the following two files:

- `ratings.dat`: This is a double colon-separated file containing the ratings for each user and movie
- `movies.dat`: This file contains information about the movies

To see the contents of these files, you can open them with a standard text editor. The data is organized in columns, with one data item per line. The meanings of the columns are described in the `README.md` file, distributed with the dataset. The data has a peculiar aspect: some of the columns use a double colon (`::`) character as a separator, while others use a vertical bar (`|`). This emphasizes a common occurrence with real-world data: we have no control on how the data is collected and formatted. For data stored in text files, such as this one, it is always a good strategy to open the file in a text editor or spreadsheet software to take a look at the data and identify inconsistencies and irregularities.

To read the ratings file, run the following command:

```
cols = ['user id', 'item id', 'rating', 'timestamp']
ratings = pd.read_csv('data/ratings.dat', sep='::',
                      index_col=False, names=cols,
                      encoding="UTF-8")
```

The first line of code creates a Python list with the column names in the dataset. The next command reads the file, using the `read_csv()` function, which is part of Pandas. This is a generic function to read column-oriented data from text files. The arguments used in the call are as follows:

- `data/ratings.dat`: This is the path to file containing the data (this argument is required).
- `sep='::'`: This is the separator, a double colon character in this case.
- `index_col=False`: We don't want any column to be used as an index. This will cause the data to be indexed by successive integers, starting with 1.
- `names=cols`: These are the names to be associated with the columns.

The `read_csv()` function returns a DataFrame object, which is the Pandas data structure that represents tabular data. We can view the first rows of the data with the following command:

```
ratings[:5]
```

This will output a table, as shown in the following image:

	user id	item id	rating	timestamp
0	1	1074638	7	1365029107
1	1	1853728	8	1366576639
2	2	104257	8	1364690142
3	2	1259521	8	1364118447
4	2	1991245	7	1364117717

To start working with the data, let us find out how many times each rating appears in the table. This can be done with the following commands:

```
rating_counts = ratings['rating'].value_counts()  
rating_counts
```

The first line of code computes the counts and stores them in the `rating_counts` variable. To obtain the count, we first use the `ratings['rating']` expression to select the `rating` column from the table `ratings`. Then, the `value_counts()` method is called to compute the counts. Notice that we retype the variable name, `rating_counts`, at the end of the cell. This is a common notebook (and Python) idiom to print the value of a variable in the output area that follows each cell. In a script, it has no effect; we could have printed it with the `print` command, `(print(rating_counts))`, as well. The output is displayed in the following image:

```
8      12012
7      11063
9      7119
6      6373
10     6281
5      3399
4      1696
3      924
1      595
2      533
0      5
Name: rating, dtype: int64
```

Notice that the output is sorted according to the count values in descending order. The object returned by `value_counts` is of the Series type, which is the Pandas data structure used to represent one-dimensional, indexed, data. The Series objects are used extensively in Pandas. For example, the columns of a DataFrame object can be thought as Series objects that share a common index.

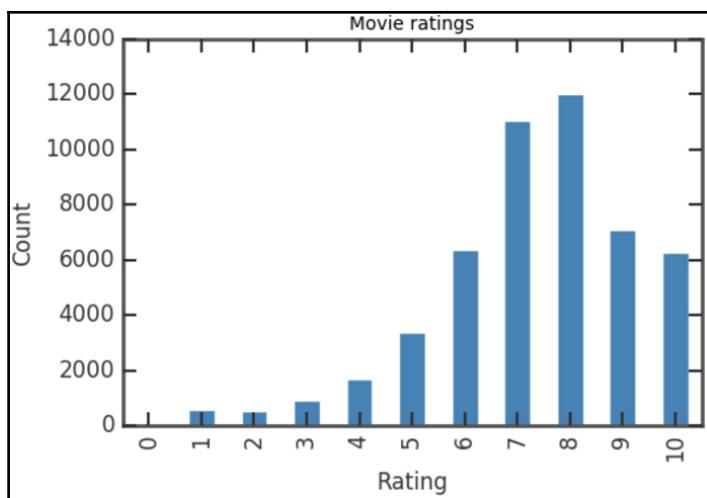
In our case, it makes more sense to sort the rows according to the ratings. This can be achieved with the following commands:

```
sorted_counts = rating_counts.sort_index()
sorted_counts
```

This works by calling the `sort_index()` method of the Series object, `rating_counts`. The result is stored in the `sorted_counts` variable. We can now get a quick visualization of the ratings distribution using the following commands:

```
sorted_counts.plot(kind='bar', color='SteelBlue')
plt.title('Movie ratings')
plt.xlabel('Rating')
plt.ylabel('Count')
```

The first line produces the plot by calling the `plot()` method for the `sorted_counts` object. We specify the `kind='bar'` option to produce a bar chart. Notice that we added the `color='SteelBlue'` option to select the color of the bars in the histogram. SteelBlue is one of the HTML5 color names (for example, http://matplotlib.org/examples/color/named_colors.html) available in matplotlib. The next three statements set the title, horizontal axis label, and vertical axis label respectively. This will produce the following plot:



The vertical bars show how many voters that have given a certain rating, covering all the movies in the database. The distribution of the ratings is not very surprising: the counts increase up to a rating of 8, and the count of 9–10 ratings is smaller as most people are reluctant to give the highest rating. If you check the values of the bar for each rating, you can see that it corresponds to what we had previously when printing the `rating_counts` object. To see what happens if you do not sort the ratings first, plot the `rating_counts` object, that is, run `rating_counts.plot(kind='bar', color='SteelBlue')` in a cell.

Let's say that we would like to know if the ratings distribution for a particular movie genre, say Crime Drama, is similar to the overall distribution. We need to cross-reference the ratings information with the movie information, contained in the movies.dat file. To read this file and store it in a Pandas DataFrame object, use the following command:

```
cols = ['movie id', 'movie title', 'genre']
movies = pd.read_csv('data/movies.dat', sep='::',
                     index_col=False, names=cols,
                     encoding="UTF-8")
```

Downloading the example code



Detailed steps to download the code bundle are mentioned in the Preface of this book. Please have a look.

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Mastering-Python-Data-Analysis>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

We are again using the `read_csv()` function to read the data. The column names were obtained from the `README.md` file distributed with the data. Notice that the separator used in this file is also a double colon, `::`. The first few lines of the table can be displayed with the command:

```
movies[:5]
```

Notice how the genres are indicated, clumped together with a vertical bar, `|`, as separator. This is due to the fact that a movie can belong to more than one genre. We can now select only the movies that are crime dramas using the following lines:

```
drama = movies[movies['genre']=='Crime|Drama']
```

Notice that this uses the standard indexing notation with square brackets, `movies[...]`. Instead of specifying a numeric or string index, however, we are using the Boolean `movies['genre']=='Crime|Drama'` expression as an index. To understand how this works, run the following code in a cell:

```
is_drama = movies['genre']=='Crime|Drama'
is_drama[:5]
```

This displays the following output:

```
0    True
1   False
2   False
3   False
4   False
Name: genre, dtype: bool
```

The `movies['genre']=='Crime|Drama'` expression returns a Series object, where each entry is either True or False, indicating whether the corresponding movie is a crime drama or not, respectively.

Thus, the net effect of the `drama = movies[movies['genre']=='Crime|Drama']` assignment is to select all the rows in the movies table for which the entry in the genre column is equal to Crime|Drama and store the result in the `drama` variable, which is an object of the DataFrame type.

All that we need is the `movie_id` column of this table, which can be selected with the following statement:

```
drama_ids = drama['movie id']
```

This, again, uses standard indexing with a string to select a column from a table.

The next step is to extract those entries that correspond to dramas from the `ratings` table. This requires yet another indexing trick. The code is contained in the following lines:

```
criterion = ratings['item id'].map(lambda x: (drama_ids==x).any())
drama_ratings = ratings[criterion]
```

The key to how this code works is the definition of the variable `criterion`. We want to look up each row of the `ratings` table and check whether the `item id` entry is in the `drama_ids` table. This can be conveniently done by the `map()` method. This method applies a function to all the entries of a Series object. In our example, the function is as follows:

```
lambda x: (drama_ids==x).any()
```

This function simply checks whether an item appears in `drama_ids`, and if it does, it returns `True`. The resulting object `criterion` will be a Series that contains the `True` value only in the rows that correspond to dramas. You can view the first rows with the following code:

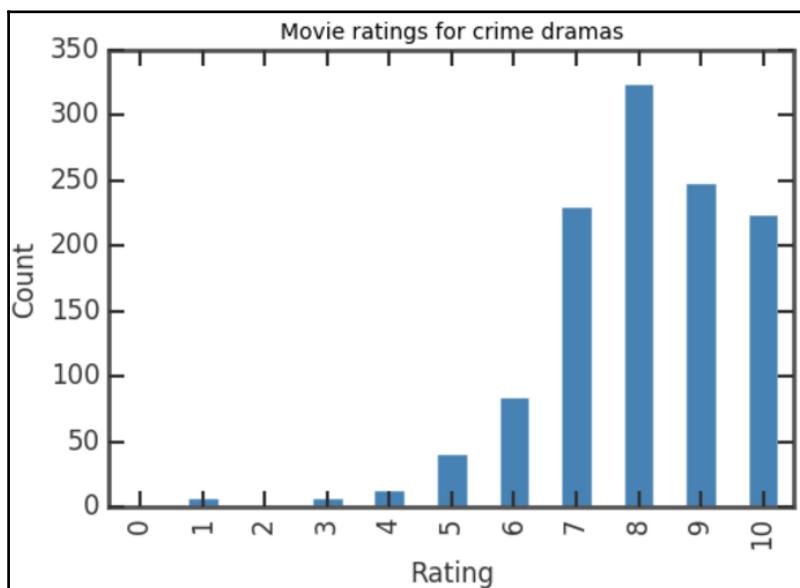
```
criterion[:10]
```

We then use the `criterion` object as an index to select the rows from the `ratings` table.

We are now done with selecting the data that we need. To produce a rate count and bar chart, we use the same commands as before. The details are in the following code, which can be run in a single execution cell:

```
rating_counts = drama_ratings['rating'].value_counts()  
sorted_counts = rating_counts.sort_index()  
sorted_counts.plot(kind='bar', color='SteelBlue')  
plt.title('Movie ratings for dramas')  
plt.xlabel('Rating')  
plt.ylabel('Count')
```

As before, this code first computes the counts, indexes them according to the ratings, and then produces a bar chart. This produces a graph that seems to be similar to the overall ratings distribution, as shown in the following figure:



Summary

In this chapter, we have seen what tools are available for data analysis in Python, reviewed issues related to installation and workflow, and considered a simple example that requires reading and manipulating data files.

In the next chapter, we will cover techniques to explore data graphically and numerically using some of the main tools provided by the Pandas module.

2

Exploring Data

When starting to work on a new dataset, it is essential to first get an idea of what conclusions can be drawn from the data. Before we can do things such as inference and hypothesis testing, we need to develop an understanding of what questions the data at hand can answer. This is the key to exploratory data analysis, which is the skill and science of developing intuition and identifying statistical patterns in the data. In this chapter, we will present graphical and numerical methods that help in this task. You will notice that there are no hard and fast rules of how to proceed at each step, but instead, we give recommendations on what techniques tend to be suitable in each case. The best way to develop the set of skills necessary to be an expert data explorer is to see lots of examples and, perhaps more importantly, work on our own datasets. More specifically, this chapter will cover the following topics:

- Performing the initial exploration and cleaning of data
- Drawing a histogram, kernel density estimate, probability, and box plot for univariate distributions
- Drawing scatterplots for bivariate relationships and giving an initial overview of various point estimates of the data, such as mean, standard deviation, and so on

Before starting through the examples in this chapter, start the Jupyter Notebook and run the same initial commands as mentioned in the previous chapter. Remember the directory where the notebook resides. The data folder for the examples needs to be stored in the same directory.

The General Social Survey

To present concrete data examples in this chapter, we will use the **General Social Survey (GSS)**. The GSS is a large survey of societal trends conducted by the **National Opinion Research Center (NORC**—<http://www3.norc.org>) at the University of Chicago. As this is a very complex dataset, we will work with a subset of the data, the compilation from the 2012 survey. With a size 5.5 MB, this is a small data size by the current standards, but still well-suited for the kind of exploration being illustrated in this chapter. (Smith, Tom W, Peter Marsden, Michael Hout, and Jibum Kim. General Social Surveys, 1972-2014 [machine-readable data file] /Principal Investigator, Tom W. Smith; Co-Principal Investigator, Peter V. Marsden; Co-Principal Investigator, Michael Hout; Sponsored by National Science Foundation. –NORC ed.– Chicago: NORC at the University of Chicago [producer]; Storrs, CT: The Roper Center for Public Opinion Research, University of Connecticut [distributor], 2015.)

Obtaining the data

The subset of the GSS used in the examples is available at the book's website, but can also be downloaded directly from the NORC website. Notice that, besides the data itself, it is necessary to obtain files with the metadata, which contains the list of abbreviations for the variables considered in the survey.

To download the data, proceed as indicated in the following steps:

1. Go to <http://www3.norc.org>.
2. In the search field, type GSS 2012 merged with all cases and variables.
3. Click on the link titled **SPSS | NORC**.
4. Scroll down to the **Merged Single-Year Data Sets** section. Click on the link named **GSS 2012 merged with all cases and variables**. If there is more than one release, choose the latest one.
5. Follow the procedure to download the file to the computer. The file will be named `gss2012merged_stata.zip`. Uncompressing the file will create the `GSS2012merged_R5.dta` data file. (The filename may be slightly different for a different release.)
6. If necessary, move the data file to the directory where your notebooks are.

We also need the file that describes the variable abbreviations in the data. This can be done in the following steps:

1. Go to <http://gss.norc.org/Get-Documentation>.
2. Click on the link named **Index to Data Set**. This will download a PDF file with a list of the variable abbreviations and their corresponding meanings. Browsing this file gives you an idea of the scope of questions asked in this survey.

You can feel free to browse the information available on the GSS website. A researcher using the GSS will probably have to familiarize themselves with all the details related to the dataset.

Reading the data

Our next step is to make sure that we can read the data into our notebook. The data is in STATA format. STATA is a well-known package for statistical analysis, and the use of its format for data files is widespread. Fortunately, Pandas allows us to read STATA files in a straightforward way.

If you have not done so yet, start a new notebook and run the default commands to import the libraries that we will need. (Refer to Chapter 1, *Tools of the Trade*.)

Next, execute these commands:

```
gss_data = pd.read_stata('data/GSS2012merged_R5.dta',
                         convert_categoricals=False)
gss_data.head()
```

Reading the data may take a few seconds, so we ask you to be a little patient. The first code line calls Pandas' `read_stata()` function to read the data, and then stores the result, which is an object of the `DataFrame` type in the `gss_data` variable.

The `convert_categoricals=False` option instructs Pandas to not attempt to convert the column data to categorical, sometimes called factor data. As the columns in the dataset are only numbers, where the supporting documents are needed to interpret many of them (for example, gender, 1=male, 2=female), converting to categorical variables does not make sense because numbers are ordered but the translated variable may not be. Categorical data is data that comes in two or more, usually limited, number of possible values. It comes in two types: ordered (for example, size) and unordered (for example, color or gender).



It is important to point out here that categorical data is a Pandas data type, which differs from a statistical categorical variable. A statistical categorical variable is only for unordered variables (as described previously); ordered variables are called statistical ordinal variable. Two examples of this are education and income level. Note that the distance (interval) between the levels need not be fixed. A third related statistical variable is the statistical interval variable, which is the same as an ordinal variable, just with a fixed interval between the levels; an example of this is income levels with a fixed interval.

Before moving on, let's make a little improvement in the way that the data is imported. By default, the `read_stata()` function will index the data records with integers starting at 0. The GSS data contains its own index in the column labelled `id`. To change the index of a DataFrame object, we simply assign a new value to the `index` field, as indicated in the following lines of code (input in a separate Notebook cell):

```
gss_data.set_index('id')
gss_data.drop('id', 1, inplace=True)
gss_data.head()
```

The first line of the preceding code sets the index of `gss_data` field to the column labelled `id`. As we don't need this column in the data any longer, we remove it from the table using the `drop()` method. The `inplace=True` option causes `gss_data` to be modified itself. (The default is to return a new DataFrame object with the changes.)

Let's now save our table to a file in the CSV format. This step is not strictly necessary, but simplifies the process of reloading the data, in case it is necessary. To save the file, run the following code:

```
gss_data.to_csv('GSS2012merged.csv')
```

This code uses the `to_csv()` method to output the table to a file named `GSS2012merged.csv`, using the default options. The CSV format does not actually have an official standard, but because of the simple rules, a file where the entries in each row are separated by some delimiter (for example, a comma), it works rather well. However, as always when reading in data, we need to inspect it to make sure that we have read it correctly. The file containing the data can now be opened with standard spreadsheet software as the dataset is not really large.

Univariate data

We are now ready to start playing with the data. A good way to get an initial feeling of the data is to create graphical representations with the aim of getting an understanding of the shape of its distribution. The word distribution has a technical meaning in data analysis, but we are not concerned with this kind of detail now. We are using this word in the informal sense of *how the set of values in our data is distributed*.

To start with the simplest case, we look at the variables in the data individually without, at first, worrying about relationships between variables. When we look at a single variable, we say that we are dealing with univariate data. So, this is the case that we will consider in this section.

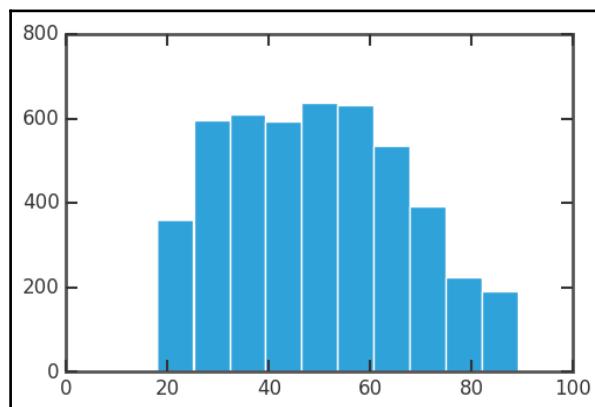
Histograms

A histogram is a standard way of displaying the distribution of quantitative data, that is, data that can be represented in terms of real numbers or integers. (Notice that integers can also be used to indicate some types of categorical data.) A histogram separates the data in a number of bins, which are simply intervals of values, and counts how many of the data points lie in each of the bins.

Let's concentrate on the column labelled `age`, which records the respondent's age. To display a histogram of the data, run the following line of code:

```
gss_data['age'].hist()  
plt.grid()  
plt.locator_params(nbins=5);
```

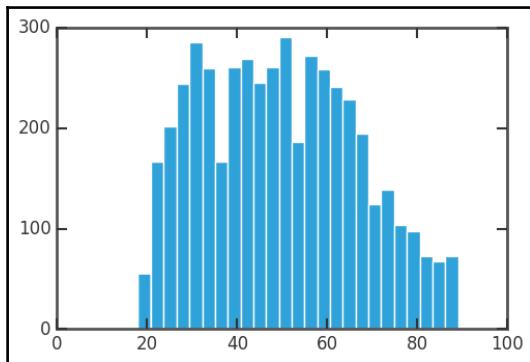
In this code, we use `gss_data['age']` to refer to the column named age, and then call the `hist()` method to draw the histogram. Unfortunately, the plot contains some superfluous elements, such as a grid. Therefore, we remove it by calling the `plt.grid()` trigger function, and after this, we redefine how many tick locators to place with the `plt.locator_params(nbins=5)` call. Running the code will produce the following figure, where the *y* axis is the number of elements in the bin and the *x* axis is the age:



The key feature of a histogram is the number of bins in which the data is placed. If there are too few bins, important features of the distribution may be hidden. On the other hand, too many bins cause the histogram to visually emphasize the random discrepancies in the sample, making it hard to identify general patterns. The histogram in the preceding figure seems to be too *smooth*, and we suspect that it may hide details of the distribution. We can increase the number of bins to 25 by adding the option `bins` to the call of `hist()`, as shown in the code that follows:

```
gss_data['age'].hist(bins=25)  
plt.grid()  
plt.locator_params(nbins=5);
```

The plot in the following screenshot is displayed:



Notice that now the histogram looks different; we see more structure in the data. However, the resolution is still good enough to show the main features of the plot:

- The distribution is approximately *unimodal*, that is, there is only one significant *hump*. Notice that in making this assessment, we do not take into account small gaps and peaks that are most likely due to sample randomness.
- The distribution is *asymmetrical*, being somewhat skewed to the right, that is, it has a longer tail extending toward the high values.
- The distribution ranges, approximately, from 20 to 90 years, and is centered somewhere near 50 years. It is not clear what is the mode, or highest point, of the distribution.
- There are no unusual features, such as outliers, gaps, or clusters.

Notice that, from these observations, we can already say something about the data collection: it is likely that there is a minimum age requirement for respondents. This may be the cause of the asymmetry in the distribution. Requiring a lower bound in the sampling usually makes the upper tail of the distribution stand out.

It is useful to compare this distribution with a distribution of the income of the respondent, contained in the `realrinc` column. There is a little trap lying there, however. Let's start by creating a DataFrame that contains only the two columns of interest and display the first few lines of the result, by running the following commands:

```
inc_age = gss_data[['realrinc', 'age']]
inc_age.head(10)
```

Notice the *double brackets* in the first line of the preceding code. We are using one of the many sophisticated indexing capabilities provided by Pandas and passing a Python list, `['realinc', 'age']`, as the index to the `gss_data` DataFrame. This has the expected effect of selecting the two columns specified in the Python list.

Looking at the output of the previous command, we can see that the `realinc` column has many missing values, indicated by the `NaN` value, which is the default that Pandas uses for missing data. This may happen due to several reasons, but some respondents simply opt not to reveal their incomes. Thus, to compare the distribution of the two columns, we can omit these rows, as shown in the following code:

```
inc_age = gss_data[['realrinc', 'age']].dropna()  
inc_age.head(10)
```

We use the same indexing to select the two columns, but now we call the `dropna()` method to exclude the rows with missing data. Examining the output, notice that Pandas smartly keeps the row indexing, ID, from the original DataFrame from where the values were extracted. This way, if needed, we can cross-reference the data with the original table.

It is now pretty straightforward to produce side-by-side histograms of the two variables, using the following lines of code:

```
ax_list = inc_age.hist(bins=40, figsize=(8,3), xrot=45)  
for ax in ax_list[0]:  
    ax.locator_params(axis='x', nbins=6)  
    ax.locator_params(axis='y', nbins=3)
```

Notice the options that we used in the `hist()` method. Besides setting the number of bins, we use the `figsize=(8,3)` option, setting the figure size to 8 inches by 3 inches and the `xrot=45` option, causing the `x` axis labels to be rotated by 45 degrees, improving readability. The command returns a list of the axes' objects of the figure. We save this to the `ax_list` variable. Next, we iterate over this list to make modifications of the axes' objects (that is, the plots we are drawing). As before, we change the number of tick marks, this time with a different function, using the object-oriented interface of matplotlib. Play around with the `nbins` setting and see what happens.

Examining the resulting histograms, we can see that they are significantly different: the distribution of incomes is heavily skewed and, more importantly, there is a large gap with an isolated bar in the region above 300,000 dollars. Let's count how many values are in this region, as in the following code:

```
inc_age[inc_age['realrinc'] > 3.0E5].count()
```

In this code, we are using a Boolean index, `inc_age['realrinc'] > 3.0E5`, in the DataFrame. This selects all the rows for which the value in the `realinc` column is larger than 300,000 dollars (`3.0E5` is equivalent to 3.0×10^5). The `count()` method simply counts how many values are there that satisfy the condition.

Looking at the output, we can see that there are 80 rows with an income above 300,000 dollars. As always, when there is something that looks unusual, we should look at the data more carefully. Let's display the data for the corresponding values by running the following code:

```
inc_age[inc_age['realrinc'] > 3.0E5].head(10)
```

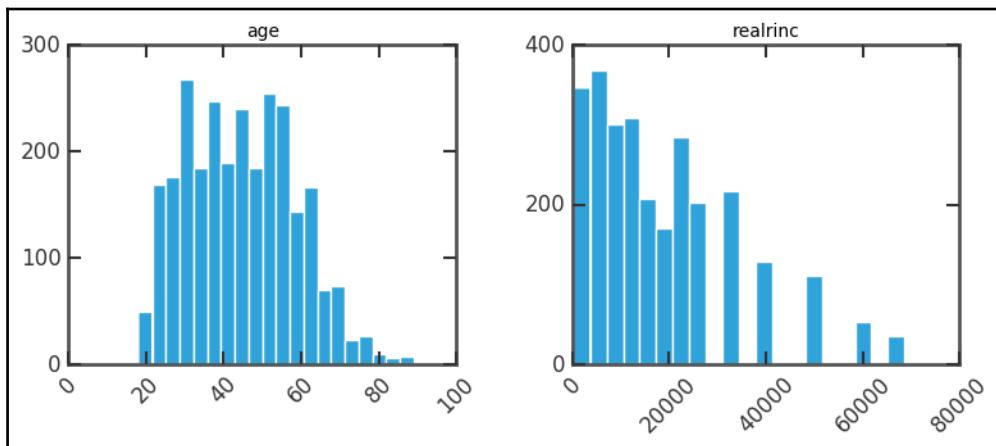
This is very similar to the command line in the previous example, but now we take a slice of the data corresponding to the first ten rows. The output contains a surprise: all data values are equal!

To understand what is happening, we have to dig deeper into the assumptions made in the GSS survey. The survey does not ask respondents for a number value for their incomes. Instead, the respondent is presented with income categories, which imposes an upper bound on the incomes in the questionnaire. That is, all respondents with an income above a certain value are lumped together. By the way, the problem of estimating the real incomes from the categories is nontrivial, and there is quite a body of research on the subject.

Anyway, for our purposes, it is legitimate to simply exclude the values above 300,000 dollars. To produce histograms with this assumption, run the code indicated in the following lines:

```
inc_age = gss_data[['realrinc', 'age']].dropna()
lowinc_age = inc_age[inc_age['realrinc'] < 3.0E5]
ax_list = lowinc_age.hist(bins=20, figsize=(8, 3), xrot=45)
for ax in ax_list[0]:
    ax.grid()
    ax.locator_params(axis='x', nbins=6)
    ax.locator_params(axis='y', nbins=3)
```

Notice the second line in the preceding code, which selects the rows in the `inc_age` DataFrame corresponding to entries where the value in the `realrinc` column is less than 300,000 and stores this in a new `lowinc_age` object. This produces the histograms displayed in the following picture:



By looking at the output, we can see that the distributions are quite distinct. The income distribution is markedly skewed toward large values and appears to have several gaps. Note that the gaps may be due to the survey construction, especially with what concerns the way the real income is computed from the income ranges.

Making things pretty

The histograms shown in the examples so far are adequate for data exploration but are not quite visually appealing. Pandas uses matplotlib for graphs. Matplotlib is an extensive library for technical plotting, capable of producing high-quality, presentation-ready graphs (<http://matplotlib.org>). To illustrate the possibilities, run the following code (`lowinc_age` stored in the previous code):

```
ax_list = lowinc_age.hist(bins=20, figsize=(8,3),
                           xrot=45, color='SteelBlue')
ax1, ax2 = ax_list[0]
ax1.set_title('Age (years)')
ax2.set_title('Real Income ($)')
for ax in ax_list[0]:
    ax.grid()
    ax.locator_params(axis='x' ,nbins=6)
    ax.locator_params(axis='y' ,nbins=4)
```

Let's start by analyzing the call to the `hist()` method in the third line of the preceding code. When using matplotlib for plotting, there are several approaches to each command. Here, we show the object-oriented way, where `ax1, ax2 = ax_list[0]` fetches and stores the two axes. Then, we set the title and turn off the background grid of each axis using these objects.

Characterization

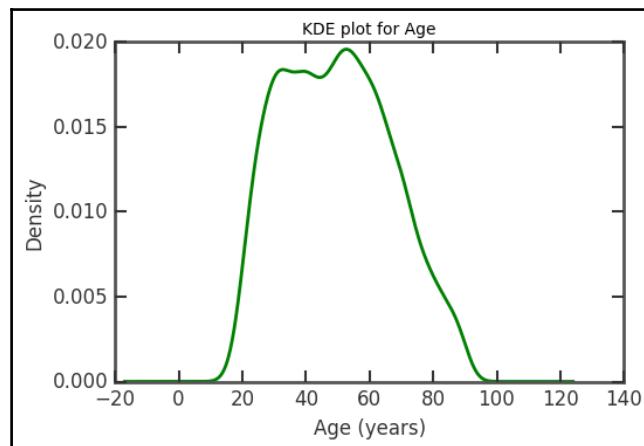
We now consider the question of trying to fit the data into one of the standard models in classical statistics. This may be a complicated problem as real data may not conform to any predefined model. The first step may be to try to approximate the density of the distribution, that is, a fraction of the people at a certain age in this case (assuming that it is continuous). A frequently used method is **Kernel Density Estimation (KDE)**, which can be thought of as a *smoothed histogram*. Pandas can easily produce KDE plots, as shown in the following code:

```
age = gss_data['age'].dropna()
age.plot(kind='kde', lw=2, color='green')
plt.title('KDE plot for Age')
plt.xlabel('Age (years)')
```

In this code, we first select the age column from the data, dropping the missing values. We then call the `plot()` method with the `kind='kde'` option (with `kde` as an option, the SciPy package is a dependency). Notice that this is a slightly different interface from what we used for histograms. Also notice that we use options to set the line width and color of the plot. The last two lines set the title and label of the *x* axis, although with the direct function approach this time as opposed to the object-oriented approach as shown previously. How would you plot this with an object-oriented approach?

An important factor to notice is that Pandas does not support setting parameters for the KDE plot, in particular, bandwidth used in the smoothing procedure. The bandwidth is somewhat analogous to the bin width in a histogram, and different bandwidth can produce significantly different approximations. Pandas uses a heuristic approximation that produces a nearly optimal fit in most cases, but this may not produce the best results in some cases. We expect that future versions of Pandas will be more flexible here.

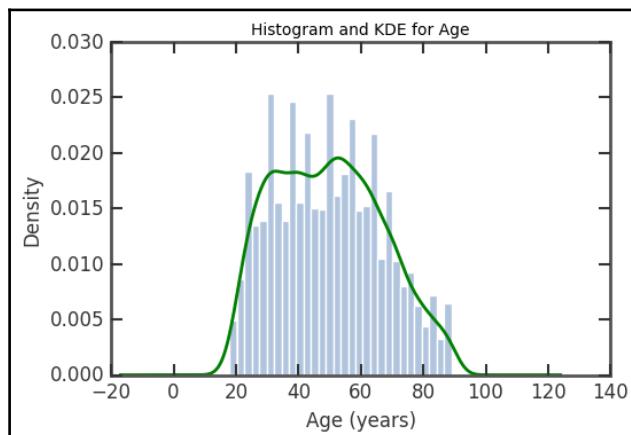
Running the preceding code, we get the following display:



While this is all good, we might want to plot the KDE over the histogram. This can be done with the following code:

```
ax = age.hist(bins=30, color='LightSteelBlue', normed=True)
age.plot(kind='kde', lw=2, color='Green', ax=ax)
plt.title('Histogram and KDE for Age')
plt.xlabel('Age (years)');
```

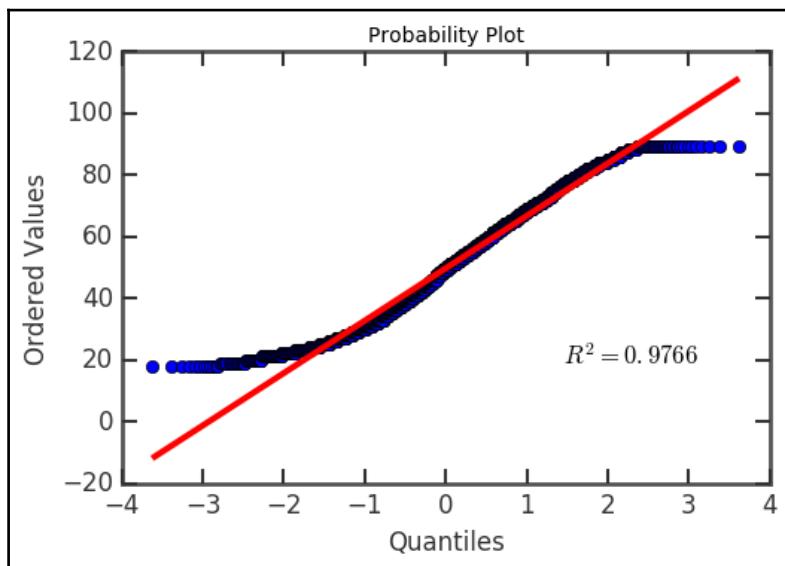
This will produce a figure with the histogram, normalized just like the KDE plot, and the KDE curve on top of it. The figure will look as follows:



The KDE curve is somewhat bell-shaped, but if you are familiar with the normal distribution, you will notice that the plot seems to drop too quickly to zero at the tails. To visually assess how much the data deviates from a normal distribution, we can use a **Normal Distribution Plot**. In a normal distribution plot, the data is plotted alongside values of a normal distribution with similar characteristics as the data would have. This kind of plot is not supported in the current version of Pandas, but we can use SciPy to create the plot, as indicated in the following lines of code:

```
import scipy.stats as stats  
stats.probplot(age, dist='norm', plot=plt)
```

The first parameter, `age`, is the data to be plotted. We then use the `dist='norm'` option to compare the data with a normal distribution. The final option, `plot=plt`, specifies that the `plt` module should be used for the plotting. This can be an existing axis instance (object) or a plotting module; in this case, we just send it the `plt`, which is the `matplotlib.pyplot` module. The resulting plot is shown in the following figure:

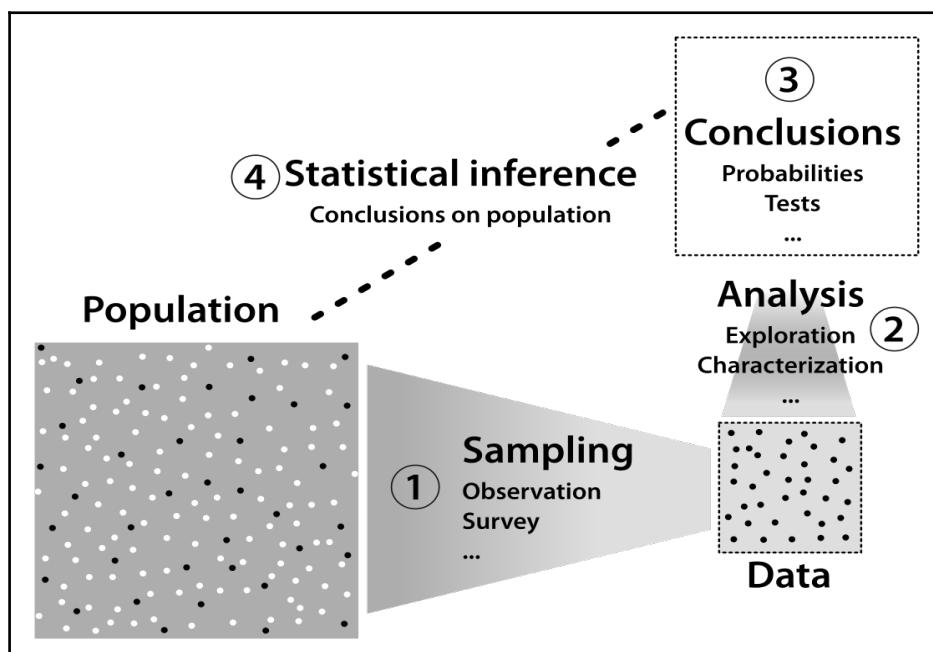


Notice that the data deviates significantly from the straight line at the ends of the plot: it is above the line on the left and below the line on the right. The more similarities with a normal distribution, the more the data values will follow the straight line. The age data presented here is consistent with data that has tails that are shorter than those of a normal distribution, which is consistent with our earlier observations. We would thus conclude that a normal distribution would not be adequate for this data.

Concept of statistical inference

At this point of the analysis, we have figured out something about the sample. It is not normally distributed. Further on in the exploration, we will find out even more about the sample. With adequately sampled data from the full population, we can draw conclusions about the population by conducting analysis on the data sample. This is the basis of statistical inference, and the following image illustrates this concept. The concept is mentioned in the following steps:

1. We draw a hopefully unbiased sample from the population.
2. Through data analysis, we characterize the sample data.
3. With statistical tests, parameter estimation, and similar tools, we draw conclusions about the sample.
4. Through inference, we can now draw conclusions on the whole population.



Numeric summaries and boxplots

We now move in the direction of describing the data numerically. Before we start, we need a word of caution. It is not good practice to simply use a set of numbers from the data to draw conclusions from it. It is very tempting to rely on the feeling that definite numbers provide a level of *certainty*. However, numerical values without context and no further analysis are not very useful and may be misleading. In this chapter, we are only considering methods for initial study, trying to familiarize ourselves with the data in order to get a feeling for how it behaves.

When considering numerical data, we may ask the following questions:

- What is the range of the data? That is, what are the smallest and largest values?
- Where is the center of the data values located? We will consider two measures of centrality, the *mean* and *median*.
- How much does the data spread from its center? We will consider the *standard deviation* as well as the notions of *quartiles* and *percentiles* as measures of spread.

All these quantities are easily computed in Pandas with the `describe()` method, available for objects of the Series and DataFrame types. Going back to the distribution of income, we can plot a data summary as follows:

```
inc = gss_data['realrinc'].dropna()
lowinc = inc[inc < 3.0E5]
lowinc.describe()
```

In the first line, we select the `realrinc` column from the table using the `dropna()` method to discard missing data. We then select only incomes less than 300,000 dollars as the income reported in that column may not reliably reflect the distribution of high incomes, which we have discussed before. Finally, we use the `describe()` method to generate a summary of the data. Running this code, we obtain the output shown in the following lines:

```
count      2751.000000
mean      18582.194656
std       14841.581333
min       245.000000
25%      6737.500000
50%     15925.000000
75%     26950.000000
max      68600.000000
Name: realrinc, dtype: float64
```

The information provided in this output is described as follows:

- The count is the number of data points. So, there are 2,751 people that reported incomes less than 300,000 dollars in the survey.
- The mean is the average of the data. So, the average income of respondents is approximately \$18,582.
- The standard deviation (std) is a measure of how the data spreads around the mean. The formula for the standard deviation is somewhat technical and will be introduced in a later chapter.
- The minimum (min) and maximum (max) are the smallest and largest values in the data. Together, they specify the range of the data. In our example, income goes from the lowest value of \$245 to the highest value of \$68,600.
- The median (50%) is the value that corresponds to the middle point of the dataset, in the sense that half of the values are below the median and half of the values are above the median. For example, for our data, half of the reported incomes are below \$15,925 and half are above this value.



The single values derived here represent certain characteristics of our data referred to as **point estimates**. The most common and widely used point estimate is the sample mean, which gives a point estimate of the population mean through statistical inference. Point estimates are compliments to *interval estimates*. These are defined by two or more numbers, for example, if the sample mean is indicated to lie in a certain interval, this indicates that the population mean lies in this certain interval as well.

- The quartiles (25% and 75%) together with the median give a more specific view of how the data is distributed. In our data, they can be interpreted as follows:
 - 25% of the incomes are below \$6,737
 - 25% of the incomes are between \$6,737 and \$15,925
 - 25% of the incomes are between \$15,925 and \$26,950
 - 25% of the incomes are between \$26,950 and \$15,925

If a more detailed view of the distribution is desired, we can request the output of a larger number of percentiles, as shown in the following code:

```
lowinc.describe(percentiles=np.arange(0, 1.0, 0.1))
```

In this code, we use the percentiles option of the `describe()` method. The `np.arange(0, 1.0, 0.1)` expression represents a NumPy array with the numbers `0.0, 0.1, 0.2, ..., 0.9`. The output of the previous example will provide values that split the data into 10 intervals, each containing 10% of the reported incomes.

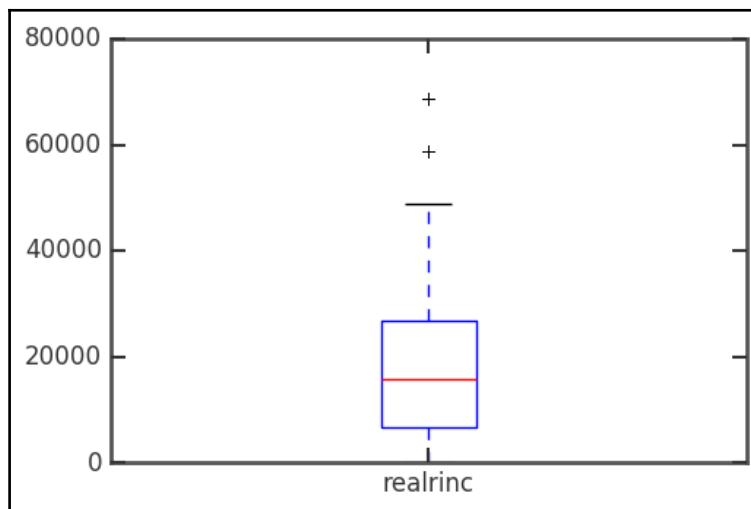
Numerical summaries can be, admittedly, hard to visualize. A very old, but still useful, graphical tool is the **boxplot**. Although somewhat out of fashion, boxplots can still be used to display the center and spread of the data. We can display a boxplot of incomes with the following code:

```
lowinc.plot(kind='box');
```

Alternatively, it is possible to run the following:

```
lowinc.plot.box();
```

Running either of these commands produces the following plot:



This boxplot can be interpreted as follows:

- The points marked with a cross symbol are outliers. Outliers are values that are far removed from the center of the distribution. There is no clear universally accepted definition of what an outlier is, so Pandas determines them heuristically (http://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.boxplot).

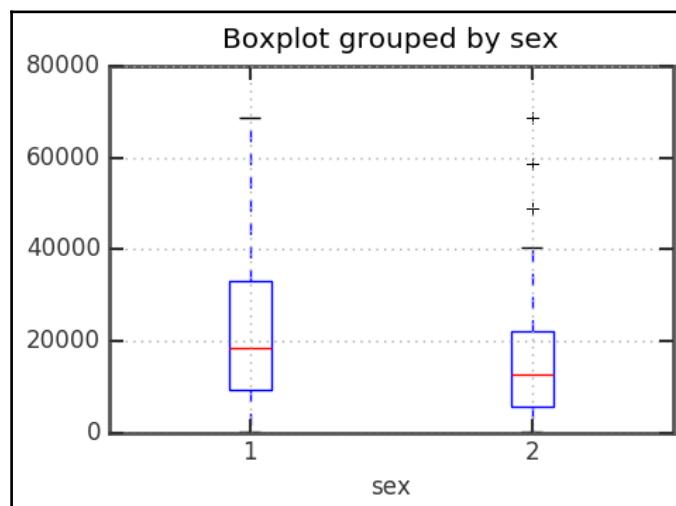
- The horizontal bars at the bottom and top (the *whiskers* of the boxplot) represent the minimum and maximum, respectively.
- The bottom and top of the box represent the 25% and 75% quartiles, respectively.
- The line inside the box represents the median or 50% quartile.

A quick way to interpret a boxplot is to remember that, excluding outliers, 50% of the data values are inside the box, 25% are below the box, and 25% are above the box. Notice that, in our example, the asymmetry of the data is evident with the boxplot being skewed toward high values.

Boxplots are also useful to compare data for different subpopulations. Let's say, for example, that we want to compare the incomes of males and females. This can be accomplished with the following code:

```
inc_gen = gss_data[['realrinc', 'sex']]
inc_gen = inc_gen[inc_gen['realrinc'] < 3.0E5]
inc_gen.boxplot(column='realrinc', by='sex');
```

The first two lines select the data to be plotted, excluding incomes above 300,000. Then, the `boxplot()` method of the `DataFrame` object is called. We specify the data to be plotted with the `column='realrinc'` option and the groups with the `by='sex'` option. When executed, the following plot will be displayed:



These boxplots tell an interesting story about how incomes are distributed. The incomes for males (1) are more spread than the values for females (2), with the incomes for males being markedly above the female incomes. For example, the top quartile of the female's distribution is just a little above the median male income. That is, almost 75% of the females have an income corresponding to the bottom half of male incomes. The upper level of the distribution tells a similar story. The portion of the boxplot above the upper quartile is much longer for the male distribution than that of the female distribution. Notice, in particular, the outliers in the female distribution. There definitely seems to be a *glass ceiling* effect, with very few women being able to command the top salaries that men achieve.

Relationships between variables – scatterplots

The real power in data analysis is realized when we study how different variables relate. At the end of the previous section, we related income and gender, that is, a quantitative variable with a categorical one. In this section, we will investigate scatterplots, which are a graphical representation of the relationship between two quantitative variables.

To illustrate how Pandas can be used to explore the relationship between two variables, we will use an important example from the history of astronomy. Astronomer Edwin Hubble, in 1929, published a very important paper where he discovered that there is an approximately linear relationship between the distance and velocity of extragalactic nebulae. This was the foundation that would come to be the big bang theory.

A reprint of the article is available at http://apod.nasa.gov/diamond_jubilee/d_1996/hub_1929.html, where the data was obtained from. Notice that the dataset is very small and simply printed in the article itself! Some minor formatting and cleanup was done to the data to make it easier to use. In particular, the velocity values were manually changed to be all positive as only the magnitude of the velocity matters. To plot a scatterplot of the data, enter and run the following code:

```
hubble_data = pd.read_csv('data/hubble-data.csv')
hubble_data.plot(kind='scatter', x='r', y='v');
```

In this code, after reading the data using the `read_csv()` function, we plot the data using the `plot()` method with the `kind='scatter'` option. The `x='r'` and `y='v'` option tell Pandas which columns to plot in the *x* and *y* axes, respectively.

Observing the plot, it is clearly seen that there is a relationship between distance and velocity, contradicting the view, prevalent at the time, that the Universe is stationary. To make the relationship clearer, we can add a trend line to the plot.



A central part of statistical inference is hypothesis testing, where one dataset/sample is tested against another or tested against a model-generated dataset. Statistical hypothesis testing is then used to investigate a proposed relationship between the two datasets. The proposed relationship is compared to an idealized null hypothesis, that is, that no relationship exists between the two datasets. The null hypothesis is rejected only if the probability of it being true is below a certain significant level. That is, hypothesis testing can only give the significance of the null hypothesis, not the proposed model. This is a very odd and usually hard concept to grasp. We will touch on hypothesis testing in this chapter, and take a deeper dive in Chapter 4, *Regression*.

We first need to compute the linear regression line for the relationship. We do this using SciPy with the following code:

```
from scipy.stats import linregress
rv = hubble_data.as_matrix(columns=['r', 'v'])
a, b, r, p, stderr = linregress(rv)
print(a, b, r, p, stderr)
```

We start by importing the `linregress` function from the `scipy.stats` module. This module is not Pandas-aware, so we first convert the data to a NumPy array using the `as_matrix()` method. Next, we call the `linregress` function, which returns the following:

- `a`: This is the slope of the regression line
- `b`: This is the intercept of the regression line
- `r`: This is the correlation coefficient
- `p`: This is the two-sided p-value for the hypotheses test-for the null hypothesis that assumes the slope is zero
- `stderr`: This is the standard error of the estimate

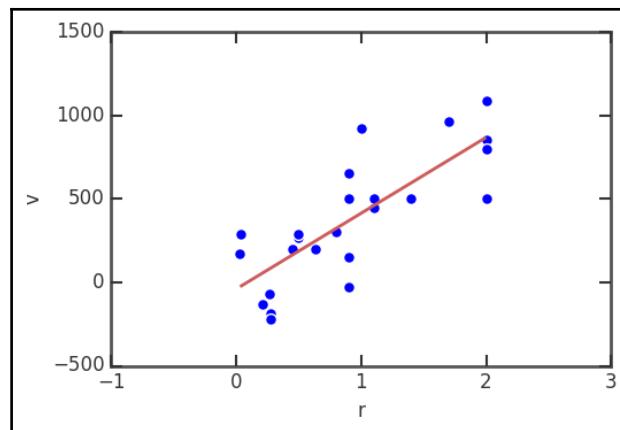
For our example, we get, rounding to two decimals, `a=454.16`, `b=-40.78`, `r=0.79`, `p=4.48E-6`, and `stderr=75.24`.

Linear regression is covered in Chapter 4, *Regression*, but we will interpret the results here. The correlation coefficient of 0.79 indicates a strong relationship, and the very small p-value indicates that the null hypothesis should be rejected, giving support to the existence of a relationship between the variables. The square of r is 0.62, so 62% of the variability in the data is explained by the linear model as opposed to random variation.

All this indicates that a linear model can describe the increase in velocity as a function of distance for galaxies in the Universe. To display this visually, we can plot the regression line together with the data using the following code:

```
hubble_data.plot(kind='scatter', x='r', y='v')
rdata = hubble_data['r']
rmin, rmax = min(rdata), max(rdata)
rvalues = np.linspace(rmin, rmax, 200)
yvalues = a * rvalues + b
plt.plot(rvalues, yvalues, color='red', lw=1.5)
plt.locator_params(nbins=5);
```

As Pandas does not currently support the option of drawing regression lines on scatterplots, we take advantage of the fact that matplotlib is used by Pandas in the background to construct plots. After graphing the scatterplot, we compute the maximum and minimum values of the data and generate a NumPy array with equally spaced values of the distance with a call to the `linspace()` function. Then, the `yvalues = a* rvalues + b` statement computes the points in the line. Finally, we call matplotlib's `plot()` function to plot the line. The resulting graph is displayed in the following image:



From this model, Hubble went on to hypothesize that the Universe is expanding, an idea that eventually yielded the model for the Universe currently accepted in cosmology.

Summary

In this chapter, you learned how to use Pandas to perform an initial exploration of the data. You learned about displays of data, including histograms, KDE plots and boxplots for univariate distributions, and scatterplots for bivariate relationships. We also discussed summaries of data, including mean, standard deviation, range, median, quartiles, and percentiles.

In the next chapter, you will learn about statistical models for data.

3

Learning About Models

In the most generic sense, a **model** is an approximate description of a portion of reality. Models are essential to science and, in fact, any area of knowledge: it is only possible to comprehend the world by concentrating on a small part of it at a time and making suitable simplifications.

In this chapter, we will discuss the following topics:

- Using basic models in data analysis
- Using the cumulative distribution function and probability density function to characterize a variable
- Using the preceding functions and various tools to make point estimates and generating random numbers with a certain distribution
- Discussing examples of discrete and continuous random variables and an overview of multivariate distributions

Models and experiments

Models can take many forms: a verbal description, set of mathematical equations, or segment of computer code. In this book, we are interested in a specific kind of model, *probabilistic* or *statistical* model, which represents the variability that occurs in a nondeterministic experiment.



We use the term **experiment** in this book in a somewhat non-technical sense. For us, an experiment is any observation of an event of interest. Examples of experiments are observing the number of visitors to a website or conducting an opinion poll or clinical trial. The main characteristic of experiments, for us, is that they can be repeated and that there is randomness, that is, each repetition of the same experiment may result in different outcomes.

The models that we will consider take the form of random variables. A random variable is an idealized representation of a probabilistic outcome that has numerical results. It is important to realize that a random variable is an abstraction: it does not represent the outcome of a particular experiment, it just models what results we expect to get once the experiment is actually performed.

In the remainder of this chapter, we will discuss how statistical models are formulated and describe the most important models used in data analysis.

Before running the examples in this chapter, start the Jupyter Notebook. After the default imports, run the following commands in a cell:

```
from pandas import Series, DataFrame
import numpy.random as rnd
import scipy.stats as st
```

You are now ready to start running the code for this chapter.

The cumulative distribution function

In the previous chapter, when discussing visual representations of numerical data, we introduced histograms, which represent the way the data is distributed across a number of intervals. One of the drawbacks of histograms is that the number of bins is always chosen somewhat arbitrarily, and incorrect choices may give useless or misleading information about the distribution of the data.

We say that histograms abstract some of the characteristics of the data. That is, a histogram allows us to ignore some of the fine-grained variability in the data so that general patterns are more apparent.

Abstraction is, in general, a good thing when analyzing a dataset but we would like to have an accurate representation of *all* data points that is visually compelling and computationally useful. This is provided by the *cumulative distribution function*. This function has always been important for statistical computations, and cumulative distribution tables were in fact an essential tool before the advent of the computer. However, as a graphical tool, the cumulative distribution function is not usually emphasized in introductory statistics texts. In my opinion, this is partly due to a historical bias as it is unwieldy to draw a cumulative distribution function without the aid of a computer.

To give a concrete example, let's first generate a set of random values following a Normal distribution with the following code segment:

```
mean = 0
sdev = 1
nvalues = 10
norm_variate = mean + sdev * rnd.randn(nvalues)
print(norm_variate)
```

In this code, we use the NumPy `numpy.random` module (abbreviated as `rnd`) to generate the randomized values. The documentation for the random module can be found at <http://docs.scipy.org>. Specifically, we use the `randn()` function, which generates normally distributed pseudorandom numbers with mean 0 and variance 1, then we move the distribution to the mean through addition and widen the distribution by `sdev` through multiplication. This function takes the number of values that we want as an argument, which we stored in the `nvalues` variable.



Notice that we use the term **pseudorandom**. When generating the random numbers, the computer actually uses a formula that produces values that are distributed, approximately, according to a given distribution. These values cannot be truly random as they are generated by a deterministic formula. It is possible to use sources of *true randomness*, as provided, for instance, by the site <https://www.random.org/>. However, pseudorandom numbers are usually sufficient for computer simulations and most data analysis problems.

The result is stored in the `norm_variates` variable, which is a NumPy array. We can pretend that the numbers represent the offset in grams from the target weight for a 10 gram package of saffron, perhaps to make more sense of the numbers. This would mean that `-0.1` means that the package contains `0.1` gram less saffron than it should and `+0.2` means that it contains more saffron than it should.

Running this cell will produce an array containing 10 numbers. Running the code with more values, say 100 (that is, `nvalues=100`), would produce a distribution that follows a normal distribution more closely. This array should follow, approximately, a Normal distribution with a mean 0 and standard deviation of 1. Values that result from the sampling from a random variable, for example, the values in the `norm_variate` array, are called **random variates**. The cumulative distribution function of a dataset is, by definition, a function that, given a value of `x`, returns the number of data points that do not exceed `x`, normalized to the range from 0 to 1. To give a concrete example, let's sort the values we generated previously and print them using the following code:

```
for i, v in enumerate(sorted(norm_variates), start=1):
    print('{0:2d} {1:+.4f}'.format(i, v))
```

This code uses a `for` loop structure to iterate over the sorted list of random variates. We use `enumerate()`, which provides a Python iterator that returns both the index `i` and corresponding value `v` from the list in each iteration. The `start=1` parameter causes the iteration number to start at 1. Then, for each pair, `i` and `v` are printed. In the `print` statement, we use format specifiers, which are the expressions enclosed by curly brackets, `{...}`. In this example, `{0:2d}` specifies that `i` is printed as a *2-digit decimal value* and `{1:+.4f}` specifies that `v` is printed as *assigned float value with four digits of precision*. As a result, we obtain a sorted list of the data, numbered from 1.

The values obtained by me are as follows:

1 -0.1412
2 +0.6152
3 +0.6852
4 +2.2946
5 +3.2791
6 +3.4699
7 +3.6961
8 +4.2375
9 +4.4977
10 +5.3756



As we are sampling from a (pseudo) random variable, you will obtain a different set of values.

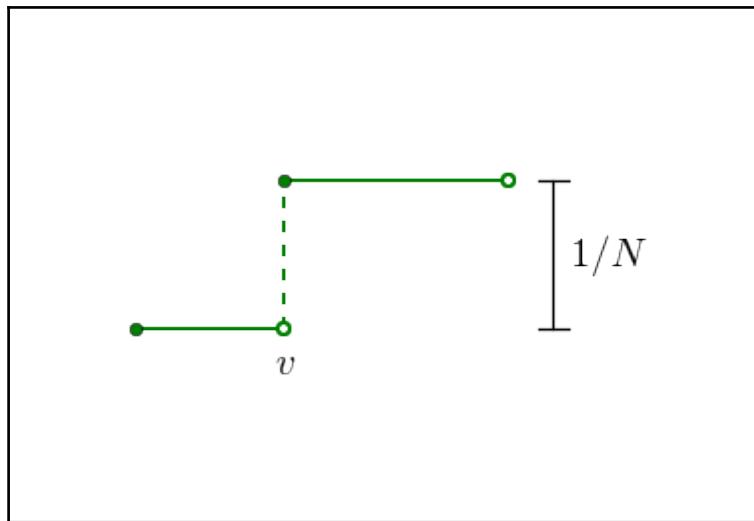
From this list, it is easy to compute values of the cumulative distribution function. Consider the value 2.2946, for example. There are four data points that are less than or equal to the given value: -0.1412, 0.6152, 0.6852, and 2.2946 itself. We now divide the number of values, 4, by the number of points so that the result is between 0 and 1. So, the value of the cumulative distribution function is $4/10=0.4$ for these 10 values. In terms of a mathematical expression, we write the following:

$$F(2.2946) = 0.4$$

It is common practice to abbreviate *cumulative distribution function* by *cdf*, and we will do so from now on.

An important point to notice is that the cdf is not defined only at values in the dataset. In fact, it is defined for any numeric value whatsoever! Let's suppose, for example, that we want to find the value of the cdf at $x=2.5$. Notice that the number 2.5 is between the fourth and fifth values in the dataset, so there are still four data values less than or equal to 2.5. As a consequence, the value of the function at 2.5 is also $4/10=0.4$. In fact, it can be seen that, for all numbers between 2.2946 and 3.2791, the cdf will have the value 0.4.

Thinking a little bit about this process, we can infer the following behavior for a cdf: it remains constant (that is, *flat*) between values in the dataset. At each data value, the function will *jump*, the size of the jump being the reciprocal of the number of data points. This is illustrated in the following figure:



In the preceding figure, v is a point in the dataset and the graph displays the cdf in the neighborhood of v . Notice the flat intervals between the data points and the jump exactly at the data point v . The filled circle at the discontinuity indicates the value of the function at that point. These are characteristics of any cumulative distribution function for a discrete dataset.

Let's now define a Python function that can be used to plot the graph of a cdf. This is done with the following code:

```
def plot_cdf(data, plot_range=None, scale_to=None, **kwargs):
    num_bins = len(data)
    sorted_data = np.array(sorted(data), dtype=np.float64)
    data_range = sorted_data[-1] - sorted_data[0]
    counts, bin_edges = np.histogram(sorted_data, bins=num_bins)
    xvalues = bin_edges[:-1]
    yvalues = np.cumsum(counts)
    if plot_range is None:
        xmin = sorted_data[0]
        xmax = sorted_data[-1]
    else:
        xmin, xmax = plot_range
```

```
#pad the arrays
xvalues = np.concatenate([[xmin, xvalues[0]], xvalues, [xmax]])
yvalues = np.concatenate([[0.0, 0.0], yvalues, [yvalues.max()]])
if scale_to is not None:
    yvalues = yvalues / len(data) * scale_to
plt.axis([xmin, xmax, 0, yvalues.max()])
return plt.plot(xvalues, yvalues, **kwargs)
```

Notice that running this code will not produce any output as we are simply defining the `plot_cdf()` function. The code is somewhat complex, but all we are doing is defining lists of points stored in the `xvalues` and `yvalues` arrays. These values are the leading edge of the staircase and the height of the specific step. The `plt.step()` function plots these in a step plot. We use NumPy's `concatenate()` function to pad the array, in the start with zeroes, and in the end with the maximum (or last) value of the array. To plot the cdf for a dataset, we can run the following code:

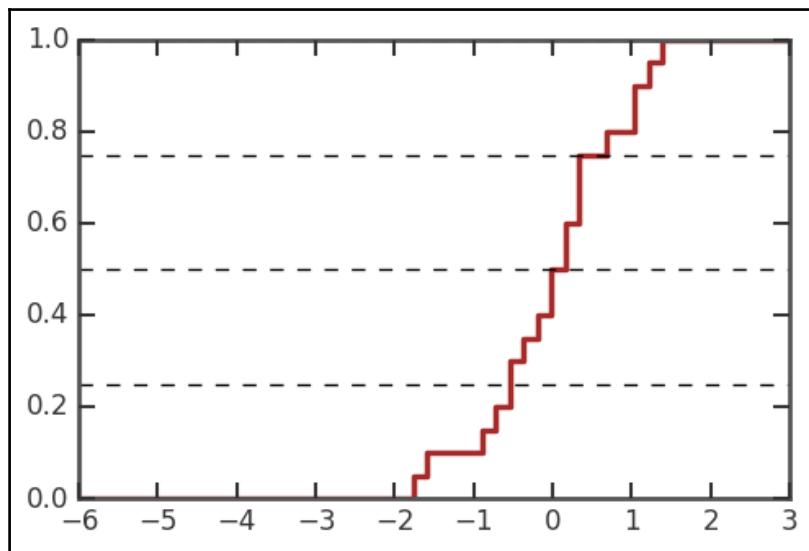
```
nvalues = 20
norm_variates = rnd.randn(nvalues)
plot_cdf(norm_variates, plot_range=[-3,3], scale_to=1.0,
          lw=2.5, color='Brown')
for v in [0.25, 0.5, 0.75]:
    plt.axhline(v, lw=1, ls='--', color='black')
```

In this code, we first generate a new set of data values. Then, we call the `plot_cdf()` function to generate the graph. The arguments of the function call are `plot_range`, specifying the range in the `x` axis, and `scale_to`, which specifies that we want the values (`y` axis) to be normalized from 0 to 1. The remaining arguments to the `plot_cdf()` function are passed to the Pyplot function, `plot()`. In this example, we set the line width with the `lw=2.5` option and line color with the `color="Brown"` option.



The purpose of the `scale_to` option is to allow setting a different range for `yvalues` in the plot. With `scale_to=100.0`, the `y` axis is scaled in percentages and `scale_to=None` represents the count of data points without any scaling.

Running this code will produce an image similar to the following one:



The graph obtained by you will be somewhat different as the data is randomly generated. In the preceding figure, we also draw horizontal lines at y values 0.25, 0.5, and 0.75. These correspond, respectively, to the first quartile, median, and third quartile in the data. Looking at the corresponding values in the x axis, we see that these values correspond approximately to -0.5, 0, and 0.5. This makes sense as the actual values for a theoretical Normal distribution are -0.68, 0.0, and 0.68. Notice that we do not expect to recover these values exactly due to randomness and the small size of the sample.

Now, I invite you to perform the following experiment. Increase the number of values in the dataset by changing the value of the `nvalues` variable in the preceding code, and run the cell again. It will be noticed that, as the number of data values increases, the curve becomes smoother and converges toward an S-shaped curve, symmetric about its center. Also notice that the quartiles and median will tend to approach the theoretical values for the Normal distribution, -0.68, 0.0, and 0.68. As we will see in the next section, these are some of the characteristics of the standard Normal distribution.

Let's investigate the cdf for a realistic dataset. The `housefly-wing-lengths.txt` file contains wing lengths in tenths of a millimeter for a small sample of houseflies. The data is from a 1968 paper by *Sokal, R. R.*, and *Rohlf, J. R.* in the journal *Biometry* (page 109) and a 1955 paper by *Sokal, R. R.* and *P.E. Hunter* in the journal *Annual Entomol. Soc. America* (Volume 48, page 499). The data is also available online at <http://www.seattlecentral.edu/qelp/sets/057/057.html>.

To read the data, make sure that the `housefly-wing-lengths.txt` file is in the same directory as the Jupyter Notebook, and then run the following code segment:

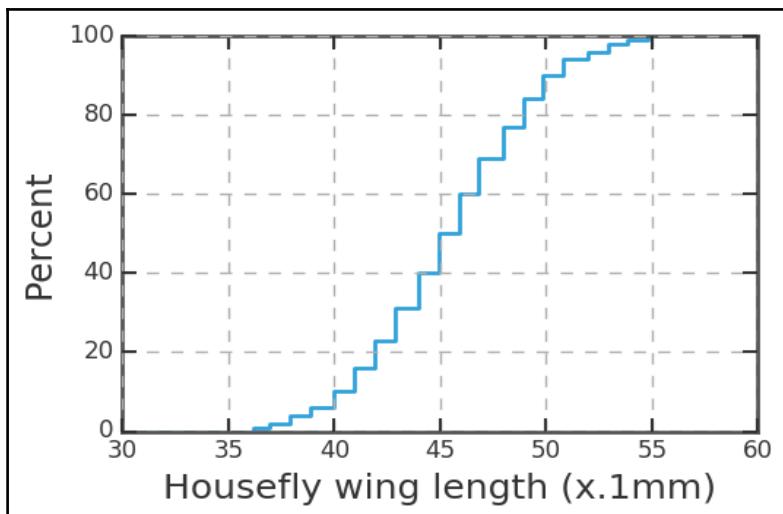
```
wing_lengths = np.fromfile('data/housefly-wing-lengths.txt',
                           sep='\n', dtype=np.int64)
print(wing_lengths)
```

This dataset is in a plain text file with one value per line, so we simply use NumPy's `fromfile()` function to load the data into an array, which we name `wing_lengths`. The `sep='\n'` option tells NumPy that this is a text file, with values separated by the new line character, `\n`. Finally, the `dtype=np.int64` option specifies that we want to treat the values as integers. The dataset is so small that we can print all the points, which we do by repeating the `wing_lengths` array name at the end of the cell.

Let's now generate a plot of the cdf for this data by running the following code in a cell:

```
plot_cdf(wing_lengths, plot_range=[30, 60],
          scale_to=100, lw=2)
plt.grid(lw=1)
plt.xlabel('Housefly wing length (x.1mm)', fontsize=18)
plt.ylabel('Percent', fontsize=18);
```

We, again, use the `plot_cdf()` function that was previously defined. Notice that now we use `scale_to=100` so that, instead of proportions, we can read a percentage in the vertical axis. We also add a grid and axis labels to the plot. We obtain the following plot:



Notice that the cdf has, roughly, the same symmetric S-shaped pattern that we observed before. This indicates that a Normal distribution might be an adequate model for this data. It is, indeed, the case that this data fits a Normal distribution quite closely.

Just as an example of the kind of information that can be extracted from this plot, let's suppose that we want to design a net that will catch 80% of the flies. That is, the mesh should allow only flies with a wing length in the bottom 20% to pass. From the preceding graph, we can see that the 20th percentile corresponds to a wing length of about 42 tenths of a millimeter. This is not intended to be a realistic application and if we were really building this net, we would probably want to do a more careful analysis, but it shows how we can get information about the data quickly from a cumulative distribution plot.

Working with distributions

The main reason that we emphasize the cumulative distribution function is because, once we have access to it, we can compute any probabilities associated with the model. This is because the cdf is a universal way to specify a random variable. In particular, there is no distinction between the descriptions for **continuous** or discrete data. The density function of a random variable is also an important concept so we will present it in the next section. In this section, we will see how to use the cdf to do computations related to a random variable.

The functions that we will use with distributions are part of SciPy and contained in the `scipy.stats` module, which we import with the following code:

```
import scipy.stats as st
```

After this, we can refer to functions in the package with the abbreviation `st`.

This module contains a large number of predefined distributions, and we encourage you to visit the official documentation at <http://docs.scipy.org/doc/scipy/reference/stats.html> to see what is available. Fortunately for us, the module is organized in such a way that all distributions are handled in a uniform way, as shown in the following line:

```
st.<rv_name>.<function>(<arguments>)
```

The components of this expression are as follows:

- `st` is the abbreviation that we chose for the stats package.
- `<rv_name>` is the name of the distribution (`rv` stands for random variable).
- `<function>` is the specific function that we want to calculate.
- `<arguments>` are the values that need to be passed to each function. These might include the shape parameters for each distribution as well as other required parameters that depend on the function being called.

The following table lists some of the functions that are available for each random variable:

Function	Description
<code>rvs()</code>	Random variates, that is, pseudorandom number generation
<code>cdf()</code>	Cumulative distribution function
<code>pdf()</code> or <code>pmf()</code>	Probability density function (for continuous variables) and probability mass function (for discrete variables)

ppf ()	Percent point function, the inverse of the cumulative distribution function
stats ()	Compute statistics (moments) for distribution
mean (), std (), or var ()	Compute mean, standard deviation, and variance, respectively
fit ()	Fit data to the distribution and return the parameters for the shape, location, and scale parameters from the data

Whenever we specify a distribution, we need to set the parameters that characterize the random variable of interest. Most models have, at least, a *location* and *scale*, which are commonly referred to as the *shape* parameters of the random variable. The location specifies a shift in the distribution and the scale represents a rescaling of the values (such as when units are changed).

In the following example, we will concentrate on Normal distribution as this is certainly an important case. However, you should be aware that the computation patterns we present can be used with *any* distribution. So, if one needs, for example, to work with a Log-Laplace distribution, all they have to do is replace `norm` in the following examples with `loglaplace`. Of course, it is necessary to consult the documentation to make sure that the correct parameters are being used for the distribution of interest.



An invaluable resource for statistical techniques is the NIST Engineering Statistics Handbook, available at <http://www.itl.nist.gov/div898/handbook/index.htm>. Section 1.3.6, *Probability Distributions*, contains an excellent introduction to random variables and distributions. Another quick reference is the Wikipedia list of probability distributions, located at http://en.wikipedia.org/wiki/List_of_probability_distributions.

For a Normal distribution, the location parameter gives the *mean* of the distribution and the scale represents the *standard deviation* of the distribution. These terms will be defined later in the chapter, but they are numbers that measure the center and spread of the distribution. To give a concrete example of a normally distributed random variable, let's consider the height of women over 20 years of age, as in National Health Statistics Reports, Number 10, October 2008, available online at <http://www.cdc.gov/nchs/data/nhsr/nhsr010.pdf>. This report contains anthropometric reference data for the U.S. population, according to age group. On page 14, the height of women over 20 is reported as having a mean of 63.8 inches with a standard error of 0.06. The sample size is 4,857.

We will assume that heights are normally distributed (which, as it happens, is a reasonable assumption). To characterize the distribution, we need the mean and standard deviation. The mean is given directly in the report. The standard deviation is not reported directly but can be computed from the standard error and sample size, according to the following formula:

$$\text{standard error} = \frac{\text{standard deviation}}{\sqrt{N}}$$

The standard error is a measure of sample variability that will be introduced in Chapter 4, *Regression*. We are also ignoring the issue that we are using the *sample standard deviation* instead of the population standard deviation, but the sample is large enough to justify this approach, as will be seen in Chapter 4, *Regression*. We start by defining the distribution to be used as the model for this situation, using the following code:

```
N = 4857
mean = 63.8
serror = 0.06
sdev = serror * np.sqrt(N)
rvnrm = st.norm(loc=mean, scale=sdev)
```

In this code, we first define variables to represent the sample size, mean, and standard error. Then, the standard deviation is computed according to the preceding formula. In the last line of code, we call the `norm()` function, passing the mean and standard deviation as parameters. The object returned by the function is assigned to the `rvnrm` variable, and it is through this variable that we access the functionality of the package.



It is possible to do all calculations without constructing an object first, but this is the recommended approach if we want to do several computations related to the same random variable.

A good place to start with any distribution is to make graphs so that we can have an idea of the shape of the distribution. This can be accomplished with the following code:

```
xmin = mean-3*sdev
xmax = mean+3*sdev
xx = np.linspace(xmin,xmax,200)
plt.figure(figsize=(8,3))
plt.subplot(1,2,1)
plt.plot(xx, rvnrm.cdf(xx))
```

```
plt.title('Cumulative distribution function')
plt.xlabel('Height (in)')
plt.ylabel('Proportion of women')
plt.axis([xmin, xmax, 0.0, 1.0])
plt.subplot(1,2,2)
plt.plot(xx, rvnorm.pdf(xx))
plt.title('Probability density function')
plt.xlabel('Height (in)')
plt.axis([xmin, xmax, 0.0, 0.1]);
```

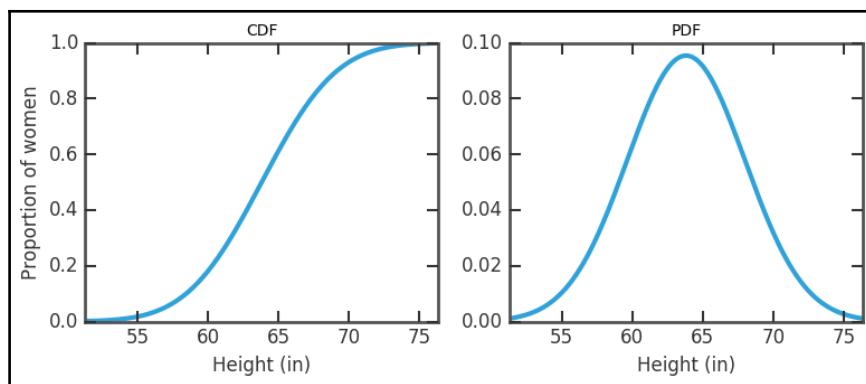
Most of the lines of the preceding code deal with setting up and formatting the plot. The two most important pieces of code are the two calls to the `plot()` function. The first call is as follows:

```
plt.plot(xx, rvnorm.cdf(xx))
```

The `xx` array was previously defined to contain the x coordinates to be plotted. The `rvnorm.cdf(xx)` function call computes the value of the cdf. The second call is analogous as follows:

```
plt.plot(xx, rvnorm.pdf(xx))
```

The only difference is that now we call `rvnorm.pdf(xx)` to compute the probability density function. The overall effect of the code is to generate side-by-side plots of the cdf and density function. These are the familiar S-shaped and bell-shaped curves, which are characteristic of a Normal distribution:



Let's now see how we can use the cdf to compute quantities related to the distribution. Suppose that a garment factory uses a classification of women's heights, as indicated in the following table:

Size	Height
Petite	59 inches to 63 inches
Average	63 inches to 68 inches
Tall	68 inches to 71 inches

What percentage of women are in the *Average* category? We can get this information directly from the cdf. Recall that this function gives the proportion of data values up to a given value. So, the proportion of women with heights up to 68 inches is computed by the following expression:

```
rvnorm.cdf(68)
```

Likewise, the proportion of women with heights up to 63 inches is computed with the following expression:

```
rvnorm.cdf(63)
```

The proportion of women with heights between 63 inches and 68 inches is the difference between these values. As we want a percentage, we have to multiply the result by 100, as shown in the following line of code:

```
100 * (rvnorm.cdf(68) - rvnorm.cdf(63))
```

The result of this computation shows that approximately 42.8% of women are *Average* according to this classification. To compute the percentages for the three categories and display them in a nice format, we can use the following code:

```
categories = [
    ('Petite', 59, 63),
    ('Average', 63, 68),
    ('Tall', 68, 71),
]
for cat, vmin, vmax in categories:
    percent = 100 * (rvnorm.cdf(vmax) - rvnorm.cdf(vmin))
    print('{:>8s}: {:.2f}'.format(cat, percent))
```

In this code segment, we start by creating a Python list that describes the categories. Each category is represented by a three-element tuple, containing the category name, minimum height, and maximum height. We then use a `for` loop to iterate over the categories. For each category, we first compute the percentage of women in the category using the cdf, and then print the result.

A somewhat unexpected feature of this classification is that it imposes the lowest and highest values on women's heights. We can compute the percentage of women that are too short or too tall to fit in any of the categories with the following code:

```
too_short = 100 * rnorm.cdf(59)
too_tall = 100 * (1 - rnorm.cdf(71))
unclassified = too_short + too_tall
print(too_short, too_tall, unclassified)
```

Running this code, we conclude that almost 17% of the women are unclassified! This may not seem too much, but any sector of industry that neglects such a portion of its customer base will be losing profits. Suppose that we have been hired to come up with a more effective classification. Let's say that we agree that 50% of women at the center of the distribution should be classified as *Average*, the top 25% should be considered *Tall*, and the lower 25% should be considered *Petite*. In other words, we want to use the quartiles of the distribution to define height categories. Notice that this is an arbitrary decision and may not be realistic.

We can find the threshold values between the categories using the inverse of the cdf. This is computed by the `ppf()` method, which stands for **percent point function**. This is shown in the following code:

```
a = rnorm.ppf(0.25)
b = rnorm.ppf(0.75)
print(a, b)
```



This computation shows you how to compute the first and third quartiles of any distribution. In general, to find the percentile c of the distribution, we can use the `rnorm.ppf(c/100.)` expression.

From the preceding computation, it follows that, according to our criterion, women should be considered *Average* if their heights are between approximately 61 inches and 67 inches. It seems that the original classification is skewed toward taller women (there is a large proportion of short women that do not fit the classification). We can only speculate why this is the case. Industry standards are sometimes inherited from tradition and may have been set without regard to actual data.

It is worthy to notice that, in all the computations we did, we used the *cumulative distribution function* and not the probability density function, which we will present in the next section. Indeed, this will be the case with most computations that are needed in inference. The fact that most people prefer to use probability densities when defining distributions may simply be due to historical bias. In fact, *both* views of a random variable are important and find a place in theory and applications.

To finish this example, let's compute some relevant parameters (that is, point estimates) for the distribution of heights using the following code:

```
mean, variance, skew, kurtosis = rvnorm.stats(moments='mvks')
print(mean, variance, skew, kurtosis)
```

This will print values of 63.8 for the mean, 17.4852 for the variance, and 0 for both skew and kurtosis. These values are interpreted as follows:

- The mean is the average value of the distribution. As the distribution is symmetric, it coincides with the median.
- The variance is the square of the standard deviation. It is defined as the average value of the square of the deviation from the mean.
- The skew measures the asymmetry of the distribution. As the Normal distribution is symmetric, the skew is zero.
- The kurtosis indicates how the distribution peaks: does it have a sharp peak or a flatter bump? The value of kurtosis for the Normal distribution is zero because it is used as a reference distribution.

For our next example, let's suppose that we need to build a simulation for the time-to-failure of some equipment. A frequently used model for this situation is the Weibull distribution, named after Waloddi Weibull, who carefully studied the distribution in 1951. This distribution is described in terms of two numbers called the `scale` parameter, denoted by η (eta) and the `shape` parameter, denoted by β (beta). Both parameters must be positive numbers.



There is a three-parameter version of the Weibull distribution that introduces a location parameter. We assume that the equipment can fail from the start of its operation, so the location parameter is zero and can be ignored.

The `shape` parameter is related to how the failure rate of the equipment depends on its age (or operation time) as follows:

- If the shape parameter is smaller than 1, then the failure rate decreases as time passes. This occurs, for example, if there is a significant number of items that are defective and tend to fail early when put in use.
- If the shape parameter is equal to 1, the failure rate is constant in time. This is the well-known **exponential distribution**. In this model, the chance that the equipment fails in a given interval does not depend on how long it has been operating. This is an unrealistic assumption in most cases.

- If the shape parameter is larger than 1, then the failure rate of the equipment increases as time passes. This is reflective of an aging process, in which older equipment is more prone to failure.

The scale parameter, on the other hand, determines how much the distribution spreads. Put in an intuitive way, a larger value of the scale parameter corresponds to more uncertainty regarding predictions for the failure time. Notice that, in this case, the scale parameter cannot be interpreted as the standard deviation of the model.

Let's say that we want to simulate a Weibull distribution with the shape parameter β and scale parameter η . You are encouraged, at this point, to generate plots of the cumulative distribution and probability density functions for the Weibull distribution by modifying the code that we previously used for the Normal distribution. To create the distribution, the following code can be used:

```
eta = 1.0
beta = 1.5
rvweib = st.weibull_min(beta, scale=eta)
```

After defining the eta and beta parameters, we call the `weibull_min()` function, which generates the appropriate object. After producing the graphs, you will notice that the Weibull distribution is markedly asymmetric, having a long right tail after peaking.

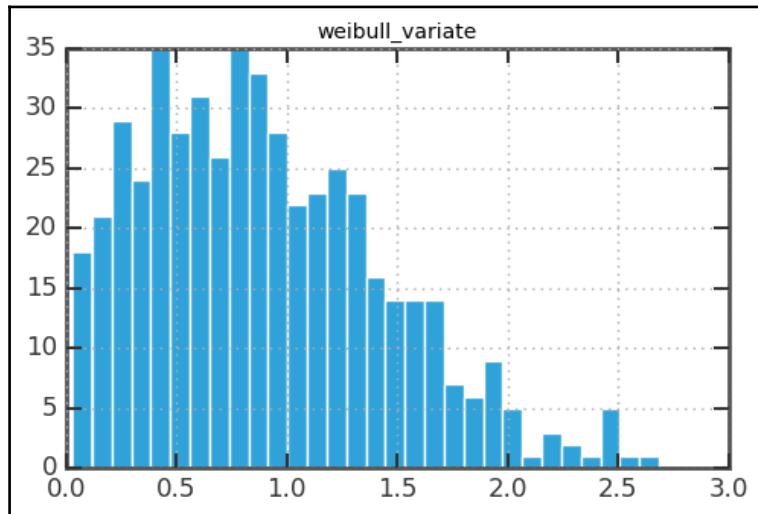
Let's now get back to the problem of simulating the distribution. To produce a sample of size 500 following a Weibull distribution, we use the following code:

```
weib_variates = rvweib.rvs(size=500)
print(weib_variates[:10])
```

In this code, we simply call the `rvs()` method, passing the desired sample size as an argument. The abbreviation `rvs` stands for **random variates**. As the generated sample is quite large, we simply print the first 10 values. We can use a histogram to visualize the sample, as shown in the following code:

```
weib_df = DataFrame(weib_variates, columns=['weibull_variate'])
weib_df.hist(bins=30);
```

In this code, we first convert the data to a Pandas DataFrame object as we want to use Pandas' plotting capabilities. Then, we simply call the `hist()` method, passing the number of bins as an argument. This results in the following histogram:



Notice the peak around 0.5 followed by a decaying tail toward the right. Also notice the few values at the extreme right of the histogram. These represent survival times that are large compared with the bulk of the data.

Finally, we want to assess how good the simulation is. To this end, we can plot the cumulative distribution function of the sample, compared with that of the theoretical distribution. This is achieved with the following code:

```
xmin = 0
xmax = 3.5
xx = np.linspace(xmin,xmax,200)
plt.plot(xx, rvweib.cdf(xx), color='orange', lw=5)
plot_cdf(weib_variates, plot_range=[xmin, xmax], scale_to=1, lw=2,
color='green')
plt.axis([xmin, xmax, 0, 1])
plt.title('Weibul distribution simulation', fontsize=14)
plt.xlabel('Failure Time', fontsize=12);
```

This code is essentially a combination of code segments that we have seen before. We call the `plot()` function with appropriate arguments to generate a plot of the theoretical cdf, and then use the `plot_cdf()` function, previously defined in this chapter, to plot the cdf of the data. It is seen that the two curves show pretty good agreement.

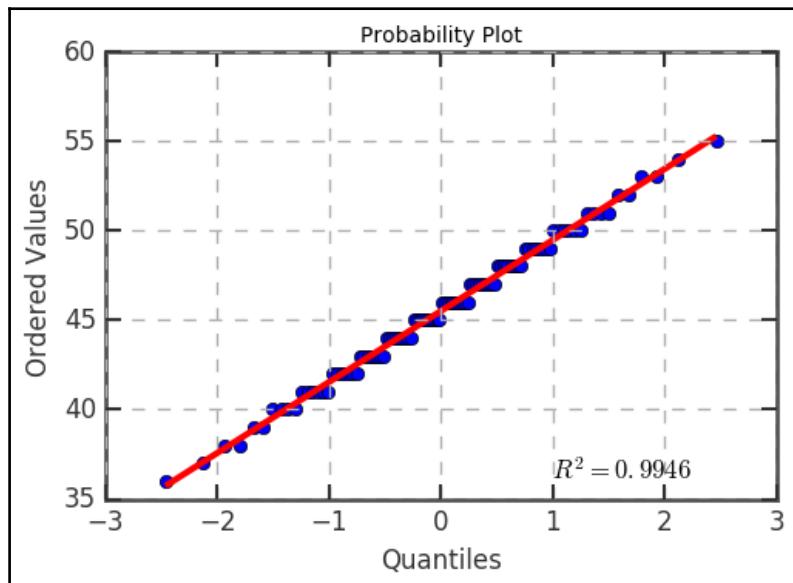
As a final example in this section, let's explore the `fit()` method, which attempts to fit a distribution to the data. Let's go back to the dataset with wing lengths of houseflies. As previously stated, we suspect that the data is normally distributed. We can find the Normal distribution that *best fits* the data with the following code:

```
wing_lengths = np.fromfile('data/housefly-wing-lengths.txt',
                           sep='\n', dtype=np.int64)
mean, std = st.norm.fit(wing_lengths)
print(mean, std)
```

Just to be on the safe side, we read the data again using the `fromfile()` function. We then use the `st.norm.fit()` function to fit a Normal distribution to the dataset. The `fit()` function returns the mean and standard deviation of the fitted Normal distribution. This is an example of parameter estimation (point estimate). The next question is: *how good is the fit?* We can evaluate this graphically by generating a quantile plot, as shown in Chapter 2, *Exploring Data*. The following is the code:

```
st.probplot(wing_lengths, dist='norm', plot=plt)
plt.grid(lw=1.5, lw='dashed');
```

This code will produce the following plot:



Notice that the sample falls very close to a straight line, indicating that a Normal distribution is a good fit for this data. Notice, however, that there is some clustering in the data, which is a consequence of repeated values (measurement imprecision and rounding).



Fitting a model to a sample is a complex topic and should be approached with care. In this section, we concentrated on learning how to use the tools provided by NumPy and SciPy, without going deeply into the question of how adequate are the methods that we used. A more careful discussion of some of the topics appears in the remaining chapters of the book.

We finalize this section by encouraging you to visit the SciPy documentation for the `stats` module. This is an extensive module that encompasses a great deal of functionality. The documentation is very well-organized and comprehensive and includes a discussion of the methods used and relevant links to the theory where adequate.

The probability density function

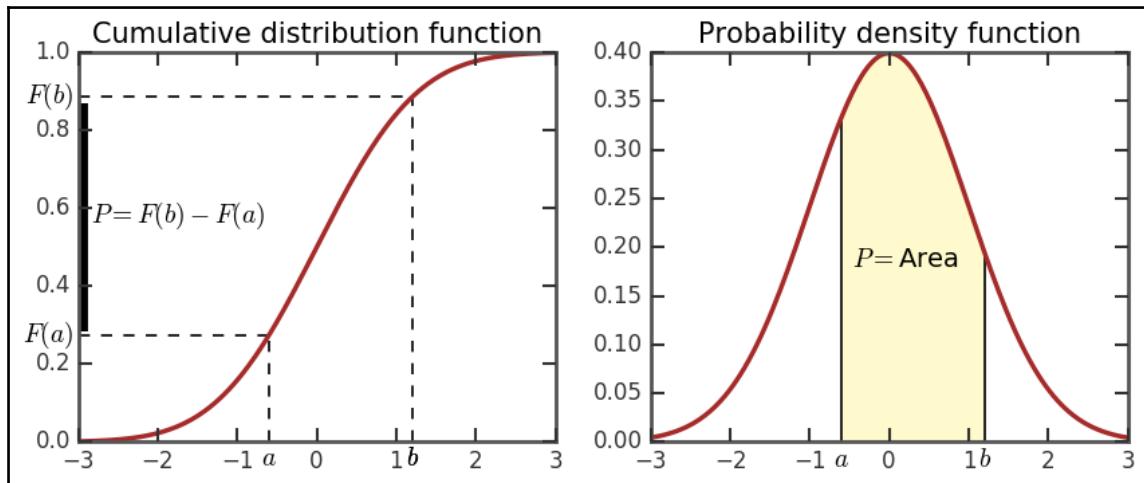
So far, we have considered the cumulative distribution function as the main way to describe a random variable. However, for a large class of important models, the **probability density function (pdf)** is an important alternative characterization.

To understand the distinction between the cdf and pdf, we need the notion of probability. In the context of random variables, probability simply means the likelihood that the random outcome falls within a certain range of values, normalized to a number between 0 and 1. For example, let's consider the example of women's heights discussed in the previous section. We concluded that 42.8% of women have a height between 63 inches and 68 inches. An alternative way to express this is to say that, for the random variable that represents women's heights, *the probability that the outcome is between 63 and 68 is .428*.

The main distinction between the cdf and pdf is the way probabilities are represented by each of them:

- For a cdf, the probability that the outcome is in a range is computed as the *difference* between the values of the cdf at the endpoints of the range
- For a pdf, the probability that the outcome is in a range is computed as the *area* under the curve determined by the range

To clarify these concepts, let's consider the following figure:



This figure displays both the cdf and pdf for a standard Normal distribution. In both figures, we have a range of values on the horizontal axis defined by the values **a** and **b**. The figure graphically illustrates the probability that the outcome falls in this range in each case:

- In the case of the cdf, the probability is given by $F(b)-F(a)$, which corresponds to the length of the highlighted segment on the y axis
- In the case of the cdf, the probability is given by the *area* bound by the curve between the values **a** and **b**

This observation actually explains why the cdf is more useful computationally. To compute the probability using the cdf, all we need is the difference between two values, while to do the same calculation using the pdf, it is necessary to find the value of an area. As this is not a simple shape, we need calculus to compute the area. In fact, in the case of the Normal distribution, this is an area that cannot be computed even by the methods usually seen in calculus courses! Of course, this complexity still exists when we do computations in Python, but the details are fortunately hidden from us.

This is a good time to mention briefly how the pdf is related to significant characteristics of a distribution such as mean and standard deviation. When talking about a random variable, the notion of *average* is technically called the **expected value** or **mean** of the random variable. Intuitively, this is the average of the values that we expect to see if a large number of trials with the same distribution is observed. Likewise, the variance of the random variable is the average squared deviation from the mean.

Finally, the *standard deviation* is the square root of the variance. Unfortunately, to give a mathematical definition of these notions for a continuous random variable, we again need calculus. As this book concentrates on the practical application of Python to data analysis, we will be content with the intuitive meaning of these concepts and let the computer do the dirty computational work under the hood.

We finalize by pointing out that we have, in this section, concentrated on a continuous distribution characterized by a smooth cdf. On the other extreme are *discrete distributions*, which have a cdf that looks like a *staircase*, much like the examples seen in the previous section. Discrete random variables cannot be represented by a pdf. Instead, they are defined in terms of a **probability mass function (pmf)**. An example of a discrete random variable is considered in the next section.

Where do models come from?

In this section, we will consider what is perhaps the most important practical point about models. How are models conceived and how do we know what is the right model to use in a given situation?

These are not simple questions, and the process of designing and choosing appropriate models is as much an art as a science. At the risk of oversimplifying, we could say that probabilistic models can come from two sources:

- In *priori models*, the researcher considers the relevant factors, identifies important quantities and relationships, and creates a description that fits the problem being considered
- In *limit models*, the researcher attempts to find an approximation to a model that is too complex, either conceptually or computationally

In both cases, the resulting model may take several different forms. It can be, for example, a mathematical formula, simulation, or algorithm. Always, the model must be validated against real data after the experiments or observations are carried out.

The most important point to emphasize is that all models have *assumptions*. These establish the boundaries within which the model is valid. Identifying the assumptions that are made is probably the first important step in successful model selection.

As examples, we will consider two models with both historical and practical importance: the *Binomial distribution* and *de Moivre approximation*.

The Binomial distribution is an example of a discrete random variable that was originally conceived in the context of gambling but has wide applicability to many situations. Here are the assumptions that underlie the model:

- A series of N trials is conducted, where each trial can only have one of two results. We will call the two possible outcomes 0 and 1 (failure and success).
- Each trial has a known probability p of having the outcome 1 and a corresponding probability $1-p$ of having the outcome 0.
- The trials are independent, in the sense that the result of each trial is not affected by the results of the other trials.
- The variable being observed is the number of outcomes equal to 1 in the N trials.

It is easy to see why this model appeals to gamblers: in a game of chance, one can either win, represented by 1 or lose, represented by 0. The probability of each outcome is known. The gambler plays the game repeatedly and is interested in knowing how much money will be made, which is determined by the number of wins.

In a more practical application, we can think of a quality control system. In this case, 1 represents a good item and 0 represents a defective item. We know the probability p of an item being good or defective and want to know the variability of the number of good items when N items are produced.

To be concrete, let's say that we are playing a *fair* game in which the probability of winning and losing are both 0.5. We assume that 20 games are played. The Binomial distribution is also part of the `scipy.stats` module, and we can create an object representing the distribution with the following code:

```
N = 20  
p = 0.5  
rv_binom = st.binom(N, p)
```

As the game is played 20 times, the number of wins in the series is an integer between 0 and 20. To compute the probability that we win 12 out of 20 games, for example, we use the probability mass function, as indicated by the following expression:

```
rv_binom.pmf(12)
```

Evaluating this code, we conclude that the probability of winning exactly 12 out of 20 games is about 0.12 or 12%.

In many cases, we are not interested in the probability of an exact number of wins but on the probability of a range. For example, let's suppose that on a particular day, we win only 7 out of 20 games and wonder if we have been cheated. One way to assess this is to compute the probability of winning seven or fewer games. If this probability is small, it is likely that the assumption that the game is fair is not valid. To compute this probability, we need the cumulative distribution function, which can be computed with the following line of code:

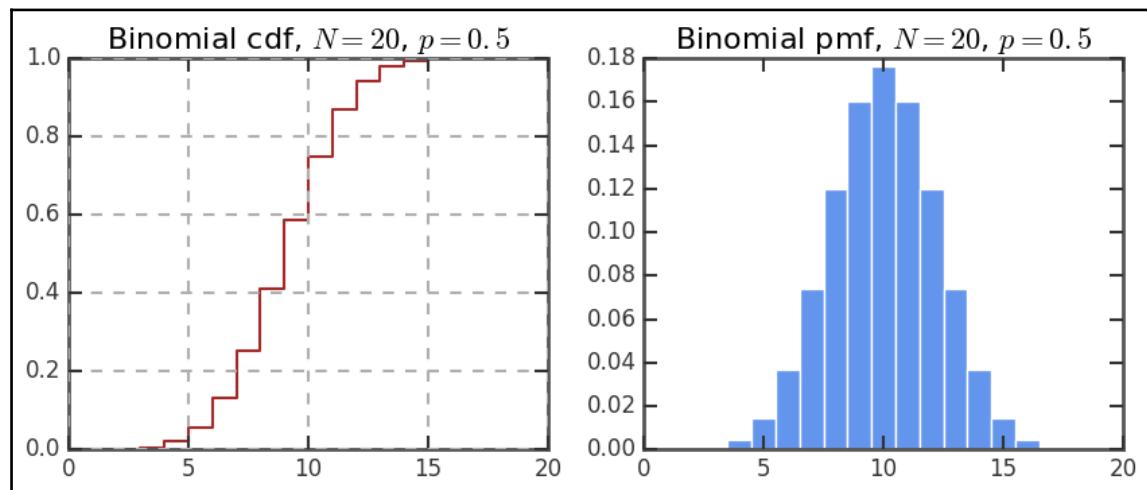
```
rv_binom.cdf(7)
```

The result shows that this event happens with a probability of approximately 0.13, that is, 13% of the time. This is not such a small number, so we would expect that, from time to time, we would actually win only 7 of the 20 games, even in a fair game. So, there does not seem to be reason to suspect cheating, at least in this isolated case.

To get an idea of how the distribution behaves, let's make plots of the `cdf` and `pmf`. This can be done with the following code:

```
xx = np.arange(N+1)
cdf = rv_binom.cdf(xx)
pmf = rv_binom.pmf(xx)
xvalues = np.arange(N+1)
plt.figure(figsize=(9, 3.5))
plt.subplot(1, 2, 1)
plt.step(xvalues, cdf, lw=2, color='brown')
plt.grid(lw=1, ls='dashed')
plt.title('Binomial cdf, $N=20$, $p=0.5$', fontsize=16)
plt.subplot(1, 2, 2)
left = xx - 0.5
plt.bar(left, pmf, 1.0, color='CornflowerBlue')
plt.title('Binomial pmf, $N=20$, $p=0.5$', fontsize=16)
plt.axis([0, 20, 0, .18]);
```

This code is similar to the methods that we used in the graphs previously displayed in this chapter, but we repeat it here as you might find it useful to have a model to make plots of discrete distributions. For the `cdf`, we simply plot the `xvalues` and `yvalues` arrays, which contain the stairs of the `cdf` taken from the `cdf` method of the `rv_binom` object. As we want it to be displayed as such (that is, a staircase), we use the `step` function to plot it. For the `pmf`, we use a slightly different approach, the `bar()` function, which is a `matplotlib` function that draws a generic bar chart. The arguments for this function are two arrays containing the left coordinate and height of each bar and a number specifying the width of the bars. The plots are displayed in the following figure:



There is an important point to notice here: the pmf is a function that is defined only for the integer values from 1 to 20. However, for visualization purposes, instead of just plotting the discrete points, we plot bars with width one since the data is discrete. Each bar is centered at the integer value that corresponds to the number of *wins*. With these choices, the probabilities in the pmf correspond to the areas of the bars, which is the same interpretation for the pdf of a continuous distribution.

You probably noticed that there is a striking similarity between the preceding plots and Normal distribution. The French mathematician, de Moivre, was the first one to notice that the plot of the pmf for the Binomial distribution approximates a smooth curve if the number of trials N is large. He realized that, if he were able to find a formula for this curve, he would have a simpler way to calculate binomial probabilities for a large number of trials. He was able to find the formula for the curve and, using this formula, he computed binomial probabilities for 3,600 trials, a remarkable feat at the time. This was the birth of the Normal distribution.

To understand the de Moivre approximation, we must first compute the mean and standard deviation of the Binomial distribution. We will do this in two ways. First, let's use the functions provided in `scipy.stats`, as indicated in the following code:

```
mean = rv_binom.mean()  
std = rv_binom.std()  
print(mean, std)
```

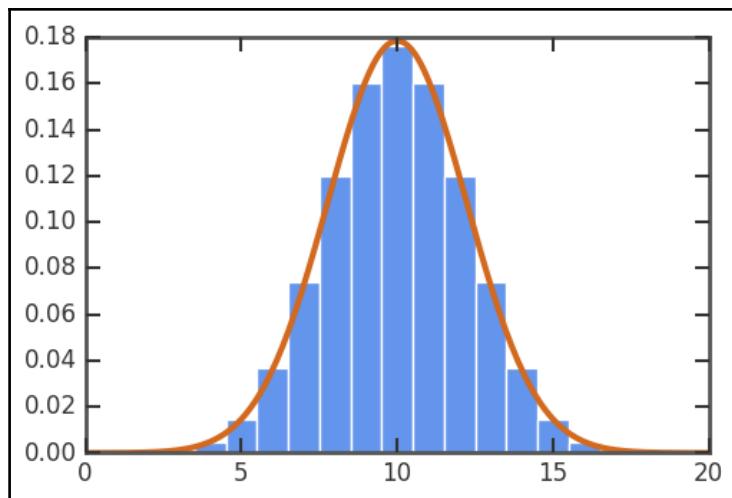
Here, we are simply calling the `mean()` and `std()` methods to compute the mean and standard deviation. An alternative approach is to use the theoretical formulas, as shown in the following code:

```
mean = N * p  
std = np.sqrt(N * p * (1 - p))  
print(mean, std)
```

Either way, we get 10.0 for the mean and approximately 2.236 for the standard deviation. The de Moivre approximation theorem can be informally stated as follows:

For large N , a Binomial distribution is approximated by a Normal distribution with the same mean and standard deviation.

The following figure shows the pmf of the Binomial distribution superimposed with a plot of the Normal distribution with the same mean and standard deviation. It can be seen that the agreement is remarkable:



Multivariate distributions

So far in this chapter, we considered only the case of a random experiment that has a single numeric outcome. Within this framework, we can model only a single variable. In most data analysis problems, we may be interested in relationships between variables. For example, we might want to understand the relation between the height and weight of a person or between income and educational levels. In another situation, we may be observing a variable repeatedly. As an example, we might be interested in the daily snowfall in a region during the winter months.

To handle these situations, we need models described by *multivariate distributions*. We have the analogous of the cdf and pdf (or pmf for discrete distributions), but now we have to use functions depending on several variables. The univariate distributions that we discussed in the previous sections are used as building blocks, but we have the extra complication of having to specify how the different variables interact with each other.

A typical example is the bivariate Normal distribution. In this model, we observe two random variables that, individually, are normally distributed. Each of the two variables is characterized by its own mean and standard deviation. However, we must also say how the two variables interact.

In the simplest case, the outcome of one of the variables has no influence whatsoever on the outcome of the other. Consider, for example, the relationship between the snowfall in London and the score of a soccer match in Sidney. Unless we believe in some kind of supernatural connection, we don't expect there to be any connection between these variables. In this case, we say that the variables are *independent*.

On the other hand, and more interestingly, the variables may be *correlated*; in the sense that the result of one of the observations will affect the probabilities for the other. For example, we expect the weight and height of a person to be correlated. In this case, we will be interested in knowing how strong the correlation is and perhaps use one of the variables to make predictions about the other.

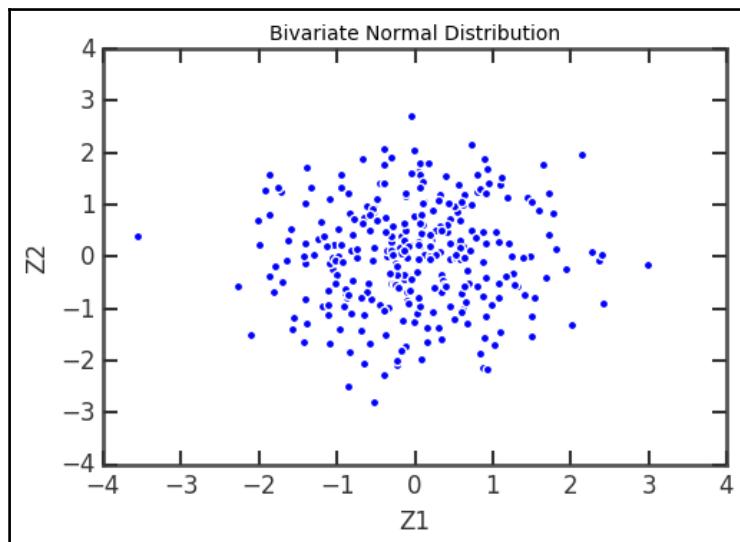
The multivariate Normal distribution is also part of the `scipy.stats` package. For now, we will just see how to generate random variates according to a bivariate Normal distribution. We will look at multivariate distributions in detail later in the book. Let's run the following code to generate random variables:

```
binorm_variates = st.multivariate_normal.rvs(mean=[0,0], size=300)
df = DataFrame(binorm_variates, columns=['Z1', 'Z2'])
df.head(10)
```

In this code, we are using the `multivariate_normal.rvs()` function to generate a sample of size 300 from a bivariate normal with the mean zero and default covariance 1. We then convert the NumPy array to a Pandas DataFrame and print the first 10 components of the DataFrame. We can now create a scatterplot of the variates using the following lines of code:

```
df.plot(kind='scatter', x='Z1', y='Z2')
plt.title('Bivariate Normal Distribution')
plt.axis([-4,4,-4,4]);
```

Here, we are using the `plot()` method of the `df` object. The `kind=scatter` option is used to produce a scatterplot. We have to specify the `x` and `y` components for the scatterplot with the corresponding options. After that, we set the title of the plot and adjust the axis ranges:



Summary

In this chapter, you learned about the basic models used in data analysis. We studied how the cumulative distribution function and probability density function can be used to characterize a random variable and how to do computations using these tools, including calculating means, standard deviation, and variance and generating random variates.

We have seen examples of continuous and discrete random variables and studied two important cases: the Binomial distribution and its approximation by a Normal distribution. The chapter concludes with an overview of multivariate distributions.

In the next chapter, we will take a closer look at various ways of doing regression.

4

Regression

Linear regression is part of the general introduction to experimental techniques; it forms the base for many of the scientific breakthroughs for the last few centuries. We made some short dives into linear regression before, looking at Hubble's law among other things. The previous chapter consisted of looking at distributions, which is an integral part of exploratory data analysis and indeed one of the first steps in gaining insights into the data. All of the things that we have gone through so far are, as you will see, useful in this chapter as well. You are strongly encouraged to experiment and try out these new things in combination with what was learned in the previous chapters. In this chapter, we will cover the following forms of regression:

- Linear regression
- Multiple regression
- Logistic regression

In the simplest formulation, linear regression deals with estimating a variable from another variable. In multiple regression, a variable is estimated from two or more others. This, of course, only works when the variables have some kind of correlation between them. It's important to point out at this point is that correlation does not imply causation; just because two or more variables show a dependence on one another, it does not mean that they actually affect/depend each other in real life. Logistic regression fits models to one or more discrete variables, which are sometimes binary (that is, can only take the values 0 or 1).

In this chapter, we will start with a quick introduction to linear regression, then we dive directly into getting the data and testing a hypothesis of a simple relationship between two variables. After this, the extension to multiple variables is covered, where we simply add data to what we have from the previous section. Logistic regression is covered in the last part of the chapter.

Curiosity is strongly encouraged, and use what we have learned in the previous chapters to explore the data. Before running the examples in this chapter, start the Jupyter Notebook and run the default imports.

Introducing linear regression

The simplest form of linear regression is given by the relation $y = k_x + k_0$, where k_0 is called **intercept**, that is, the value of y when $x=0$ and k is the slope. A general expression for this could be found by thinking of each point as the preceding relation plus an error ε . This would then look, for N points, as follows:

$$y_1 = kx_1 + k_0 + \varepsilon_1$$

$$y_2 = kx_2 + k_0 + \varepsilon_2$$

⋮

$$y_{N-1} = kx_{N-1} + k_0 + \varepsilon_{N-1}$$

$$y_N = kx_N + k_0 + \varepsilon_N$$

We can express this in matrix form:

$$\mathbf{Y} = \mathbf{X}\mathbf{k} + \boldsymbol{\epsilon}$$

Here, the various matrices/vectors are represented as follows:

$$\mathbf{Y} = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}$$

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 \\ 1 & \vdots \\ 1 & x_N \end{bmatrix}$$

$$\mathbf{k} = \begin{bmatrix} k_0 \\ k \end{bmatrix} \boldsymbol{\epsilon} = \begin{bmatrix} \epsilon_1 \\ \vdots \\ \epsilon_N \end{bmatrix}$$

Performing the multiplication and addition of the matrix and vectors should yield the same set of equations that are defined here. The goal of the regression is to estimate the parameters, \mathbf{k} , in this case. There are many types of parameter estimation methods—ordinary least squares being one of the most common—but there are also maximum likelihood, Bayesian, mixed model, and several others. In ordinary least-squares minimization, the square of the residuals are minimized, that is, $r^T r$ is minimized (T denotes the transpose and r denotes the residuals, that is $Y_{\text{data}} - Y_{\text{fit}}$). It is nontrivial to solve the matrix equation; however, it is instructive to at least do this once if time permits. In the following example, we will use the least-squares minimization method. Most of the time, underlying computations use matrices to calculate the estimates of the parameters and uncertainty in the determination. What also comes out of the analysis is how correlated the variables are, that is, how likely it is that a linear relation exists between them.

Getting the dataset

Before we start estimating parameters of linear relationships, we need a dataset. In this first example, we will look at the suicide rate data from the **World Health Organization (WHO)** at <http://www.who.int>. One very important and complex part of data analysis is getting the data into manageable data structures appropriate for our analysis. Therefore, we will see how to get the data and map it to our desired data structures. The first part of the dataset is age-standardized suicide rates (per 100,000 inhabitants) per country and gender.



Age-standardization (also called age-adjustment) is a technique used to allow populations to be compared when the age profiles of the populations are different.

The following code downloads the data and stores it in a file:

```
import urllib.request  
payload='target=GHO/MH_12&profile=crosstable&filter=COUNTRY:*,REGION:*&x-  
sideaxis=COUNTRY&x-topaxis=GHO;YEAR;SEX'  
suicide_rate_url='http://apps.who.int/gho/athena/data/xmart.csv?'  
  
local_filename, headers =  
urllib.request.urlretrieve(suicide_rate_url+payload,  
filename='data/who_suicide_rates.csv')
```

The `urllib` module is part of the Python standard library (<https://docs.python.org/3/library>). If the filename input is not given, the file is stored in a temporary location on the disk. If problems arise, it is possible to also go directly to the URL and download the file. Alternatively, the OECD database contains suicide rates that also go back in time to 1960 (<http://stats.oecd.org>).

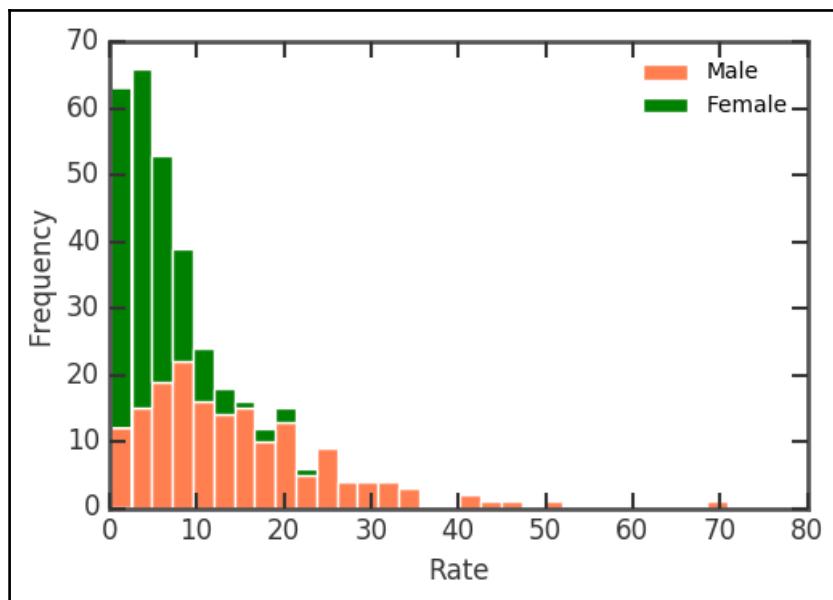
As before, we use Pandas' `read_csv` function from the Pandas data reader. Here, we give names to the columns; it is possible to send only the `header=2` parameter. This would tell you that the column names are given in the header; however, this might not always be what we want:

```
LOCAL_FILENAME = 'data/who_suicide_rates.csv_backup'  
rates = pandas.read_csv(LOCAL_FILENAME, names=['Country', 'Both', 'Female',  
'Male'], skiprows=3)  
rates.head(10)
```

	Country	Both	Female	Male
0	Afghanistan	5.7	5.3	6.2
1	Albania	5.9	5.2	6.6
2	Algeria	1.9	1.5	2.3
3	Angola	13.8	7.3	20.7
4	Argentina	10.3	4.1	17.2
5	Armenia	2.9	0.9	5.0
6	Australia	10.6	5.2	16.1
7	Austria	11.5	5.4	18.2
8	Azerbaijan	1.7	1.0	2.4
9	Bahamas	2.3	1.3	3.6

To make it easier for you, we tell it to skip the first three rows, where the metadata of the file is stored. As covered before, the CSV file format lacks a proper standard, hence it can be difficult to interpret everything correctly, so skipping the header makes it more robust in this case. However, experimenting with different input parameters is encouraged and can be very instructive. As shown in previous chapters, we start out by exploring this dataset:

```
rates.plot.hist(stacked=True, y=['Male', 'Female'],
                bins=30, color=['Coral', 'Green'])
plt.xlabel('Rate');
```

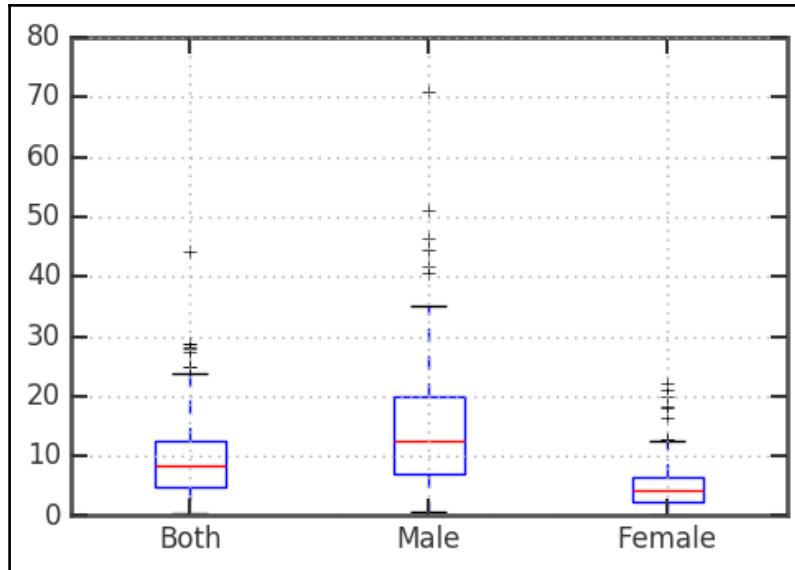


The histogram now plots the columns with names matching **Male** and **Female**, stacked on top of each other. It shows that the suicide rates are a bit different for males and females. Printing out the mean suicide rates for the genders shows that males have a significantly higher rate of suicide:

```
print(rates['Male'].mean(), rates['Female'].mean())
14.69590643274854 5.070602339181275
```

To look closer at some of the basic statistics of the rates, we use the boxplot command:

```
rates.boxplot();
```



As it's clear, the suicide rate is higher for men in comparison with women. Looking at the boxplot and also the combined distribution (that is, the **Both** key), we can see that there is one outlier (that is, crosses) with a really high rate. It has over 40 suicides per 100,000; which country/countries is/are this? Let's have a look:

```
print(rates[rates['Both']>40])
   Country Both Female Male
66   Guyana 44.2   22.1 70.8
```

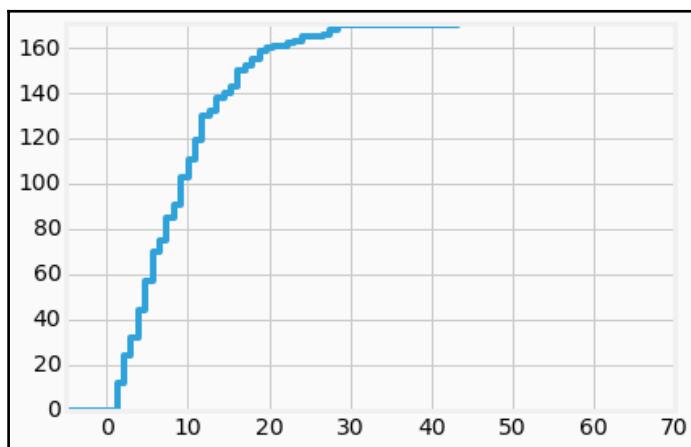
Here, we filter by saying that we want only the indices where the rates in the **Both** column are higher than 40. Apparently, the suicide rate in Guyana is very high. There are some interesting and, obviously, troublesome facts surrounding this. A quick web search reveals that while theories to explain the high rate have been put forth, studies are yet to reveal the underlying reason(s) for the significantly higher than average rate.

As seen in the preceding histograms, the suicide rates all have a similar distribution (shape). Let's first use the CDF plotting function defined in the examples in previous chapters:

```
def plot_cdf(data, plot_range=None, scale_to=None, nbins=False, **kwargs):
    if not nbins:
        nbins = len(data)
    sorted_data = np.array(sorted(data), dtype=np.float64)
    data_range = sorted_data[-1] - sorted_data[0]
    counts, bin_edges = np.histogram(sorted_data, bins=nbins)
    xvalues = bin_edges[1:]
    yvalues = np.cumsum(counts)
    if plot_range is None:
        xmin = xvalues[0]
        xmax = xvalues[-1]
    else:
        xmin, xmax = plot_range
    # pad the arrays
    xvalues = np.concatenate([[xmin, xvalues[0]], xvalues, [xmax]])
    yvalues = np.concatenate([[0.0, 0.0], yvalues, [yvalues.max()]])
    if scale_to:
        yvalues = yvalues / len(data) * scale_to
    plt.axis([xmin, xmax, 0, yvalues.max()])
    return plt.step(xvalues, yvalues, **kwargs)
```

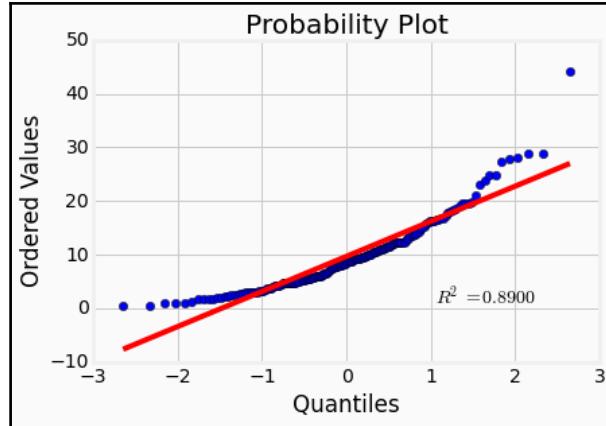
With this, we can again study the distribution of suicide rates. Run the function on the combined rates, that is, the Both column:

```
plot_cdf(rates['Both'], nbins=50, plot_range=[-5, 70])
```



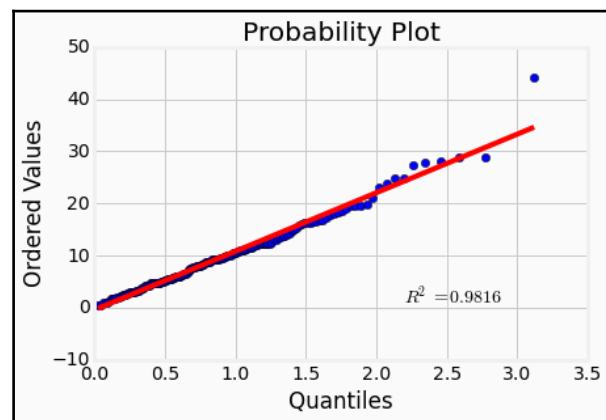
We can first test the normal distribution as it is the most common distribution:

```
st.probplot(rates['Both'], dist='norm', plot=plt);
```



Comparing what we saw previously when creating plots like this, the fit is not good at all. Recall the Weibull distribution from the previous chapter; it might fit better with its skew toward lower values. So let's try it:

```
beta = 1.5
eta = 1.
rvweib = st.weibull_min(beta, scale=eta)
st.probplot(rates['Both'], dist=rvweib, plot=plt);
```

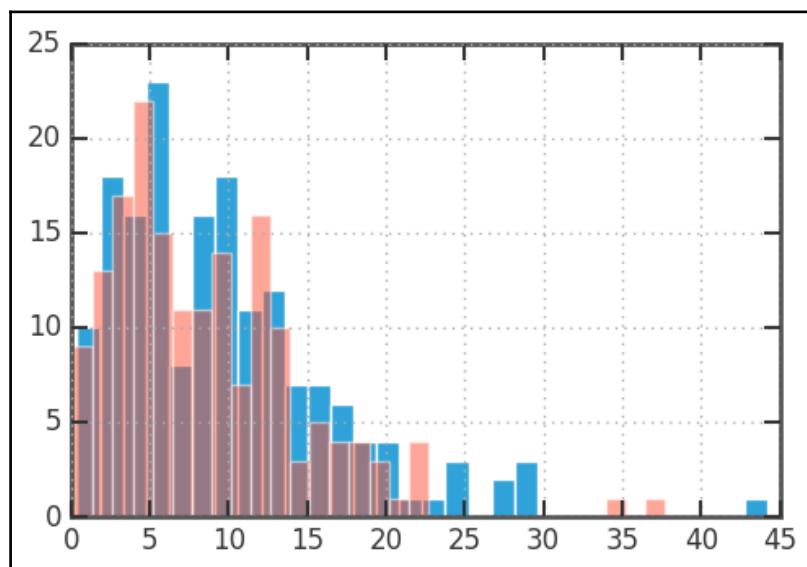


The Weibull distribution seems to reproduce the data quite well. The `r`-value, or the **Pearson correlation coefficient**, is a measure of how well a linear model represents the relationship between two variables. Furthermore, the `r-square` value given by `st.probplot` is the square of the `r`-value, in this case. It is possible to fit the distributions to the data. Here, we fix the location parameter to 0 with `floc=0`:

```
beta, loc, eta = st.weibull_min.fit(rates['Both'],
                                    floc=0, scale = 12)
```

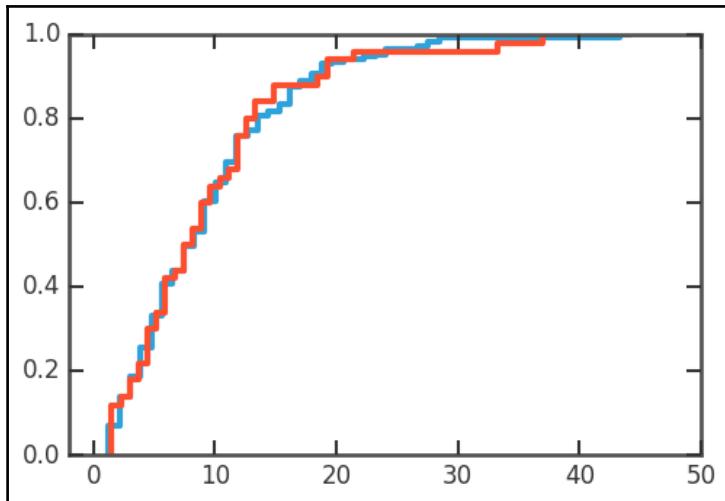
This gives `beta` of 1.49, `loc` of 0, and `scale` of 10.76. With the fitted parameters, we can plot the histogram of the data and overplot the distribution. I have included a fixed random seed value so that it should reproduce the results in the same way for you:

```
rates['Both'].hist(bins=30)
np.random.seed(1100)
rvweib = st.weibull_min(beta, scale=eta)
plt.hist(rvweib.rvs(size=len(rates.Both)),bins=30, alpha=0.5);
```



Then, comparing the CDFs of the Weibull and our data, it is obvious that they are similar. Being able to fit the parameters of a distribution is very useful:

```
plot_cdf(rates['Both'], nbins=50, scale_to=1)
np.random.seed(1100)
plot_cdf(rvweib.rvs(size=50), scale_to=1)
plt.xlim((-2,50));
```



Now that we have the first part of the dataset, we can start to try to understand this. What are some possible parameters that go into suicide rates? Perhaps economic indicators or depression-related variables, such as the amount of sunlight in a year? In the coming section, we will test if the smaller amount of sunlight that one gets, the further away from the equator you go, will show any correlation with the suicide rate. However, as shown in the case of Guyana, some outliers might not fall into any general interpretation of any discovered trend.

Testing with linear regression

With linear regression, it is possible to test a proposed correlation between two variables. In the previous section, we ended up with a dataset of the suicide rates per country. We have a Pandas DataFrame with three columns: the country name, suicide rates for males and females, and mean of both genders. To test the hypothesis that suicide rate depends on how much sunlight the country gets, we are going to use the country coordinate centroid, that is, latitude (and longitude) for each country. We assume that the amount of sunlight in each country is directly proportional to the latitude. Getting centroids of each country in the world is more difficult than what one might imagine. Some of the resources for this are as follows:

- A simple countries centroid can be found on the Gothos web page: <http://gothos.info/2009/02/centroids-for-countries/>
- A more complex table, which would demand more processing before the analysis, is at OpenGeocode: <http://www.opengeocode.org/download.php#cow>
- OpenGeocode has geolocation databases that are free public domain

We are using the Gothos version, so you should make sure that you have the data file (CSV format):

```
coords=pandas.read_csv('data/country_centroids/  
country_centroids_primary.csv', sep='\t')  
coords.keys()
```

```
Index(['LAT', 'LONG', 'DMS_LAT', 'DMS_LONG', 'MGRS', 'JOG', 'DSG', 'AFFIL',  
       'FIPS10', 'SHORT_NAME', 'FULL_NAME', 'MOD_DATE', 'ISO3136'],  
      dtype='object')
```

A lot of column names to work with! First, we take a peek at the table:

```
coords.head()
```

	LAT	LONG	DMS_LAT	DMS_LONG	MGRS	JOG	DSG	AFFIL	FIPS10	SHORT_NAME	FULL_NAME	MOD_DATE	ISO3136
0	33.000000	66.0	330000	660000	42STB1970055286	NI42-09	PCLI	NaN	AF	Afghanistan	Islamic Republic of Afghanistan	2009-04-10	AF
1	41.000000	20.0	410000	200000	34TDL1589839239	NK34-08	PCLI	NaN	AL	Albania	Republic of Albania	2007-02-28	AL
2	28.000000	3.0	280000	30000	31REL0000097202	NH31-15	PCLI	NaN	AG	Algeria	People's Democratic Republic of Algeria	2011-03-03	DZ
3	-14.333333	-170.0	-142000	-1700000	1802701	NaN	PCLD	US	AS	American Samoa	Territory of American Samoa	1998-10-06	AS
4	42.500000	1.5	423000	13000	31TCH7675006383	NK31-04	PCLI	NaN	AN	Andorra	Principality of Andorra	2007-02-28	AD

The interesting columns are `SHORT_NAME` and `LAT`. We shall now match the `SHORT_NAME` in the `coords` DataFrame with `Country` in the `rates` DataFrame and store the `Lat` and `Lon` value when the country name matches. In theory, it would be best if the WHO tables also had the ISO country code, which is a standard created by the **International Organization for Standardization (ISO)**:

```
rates['Lat'] = ''
rates['Lon'] = ''
for i in coords.index:
    ind = rates.Country.isin([coords.SHORT_NAME[i]])
    val = coords.loc[i, ['LAT', 'LONG']].values.astype('float')
    rates.loc[ind, ['Lat', 'Lon']] = list(val)
```

Here, we loop over the index in the `coords` DataFrame,

and `rates.Country.isin([coords.SHORT_NAME[i]])` finds the country that we have taken from the `coords` object in the `rates` object. Thus, we are at the row of the country that we have found. We then take the `LAT` and `LONG` values found in the `coords` object and put it into the `rates` object in the `Lat` and `Lon` column. To check whether everything worked, we print out the first few rows:

```
rates.head()
```

	Country	Both	Female	Male	Lat	Lon
0	Afghanistan	5.7	5.3	6.2	33	66
1	Albania	5.9	5.2	6.6	41	20
2	Algeria	1.9	1.5	2.3	28	3
3	Angola	13.8	7.3	20.7	-12.5	18.5
4	Argentina	10.3	4.1	17.2	-34	-64

Some of the values are still empty, and Pandas, matplotlib, and many other modules do not handle empty values in a great way. So we find them and set the empty values to NaN (Not A Number). These are empty because we did not have the Country centroid or the names did not match:

```
rates.loc[rates.Lat.isin(['']), ['Lat']] = np.nan
rates.loc[rates.Lon.isin(['']), ['Lon']] = np.nan
rates[['Lat', 'Lon']] = rates[['Lat', 'Lon']].astype('float')
```

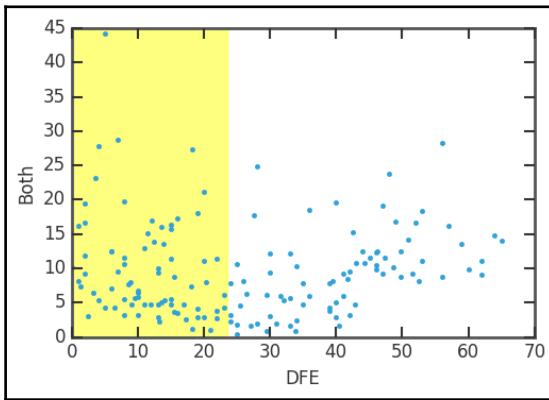
At the same time, we converted the values to floats (the last line of code) instead of strings; this makes it easier to plot and perform other routines. Otherwise, the routine has to convert to float and might run into problems that we have to fix. Converting it manually makes certain that we know what we have. In our simple approximation, the amount of sunlight is directly proportional to the distance from the equator, but we have Latitude. Therefore, we create a new column and calculate the **distance from equator (DFE)**, which is just the absolute value of the Latitude:

```
rates['DFE'] = ''
rates['DFE'] = abs(rates.Lat)
rates['DFE'] = rates['DFE'].astype('float')
```

Furthermore, countries within +/-23.5 degrees away from the equator get an equal amount of sunlight throughout the year, so they should be considered to have the same suicide rate according to our hypothesis. To first illustrate our new dataset, we plot the rates versus DFE:

```
import matplotlib.patches as patches
import matplotlib.transforms as transforms
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(rates.DFE, rates.Both, '.')
trans = transforms.blended_transform_factory(
```

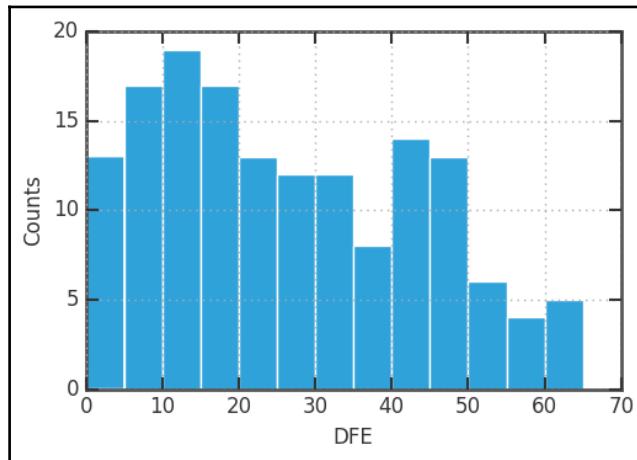
```
    ax.transData, ax.transAxes)
rect = patches.Rectangle((0, 0), width=23.5, height=1,
                       transform=trans, color='yellow', alpha=0.5)
ax.add_patch(rect)
ax.set_xlabel('DFE')
ax.set_ylabel('Both');
```



First, we plot the rates versus DFE, then we add a rectangle, for which we do a blended transform on the coordinates. With the blended transform, we can define the x coordinates to follow the data coordinates and the y coordinates to follow the axes (0 being the lower edge and 1 the upper edge). The region with DFE equal to or smaller than 23.5 degrees are marked with the yellow rectangle. It seems that there is some trend toward higher DFE and higher suicide rates. Although tragic and a sad outlook for people living further away from the equator, it lends support to our hypothesis.

To check whether we are sampling roughly even over the DFE, we draw a histogram over the DFE. A uniform coverage ensures that we run a lower risk of sample bias:

```
rates.DFE.hist(bins=13)
plt.xlabel('DFE')
plt.ylabel('Counts');
```



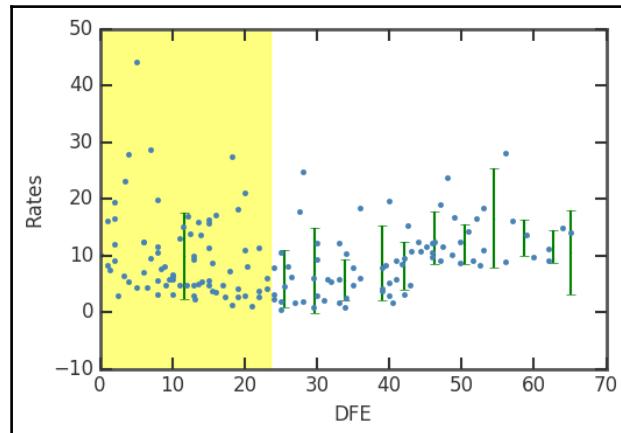
There are fewer countries with $\text{DFE} > 50$ degrees; however, there still seem to be enough for a comparison and regression. Eyeballing the values, it seems that we have about as many countries with $\text{DFE} > 50$ as one of the bins with $\text{DFE} < 50$. One possibility at this stage is to bin the data. Basically, this is taking the histogram bins and the mean of all the rates inside that bin, and letting the center of the bin represent the position and the mean, the value. To bin the data, we use the `groupby` method in Pandas together with the `digitize` function of NumPy:

```
bins = np.arange(23.5, 65+1, 10, dtype='float')
groups_rates = rates.groupby(np.digitize(rates.DFE, bins))
```

`Digitize` finds the indices for the input array of each bin. The `groupby` method then takes the list of indices, takes those positions from the input array, and puts them in a separate data group. Now we are ready to plot both the unbinned and binned data:

```
import matplotlib.patches as patches
import matplotlib.transforms as transforms
fig = plt.figure()
ax = fig.add_subplot(111)
ax.errorbar(groups_rates.mean().DFE,
            groups_rates.mean().Both,
            yerr=np.array(groups_rates.std().Both),
            marker='.',
            ls='None',
            lw=1.5,
            color='g',
            ms=1)
ax.plot(rates.DFE, rates.Both, '.', color='SteelBlue', ms=6)
```

```
trans = transforms.blended_transform_factory(
    ax.transData, ax.transAxes)
rect = patches.Rectangle((0,0), width=23.5, height=1,
    transform=trans, color='yellow', alpha=0.5)
ax.add_patch(rect)
ax.set_xlabel('DFE')
ax.set_ylabel('Both');
```



We now perform linear regression and test our hypothesis that less sunlight means a higher suicide rate. As before, we are using `linregress` from SciPy:

```
From scipy.stats import linregress
mindfe = 30
selection = ~rates.DFE.isnull() * rates.DFE>mindfe
rv = rates[selection].as_matrix(columns=['DFE','Both'])
a, b, r, p, stderr = linregress(rv.T)
print('slope:{0:.4f}\nintercept:{1:.4f}\nrvalue:{2:.4f}\npvalue:{3:.4f}\nstderr:{4:.4f}'.format(a, b, r, p, stderr))
```

```
slope:0.3204
intercept:-4.2373
rvalue:0.5102
pvalue:0.0000
stderr:0.0715
```

Here, the `mindfe` parameter was introduced only to fit a line where DFE is higher than this; you can experiment with the value. Logically, we will start this where DFE is 23.5; you will get slightly different results for different values. In our example, we use 30 degrees. If you want, you can plot the results just as in the previous chapter with the output of `linregress`.

As an alternative fitting method to `linregress`, we can use the powerful `statsmodels` package. It is installed by default in the Anaconda 3 Python distribution. The `statsmodels` package has a simple way of putting in the assumed relationship between the variables; it is the same as the R-style formulas and was included in `statsmodels` as of version 0.5.0. We want to test a linear relationship between `rates.DFE` and `rates.Both`, so we just tell this to `statsmodels` with `DFE ~ Both`. We are simply giving it the relationship between the keys/column names of the DataFrame.



The formula framework makes it very easy and clear to express the relationship that you want to fit. Except for relationships such as `Y ~ X + z`, it is also possible to add functions to the formula, such as `Y ~ X + np.log10(Z)`, to investigate more complex relationships.

The function is fitted with the **Ordinary Least Squares (OLS)** method, which basically minimizes the square of the difference between the fit and data (also known as the `loss` function):

```
import statsmodels.formula.api as smf
mod = smf.ols("DFE ~ Both", rates[selection]).fit()
print(mod.summary())
```

OLS Regression Results						
Dep. Variable:	Both	R-squared:	0.260			
Model:	OLS	Adj. R-squared:	0.247			
Method:	Least Squares	F-statistic:	20.06			
Date:	Mon, 21 Dec 2015	Prob (F-statistic):	3.65e-05			
Time:	03:40:19	Log-Likelihood:	-175.72			
No. Observations:	59	AIC:	355.4			
Df Residuals:	57	BIC:	359.6			
Df Model:	1					
Covariance Type:	I	nonrobust				
coef	std err	t	P> t	[95.0% Conf. Int.]		
Intercept	-4.2373	3.272	-1.295	0.201	-10.789	2.315
DFE	0.3204	0.072	4.479	0.000	0.177	0.464
Omnibus:	13.615	Durbin-Watson:	2.424			
Prob(Omnibus):	0.001	Jarque-Bera (JB):	14.566			
Skew:	1.099	Prob(JB):	0.000687			
Kurtosis:	4.047	Cond. No.	238.			
Warnings:						
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.						

The first part, that is, the left side of the top table gives general information. The dependent variable (**Dep. Variable**) states the name of the variable that is fitted. **Model** states what model we used in the fit; except OLS, there are several other models such as **weighted least squares (WLS)**. The number of observations (**No. Observations**) are listed and the degrees of freedom of the residuals (**Df Residuals**), that is, the number of observations (59) minus the parameters determined through the fitting 2 (k and k). **Df Model** shows how many parameters were determined (except the constant, that is, intercept). The table to the right of the top table shows you information on how well the model fits the data. R-squared was covered before; here, the adjusted R-square value (**Adj. R-squared**) is also listed and this is the R-square value corrected for the number of data points and degrees of freedom. The **F-statistic** number gives you an estimate of how significant the fit is. Practically, it is the mean squared error of the model divided by the mean squared error of the residuals. The next value, Prob (F-statistic), gives you the probability to get the F-statistic value if the null hypothesis is true, that is, the variables are not related. After this, three sets of log-likelihood function values follow: the value of the log-likelihood value of the fit, the **Akaike Information Criterion (AIC)**, and **Bayes Information Criterion (BIC)**. The AIC and BIC are various ways of adjusting the log-likelihood function for the number of observations and model type.

After this, there is a table of the determined parameters, where, for each parameter, the estimated value (`coeff`), standard error of the estimate (`std_err`), t-statistic value (`t`), P-value (`P>|t|`), and 95% confidence interval is shown. A P-value lower than a fixed confidence level, here 0.05 (that is, 5%), shows that there is a statistically significant relationship between the data and `model` parameter.

The last section shows the outcome of several statistical tests, which relates to the distribution of the fit residuals. Information about these can be found in the literature and the `statsmodels` documentation (<http://statsmodels.sourceforge.net/>). In general, the first few test for the shape (skewness) of the errors (residuals): Skewness, Kurtosis, Omnibus, and Jarque-Bera. The rest test if the errors are independent (that is, autocorrelation)—Durbin-Watson—or how the fitted parameters might be correlated with each other (for multiple regression)—**Conditional Number (Cond. No.)**.

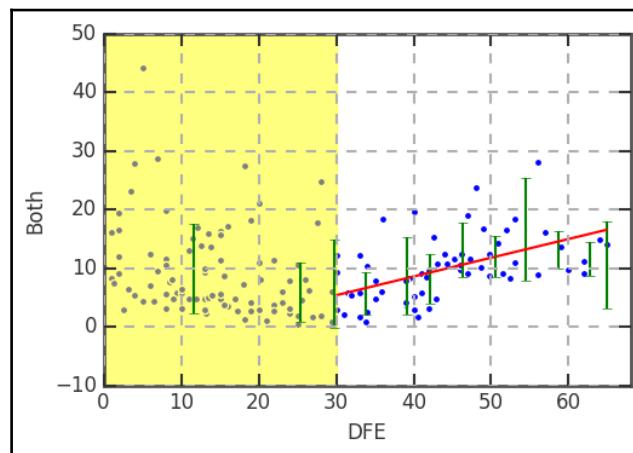
From this, we can now say that the suicide rate increases roughly by $0.32+/-0.07$ per increasing degree of absolute latitude above 30 degrees per 100,000 inhabitants. There is a weak correlation with absolute Latitude (DFE). The very low Prob (F-statistic) value shows that we can reject the null hypothesis that the two variables are unrelated with quite a high certainty, and the low $P>|t|$ value shows that there is a relationship between DFE and suicide rate.

We can now plot the fit together with the data. To get the uncertainty of the fit drawn in the figure as well, we use the built-in `wls_prediction_std` function that calculates the lower and upper bounds with 1 standard deviation uncertainty. The WLS here stands for Weighted Least Squares, a method like OLS, but where the uncertainty in the input variables is known and taken into account. It is a more general case of OLS; for the purpose of calculating the bounds of the uncertainty, it is the same:

```
from statsmodels.sandbox.regression.predstd import wls_prediction_std
prstd, iv_l, iv_u = wls_prediction_std(mod)
fig = plt.figure()
ax = fig.add_subplot(111)
rates.plot(kind='scatter', x='DFE', y='Both', ax=ax)
xmin, xmax = min(rates['DFE']), max(rates['DFE'])
ax.plot([mindfe, xmax],
        [mod.fittedvalues.min(), mod.fittedvalues.max()],
        'IndianRed', lw=1.5)
ax.plot([mindfe, xmax], [iv_u.min(), iv_u.max()], 'r--', lw=1.5)
ax.plot([mindfe, xmax], [iv_l.min(), iv_l.max()], 'r--', lw=1.5)
ax.errorbar(groups_rates.mean().DFE,
            groups_rates.mean().Both,
            yerr=np.array(groups_rates.std().Both),
            ls='None',
            lw=1.5,
```

```
    color='Green')
trans = transforms.blended_transform_factory(
    ax.transData, ax.transAxes)

rect = patches.Rectangle((0,0), width=mindfe, height=1,
                        transform=trans, color='Yellow',
                        alpha=0.5)
ax.add_patch(rect)
ax.grid(lw=1, ls='dashed')
ax.set_xlim((xmin,xmax+3));
```



There are quite a few studies showing the dependency of suicide rates and Latitude (for example, *Davis GE and Lowell WE, Can J Psychiatry. 2002 Aug; 47(6):572-4. Evidence that latitude is directly related to variation in suicide rates.*). However, some studies used fewer countries (20), so they could suffer from selection bias, that is, accidentally choosing those countries that favor a strong correlation. What's interesting from this data though is that there seems to be a minimum suicide rate at a higher latitude, which speaks in favor of some relationship.

A few things to remember here are as follows:

- There is a significant spread around the trend. This indicates that this is not one of the major causes for the spread in suicide rates over the world. However, it does show that it is one of the things that might affect it.
- In the beginning, we took the coordinate centroids; some countries span long ranges of latitude. Thus, the rates might vary within that country.

- Although we assume that a direct correlation with the amount of sunlight per year one receives is directly proportional to the latitude, weather of course also plays a role in how much sunlight we see and get.
- It is difficult interpreting the gender data; perhaps women try suicide just as much as men but fail more often, then getting proper help. This would bias the data so that we believe men are more suicidal.

In the long run, the amount of exposure to sunlight affects the production of vitamin D in the body. Many studies are trying to figure out how all of these factors affect the human body. One indication of the complex nature of this comes from studies that show seasonal variations in the suicide rates closer to the equator (Cantor, Hickey, and De Leo, *Psychopathology* 2000; 33:303-306), indicating that increased and sudden changes in sun exposure increase the suicide risk. The body's reaction to changes in sun exposure is to release or inhibit the release various hormones (melatonin, serotonin, L-tryptophan, among others); a sudden change in the levels of the hormones seems to increase the risk of suicide.

Another potential influence on suicide rates is economic indicators. Thus, we will now try to see if there is a correlation between these three variables through multivariate regression.

Multivariate regression

In this section, we are adding a second variable to the linear model constructed previously. The function that we use to express the correlation now basically becomes a plane, $y = k_2x_2 + k_1x_1 + k$. Just keep in mind that x_1 and x_2 are different axes/dimensions. To clarify a bit, we can also write it as $z = k_2y + k_1x + k$. Just as described in the beginning of the chapter, we can write this as a matrix multiplication. The variable that we choose to include is an economic variable, the **gross domestic product (GDP)**. As hypothesized before, the economy of the country could affect the suicide rate, as suicide prevention needs a developed medical system that isolates people in need and provides help, which is expensive.

Adding economic indicators

Luckily in this case, Pandas has a built-in remote data module that can be used to get certain indicators (http://pandas.pydata.org/pandas-docs/stable/remote_data.html). Currently, the services possible to query from Pandas directly are as follows:

- Yahoo! Finance
- Google Finance
- St.Louis FED (FRED)

- Kenneth French's data library
- World Bank
- Google Analytics

To query the World Bank for GDP per capita indicators, we simply do a search for it:

```
from pandas.io import wb
wb.search('gdp.*capita.*').iloc[:, :2]
```

	id	name
680	6.0.GDPpc_constant	GDP per capita, PPP (constant 2011 internation...
4611	GDPPCKD	GDP per Capita, constant US\$, millions
4612	GDPPCKN	Real GDP per Capita (real local currency units...
6390	NE.GDI.FTOT.CR	GDP expenditure on gross fixed capital formati...
6478	NV.AGR.PCAP.KD.ZG	Real agricultural GDP per capita growth rate (%)
6600	NY.GDP.PCAP.CD	GDP per capita (current US\$)
6601	NY.GDP.PCAP.CN	GDP per capita (current LCU)
6602	NY.GDP.PCAP.KD	GDP per capita (constant 2005 US\$)
6603	NY.GDP.PCAP.KD.ZG	GDP per capita growth (annual %)
6604	NY.GDP.PCAP.KN	GDP per capita (constant LCU)



In upcoming versions of Pandas, the `pandas.io.data` module will be a separate package called `pandas-datareader`. If you happen to read this when this update has occurred, please install `pandas-datareader` (`conda install pandas-datareader`) and replace the import from `pandas.io import wb` with `from pandas_datareader import wb`.

The indicator that we are looking for is the GDP per capita (current U.S. \$). Now we can download this dataset in a very easy way by asking for its ID, `NY.GDP.PCAP.PP.CD`, directly:

```
dat = wb.download(indicator='NY.GDP.PCAP.PP.CD', country='all', start=2014,
end=2014)
dat.head()
```

		NY.GDP.PCAP.PP.CD
country	year	
Arab World	2014	15975.039211
Caribbean small states	2014	15231.111124
Central Europe and the Baltics	2014	23884.797208
East Asia & Pacific (all income levels)	2014	14853.204148
East Asia & Pacific (developing only)	2014	11922.720831

The data structure is a bit complex, so we need to make it more easily accessible for us. We do this by getting the data out into arrays and creating a new Pandas DataFrame called *data*:

```
country = np.array(dat.index.tolist())[:,0]
gdp = np.array(np.array(dat['NY.GDP.PCAP.PP.CD']))
data = pd.DataFrame(data=np.array([country,gdp]).T, columns=['country',
'gdp'])
print(dat['NY.GDP.PCAP.PP.CD'].head())
print(data.head())
```

```
country                               year
Arab World                           2014   15975.039211
Caribbean small states               2014   15231.111124
Central Europe and the Baltics      2014   23884.797208
East Asia & Pacific (all income levels) 2014   14853.204148
East Asia & Pacific (developing only) 2014   11922.720831
Name: NY.GDP.PCAP.PP.CD, dtype: float64
                                                country          gdp
0                      Arab World  15975.039210528601
1            Caribbean small states  15231.111124481498
2        Central Europe and the Baltics  23884.797208032996
3  East Asia & Pacific (all income levels)  14853.204148331899
4    East Asia & Pacific (developing only)  11922.7208309251
```

The data is now much more accessible for us and in the same format as the previous dataset. Just as with the coordinate centroid, we need to match the country names and put the relevant data into the rates object. We do this in exactly the same way as before:

```
rates['GDP_CD'] = ''
for i in np.arange(len(data)):
    ind = rates.Country.isin([data.country[i]]
    val = data.loc[i, ['gdp']].values.astype('float')
    rates.loc[ind], ['GDP_CD']] = val
    rates.loc[rates.GDP_CD.isin([''])], ['GDP_CD']] = np.nan
```

To check whether everything worked, we can print one of the items. In this example, we look at Sweden:

```
print(rates[rates.Country=='Sweden'])
print(data[data.country=='Sweden'])
print(data.loc[218, ['gdp']].values.astype('float'))
rates.loc[rates.Country.isin(['Sweden'])]
```

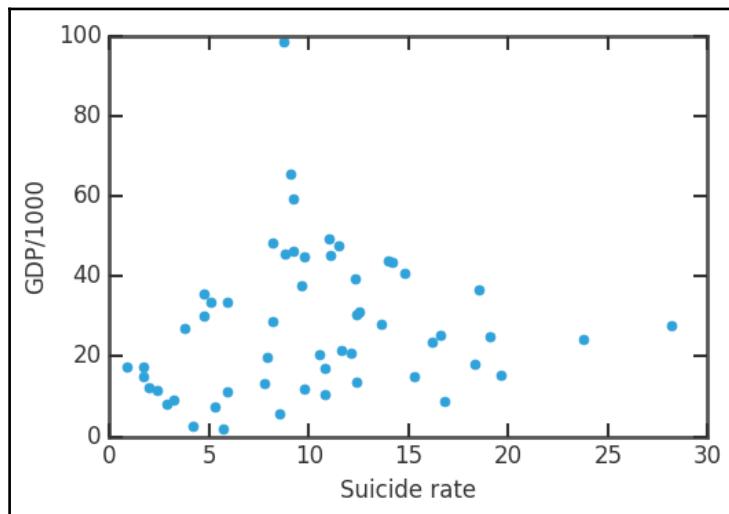
	Country	Both	Female	Male	Lat	Lon	DFE	GDP_CD
146	Sweden	11.1	6.1	16.2	62	15	62	45183
	country	gdp						
218	Sweden	45183.0196872713						
	[45183.01968727]							
	Country	Both	Female	Male	Lat	Lon	DFE	GDP_CD
146	Sweden	11.1	6.1	16.2	62	15	62	45183

This looks like we got it right. When we were working with **DFE**, we defined a `mindfe` variable. We use that here to select only those countries that satisfy being above the minimum latitude that we set previously and should have values for DFE as well. Then we add the rows where we have GDP to the selection:

```
selection = ~rates.DFE.isnull() * rates.DFE>mindfe
selection *= ~rates.GDP_CD.isnull()
```

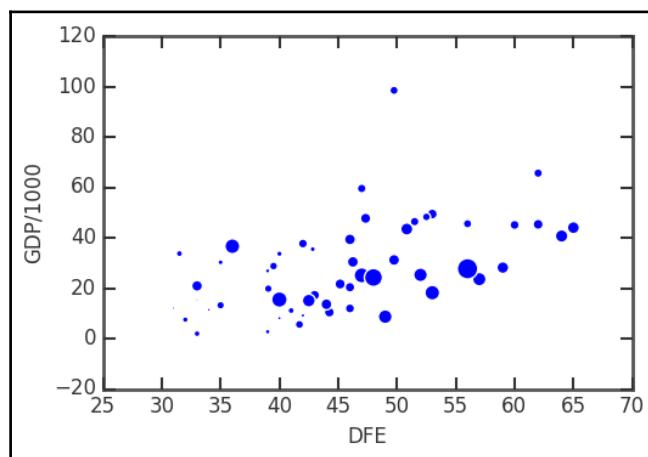
Check first whether there is any obvious correlation between the GDP and suicide rate:

```
plt.plot(rates[selection].GDP_CD.values, rates[selection].Both.values, '.', ms=10)
plt.xlabel('GDP')
plt.ylabel('Suicide rate');
```



It seems like a very broad relationship. We add the DFE variable and plot the size of the markers as the suicide rate:

```
plt.scatter(rates[selection].GDP_CD.values/1000,  
           rates[selection].DFE.values,  
           s=rates[selection].Both.values**1.5)  
plt.xlabel('GDP/1000')  
plt.ylabel('DFE')
```



It seems that high GDP countries have high DFE but also high suicide rates. Now, of course, we want to fit a linear model to this. Basically, the model will be a plane fitted to the points. To do this, we can use `statsmodels` again, and for the plotting, we import `matplotlib` 3D axes:

```
import statsmodels.api as sm

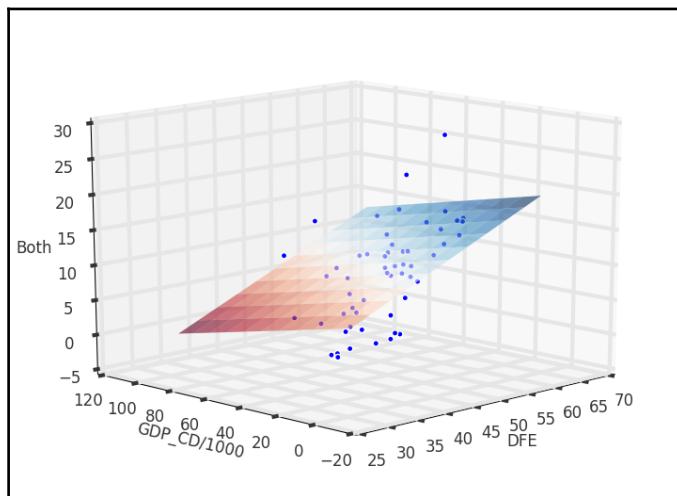
A = rates[selection][['DFE', 'GDP_CD']].astype('float')
A['GDP_CD'] = A['GDP_CD']/1000
b = rates[selection]['Both'].astype('float')
A = sm.add_constant(A)
est = sm.OLS(b, A).fit()
```

First, we select the data that we need for the fit, `DFE` and `GDP_CD` for the `A` matrix. Then, we run the fitting assuming that the suicide rates are dependent on `A` (that is, both `DFE` and `GDP`). For it to work, we have to add a column with a constant value (1); the `statsmodels` developers have provided such a function that we can use. Note that we show how to use another way of defining a fitting function in `statsmodels`, which is not the R-formula method but uses NumPy arrays instead. The linear function being fitted is exactly what we covered in the beginning of this section, which is the following relationship:

$$z = k_0 + k_1x + k_2y$$

Here, we express the relationship between the variables with a matrix multiplication (that is, k_0 , k_1 , k_2 are found by the fitting routine). Note the different import here; the method is called **OLS** (capital letters) instead of **ols**, which was used with the formula in the previous example. We can now plot the fit together with the data:

```
from mpl_toolkits.mplot3d import Axes3D
X, Y = np.meshgrid(np.linspace(A.DFE.min(), A.DFE.max(), 100),
np.linspace(A.GDP_CD.min(), A.GDP_CD.max(), 100))
Z = est.params[0] + est.params[1] * X + est.params[2] * Y
fig = plt.figure(figsize=(12, 8))
ax = Axes3D(fig, azim=-135, elev=15)
surf = ax.plot_surface(X, Y, Z, cmap=plt.cm.RdBu, alpha=0.6,
linewidth=0)
ax.scatter(A.DFE, A.GDP_CD, y, alpha=1.0)
ax.set_xlabel('DFE')
ax.set_ylabel('GDP_CD/1000')
ax.set_zlabel('Both');
```



The viewing direction can be controlled with the `azim` and `elev` keywords during the `Axes3D` object creation. To plot the plane, we use the `meshgrid` function from NumPy to create a coordinate grid and then use the parameters determined from the fitting routine to get the values of the plane. Once again, to get the fitting summary, we print it:

```
print(est.summary())
```

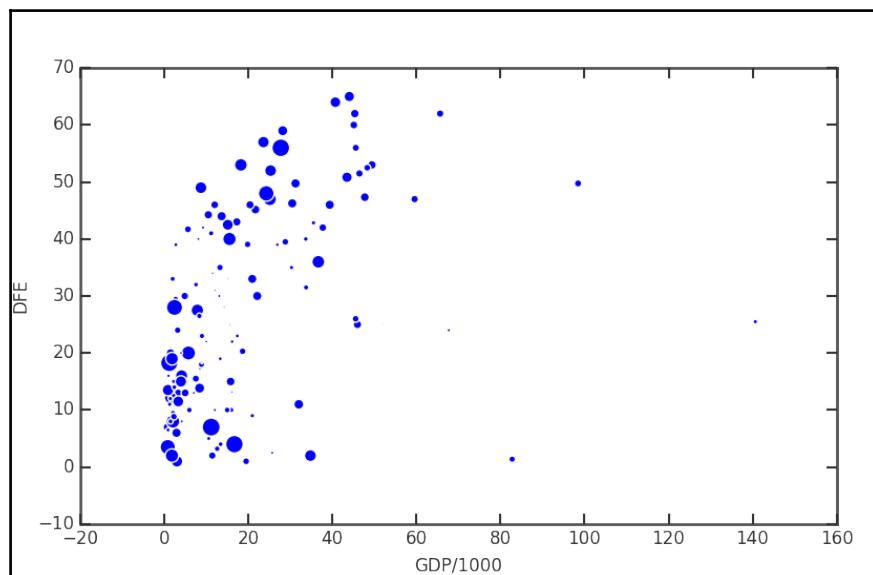
OLS Regression Results						
Dep. Variable:	Both	R-squared:	0.288			
Model:	OLS	Adj. R-squared:	0.260			
Method:	Least Squares	F-statistic:	10.33			
Date:	Mon, 21 Dec 2015	Prob (F-statistic):	0.000171			
Time:	03:40:42	Log-Likelihood:	-161.90			
No. Observations:	54	AIC:	329.8			
Df Residuals:	51	BIC:	335.8			
Df Model:	2					
Covariance Type:	nonrobust					
coef	std err	t	P> t	[95.0% Conf. Int.]		
const	-5.9298	3.626	-1.635	0.108	-13.210	1.350
DFE	0.3942	0.090	4.397	0.000	0.214	0.574
GDP_CD	-6.238e-05	4.55e-05	-1.371	0.176	-0.000	2.9e-05
Omnibus:	11.836	Durbin-Watson:	2.272			
Prob(Omnibus):	0.003	Jarque-Bera (JB):	11.958			
Skew:	1.062	Prob(JB):	0.00253			
Kurtosis:	3.899	Cond. No.	1.72e+05			
Warnings:						
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.						
[2] The condition number is large, 1.72e+05. This might indicate that there are strong multicollinearity or other numerical problems.						

The resulting r-squared is similar to the one obtained when only using DFE. The hypothesis is not completely wrong; however, the situation is much more complex than these two variables. This is reflected in the fitting results. The coefficient for the GDP is small, very close to zero (-6.238×10^{-5}), that is, the dependence on GDP is less pronounced than for Latitude. What about the $P > |t|$ values—what conclusion can you draw from it? Furthermore, there is a warning about multicollinearity because of the high conditional number. Thus, some of the fitted variables may be interdependent.

Taking a step back

Remember that we cut out everything with a low value of DFE; let's check what the full dataset looks like. In this plot, you also see that we use another magic command (starts with %); instead of the matplotlib inline, we use the notebook plotting interface. This gives you interactive controls to pan, zoom, and save the data directly in the Jupyter Notebook:

```
%matplotlib notebook
selection2 = ~rates.DFE.isnull()
plt.scatter(rates[selection2].GDP_CD.values/1000,
            rates[selection2].DFE.values, s=rates[selection2].Both.values**1.5)
plt.xlabel('GDP/1000')
plt.ylabel('DFE');
```



Introducing the size of the markers depending on the suicide rates, we can see that there seems to be at least two main clusters in the data with higher suicide rates—one with low GDP and absolute Latitude and one with high. We will use this in the next chapter to continue our analysis and attempt to identify the clusters. Thus, we should save this data, but we first create a new DataFrame with only the columns needed:

```
data=pd.DataFrame(data=rates[['Country','Both','Male','Female','GDP_CD',
'DFE']] [~rates.DFE.isnull()])
data.head()
```

	Country	Both	Male	Female	GDP_CD	DFE
0	Afghanistan	5.7	6.2	5.3	1932.89	33.0
1	Albania	5.9	6.6	5.2	10304.7	41.0
2	Algeria	1.9	2.3	1.5	14193.4	28.0
3	Angola	13.8	20.7	7.3	NaN	12.5
4	Argentina	10.3	17.2	4.1	NaN	34.0

We only include data where we have DFE; some rows will still lack GDP though. Now that we have created a new DataFrame, it is very easy to save it. This time, we save it in a more standardized format, the HDF format:

```
TABLE_FILE = 'data_ch4.h5'
data.to_hdf(TABLE_FILE, 'ch4data', mode='w', table=True)
```

This code will save the data to the `data_ch4.h5` file in the current directory.



HDF stands for **Hierarchical Data Format**; it is a scientific data format developed by the **National Center for Supercomputing Applications** (NCSA), specifically for large datasets. It is very fast at reading and writing large data from/to disk. Most major programming languages have libraries to interact with HDF files. The current legacy format version is HDF version 5.

To read the HDF data, we simply use the `read_hdf` function of Pandas:

```
d2 = pd.read_hdf(TABLE_FILE)
```

You can now run `d2.head()` for a sanity check, whether it is the same as what we wrote to the file. Remember to have this file handy for the next chapter.

Logistic regression

The examples so far have been of continuous variables. However, other variables are discrete and can be of a binary type. Some common examples of discrete binary variables are if it is snowing in a city on a given day or not, if a patient is carrying a virus or not, and so on. One of the main differences between binary logistic and linear regression is that in binary logistic regression, we are fitting the probability of an outcome given a measured (discrete or continuous) variable, while linear regression models deal with characterizing the dependency of two or more continuous variables on each other. Logistic regression gives the probability of an occurrence given some observed variable(s). Probability is sometimes expressed as $P(Y|X)$ and read as *Probability that the value is Y given the variable X*.

Algorithms that guess the discrete outcome are called classification algorithms and are a part of machine learning techniques, which will be covered later in the book.

The logistic regression model can be expressed as follows:

$$\ln\left(\frac{P}{1-P}\right) = m + kx$$

Solving this equation for P , we get the logistic probability:

$$\frac{P}{1-P} = e^{m+kx}$$

$$P = \frac{1}{1+e^{-(m+kx)}}$$

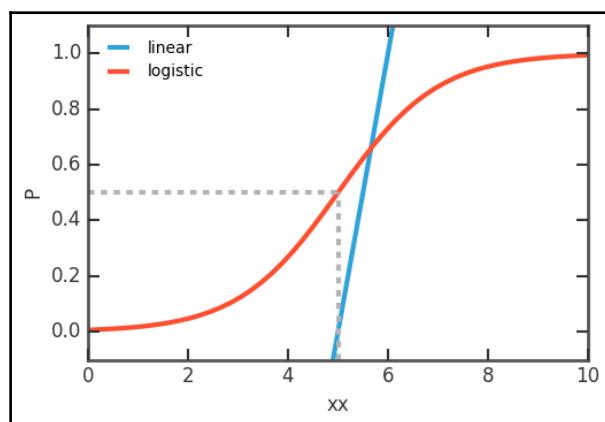
$$P = \frac{1}{1+e^{-(m+kx)}}$$

We can, just as with linear regression, add several dimensions (dependent variables) to the problem:

$$P = \frac{1}{1 + e^{-(m + k_1x_1 + k_2x_2 + \dots + k_{N-1}x_{N-1}k_Nx_N)}}$$

To illustrate what this function looks like and the difference from fitting a linear model, we plot both functions:

```
k = 1.  
m = -5.  
y = lambda x: k*x + m  
#p = lambda x: np.exp(k*x+m) / (1+np.exp(k*x+m))  
p = lambda x: 1 / (1+np.exp(-1*(k*x+m)))  
  
xx = np.linspace(0,10)  
plt.plot(xx,y(xx), label='linear')  
plt.plot(xx,p(xx), label='logistic')  
plt.plot([0,abs(m)], [0.5,0.5], dashes=(4,4), color='.7')  
plt.plot([abs(m),abs(m)], [-.1,.5], dashes=(4,4), color='.7')  
  
# limits, legends and labels  
plt.ylim((-1,1.1))  
plt.legend(loc=2)  
plt.ylabel('P')  
plt.xlabel('xx')
```



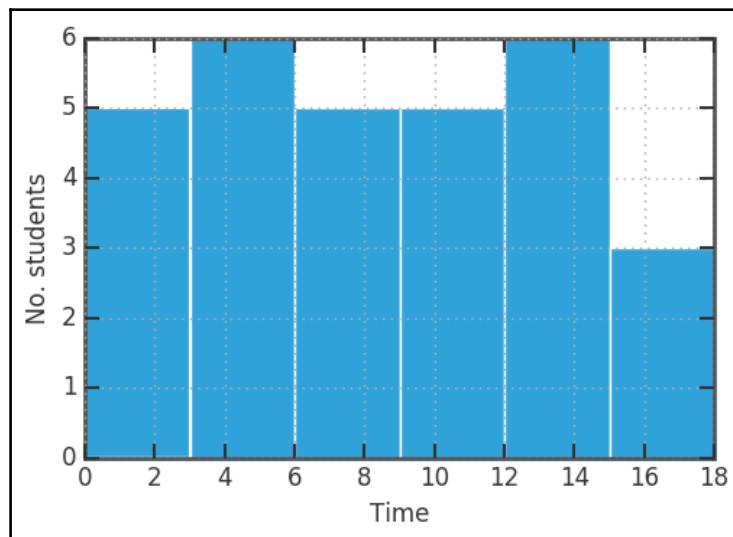
As is clearly seen, the S-shaped curve, our logistic fitting function (more generally called **sigmoid function**), can illustrate binary logistic probabilities much better. Playing around with k and m , we quickly realize that k determines the steepness of the slope and m moves the curve left or right. We also notice that $P(Y|xx=5) = 0.5$, that is, at $xx=5$, the outcome Y (corresponding to $P=1$) has a 50% probability.

Imagine that we asked students how long they studied for an exam. Can we check how long you need to study to be fairly sure to pass? To investigate this, we need to use logistic regression. It is only possible to pass or fail an exam, that is, it is a binary variable. First, we need to create the data:

```

studytime=[0,0,1.5,2,2.5,3,3.5,4,4,4,5.5,6,6.5,7,7,8.5,9,9,9,10.5,10.5,12,12,12.5,13,14,15,16,18]
passed=[0,0,0,0,0,0,0,0,0,1,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
data = pd.DataFrame(data=np.array([studytime, passed]).T, columns=['Time', 'Pass'])
data.Time.hist(bins=6)
plt.xlabel('Time')
plt.ylabel('No. students');

```

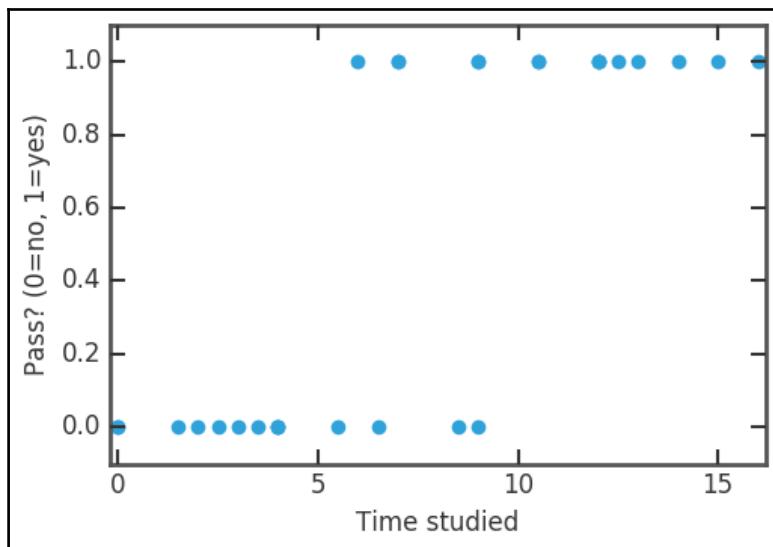


The first thing plotted is the histogram of how much time the students spent studying for the exam. It seems like it is a rather flat distribution. Here, we will check how they did on the exam:

```
plt.plot(data.Time, data.Pass, 'o', mew=0, ms=7,)  
plt.ylim(-.1,1.1)
```

```
plt.xlim(-0.2,16.2)
plt.xlabel('Time studied')
plt.ylabel('Pass? (0=no, 1=yes)');
```

This plot will now show you how much time someone studied and the outcome of the exam-if they passed given by a value of 1.0 (yes) or if they failed given by a value of 0.0 (no). The x axis goes from 0 hours (a student who did not study at all) up to 18 hours (a student who studied more).



By simply inspecting the figure, it seems like sometime between 5-10 hours is needed to at least pass the exam. Once again, we use statsmodels to fit the data with our model. In statsmodels, there is a `logit` function that performs logistic regression:

```
import statsmodels.api as sm
probfit = sm.Logit(data.Pass, sm.add_constant(data.Time, prepend=True))
```

```
Optimization terminated successfully.
Current function value: 0.251107
Iterations 8
```

The optimization worked; if there would have been any problems converging, error messages would have been printed instead. After running the fit, checking the summary is a good idea:

```
fit_results = probfit.fit()  
print(fit_results.summary())
```

Logit Regression Results						
Dep. Variable:	Pass	No. Observations:	30			
Model:	Logit	Df Residuals:	28			
Method:	MLE	Df Model:	1			
Date:	Mon, 21 Dec 2015	Pseudo R-squ.:	0.6366			
Time:	03:40:43	Log-Likelihood:	-7.5332			
converged:	True	LL-Null:	-20.728			
		LLR p-value:	2.791e-07			
coef	std err	z	P> z	[95.0% Conf. Int.]		
const	-5.7980	2.240	-2.588	0.010	-10.188	-1.408
Time	0.8020	0.297	2.703	0.007	0.220	1.384

The `const` variable is the intercept, that is, k of the fit-function, and `Time` is the intercept, m . The covariance parameters can be used to estimate the standard deviation by taking the square root of the diagonal of the covariance matrix:

```
logit_pars = fit_results.params  
intercept_err, slope_err = np.diag(fit_results.cov_params() )**.5  
fit_results.cov_params()
```

	const	Time
const	5.017663	-0.635081
Time	-0.635081	0.088035

However, statsmodels also gives the uncertainty in the parameters, as can be seen from the fit summary output:

```
intercept = logit_pars['const']
slope = logit_pars['Time']
print(intercept,slope)
-5.79798670884 0.801979232718
```

-5.79798670884 0.801979232718

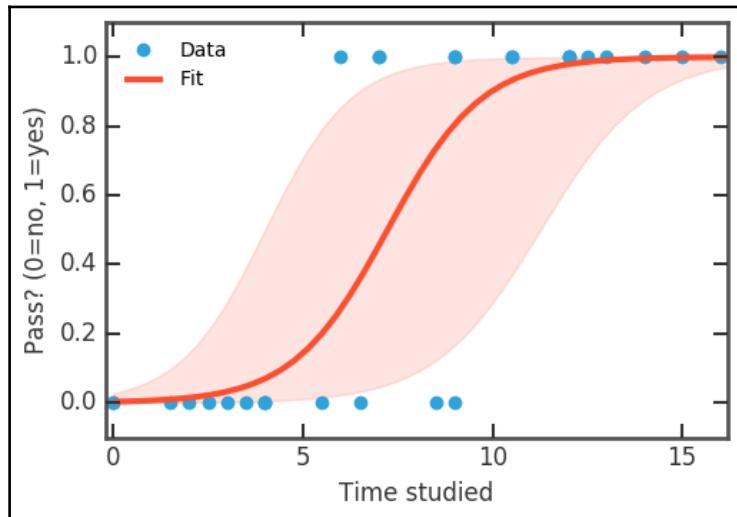
It is also possible to print out the confidence intervals directly:

```
fit_results.conf_int()
```

	0	1
const	-10.188333	-1.407640
Time	0.220444	1.383514

Now it is appropriate to plot the fit on top of the data. We have estimated the parameters of the fit function:

```
plt.plot(data.Time, data.Pass, 'o', mew=0, ms=7, label='Data')
p = lambda x,k,m: 1 / (1+np.exp(-1*(k*x+m)))
xx = np.linspace(0,data.Time.max())
l1 = plt.plot(xx, p(xx,slope,intercept), label='Fit')
plt.fill_between(xx, p(xx,slope+slope_err**2, intercept+intercept_err),
p(xx,slope-slope_err**2, intercept-intercept_err), alpha=0.15,
color=l1[0].get_color())
plt.ylim(-.1,1.1)
plt.xlim(-0.2,16.2)
plt.xlabel('Time studied')
plt.ylabel('Pass? (0=no, 1=yes)')
plt.legend(loc=2, numpoints=1);
```



Here, we have not only plotted the best fit curve, but also the curve corresponding to one standard deviation away. The uncertainty encompasses a lot of values. Now, with the estimated parameters, it is possible to calculate how long should we study for a 50% chance of success:

```
target=0.5
x_prob = lambda p,k,m: (np.log(p/(1-p))-m)/k
T_max = x_prob(target, slope-slope_err, intercept-intercept_err)
T_min = x_prob(target, slope+slope_err, intercept+intercept_err)
T_best = x_prob(target, slope, intercept)
print('{0}% sucess rate: {1:.1f} +{2:.1f}/-
{3:.1f}'.format(int(target*100),T_best,T_max-T_best,T_best-T_min))
50% success rate: 7.2 +8.7/-4.0
```

So studying for 7.2 hours for this test, the chance of passing is about 50%. The uncertainty is rather large, and the 50% chance of passing could also be for about 15 hours of studying or as little as three hours. Of course, there is more to studying for an exam than the absolute number of hours put in. However, by not studying at all, the chances of passing are very slim.

Some notes

Logistic regression assumes that the probability at the inflection point, that is, halfway through the S-curve, is 0.5. There is no real reason to assume that this is always true; thus, using a model that allows the inflection point to move could be a more general case. However, this will add another parameter to estimate and, given the quality of the input data, this might not make it easier or increase the reliability.

Summary

In this chapter, we looked at linear, multiple, and logistic regression. We fetched data from online sources, cleaned it up, and mapped it to the data structures that we are interested in. The world of statistics is huge and there are numerous special areas even for these somewhat straightforward concepts and methods. For regression analysis, it is important to note that correlation does not always mean causation, that is, just because there is a correlation between two variables, it does not mean that they depend on one another in nature. There are websites that show these spurious correlations; some of them are quite entertaining (<http://www.tylervigen.com/spurious-correlations>).

In the next chapter, we will look at clustering techniques to find similarities in data. We will start out with an example using the same data that we saved in this chapter when performing multiple regression analysis.

5

Clustering

With data comprising of several separated distributions, how do we find and characterize them? In this chapter, we will look at some ways to identify clusters in data. Groups of points with similar characteristics form clusters. There are many different algorithms and methods to achieve this with good and bad points. We want to detect multiple separate distributions in the data and determine the degree of association (or similarity) with another point or cluster for each point. The degree of association needs to be high if they belong in a cluster together or low if they do not. This can of course, just as previously, be a one-dimensional problem or multi-dimensional problem. One of the inherent difficulties of cluster finding is determining how many clusters there are in the data. Various approaches to define this exist; some where the user needs to input the number of clusters and then the algorithm finds which points belong to which cluster, and some where the starting assumption is that every point is a cluster and then two nearby clusters are combined iteratively on a trial basis to see if they belong together.

In this chapter, we will cover the following topics:

- A short introduction to cluster finding—reminding you of the general problem—and an algorithm to solve it
- Analysis of a dataset in the context of cluster finding—the Cholera outbreak in central London 1854
 - By simple zeroth order analysis, calculating the centroid of the whole dataset
 - By finding the closest water pump for each recorded Cholera-related death
- Using the K-means nearest neighbor algorithm for cluster finding, applying it to data from Chapter 4, *Regression*, and identifying two separate distributions
- Using hierarchical clustering by analyzing the distribution of galaxies in a confined slice of the Universe

The algorithms and methods covered here are focused on those available in SciPy.

As before, start a new Notebook and put in the default imports. Perhaps you want to change to interactive Notebook plotting to try it out a bit more. For this chapter, we are adding the following specific imports. The ones related to clustering are from SciPy, while later on, we will need some packages to transform astronomical coordinates. These packages are all preinstalled in the Anaconda Python 3 distribution and have been tested there:

```
import scipy.cluster.hierarchy as hac  
import scipy.cluster.vq as vq
```

Introduction to cluster finding

There are many different algorithms for cluster identification. Many of them try to solve a specific problem in the best way. Therefore, the specific algorithm that you want to use might depend on the problem you are trying to solve and also on what algorithms are available in the specific package that you are using.

Some of the first clustering algorithms consisted of simply finding the centroid positions that minimize the distances to all the points in each cluster. The points in each cluster are closer to that centroid than other cluster centroids. As might be obvious at this point, the hardest part with this is figuring out how many clusters there are. If we can determine that, it is fairly straightforward to try various ways of moving the cluster centroid around, calculate the distance to each point, and then figure out where the cluster centroids are. There are also obvious situations where this might not be the best solution, for example, if you have two very elongated clusters next to each other.

Commonly, the distance is the Euclidean distance:

$$\boxed{\bar{p} - \mu_i}$$

Here, p is a vector with all the points' positions, that is, $\{p_1, p_2, \dots, p_{N-1}, p_N\}$ in cluster C_k , and the distances are calculated from the cluster centroid, μ_i . We have to find the cluster centroids that minimize the sum of the absolute distances to the points:

$$\min \sum_{i=1}^K | \bar{p} - \mu_i |$$

In this first example, we shall first work with fixed cluster centroids.

Starting out simple – John Snow on cholera

In 1854, there was an outbreak of cholera in North-western London, in the neighborhood around Broad Street. The leading theories at the time claimed that cholera spread, just like it was believed the plague spread, through foul, bad air. John Snow, a physician at the time, hypothesized that cholera spread through drinking water. During the outbreak, John tracked the deaths and drew them on a map of the area. Through his analysis, he concluded that most of the cases were centered on the Broad Street water pump. Rumors say that he then removed the handle of the water pump, thus stopping an epidemic. Today, we know that cholera is usually transmitted through contaminated food or water, thus confirming John's hypothesis. We will do a short but instructive reanalysis of John Snow's data.

The data comes from the public data archives of The National Center for Geographic Information and Analysis (<http://www.ncgia.ucsb.edu/> and <http://www.ncgia.ucs.edu/pubs/data.php>). A cleaned-up map and copy of the data files along with an example of a geospatial information analysis of the data can also be found at <https://www.udel.edu/johnmack/frec682/cholera/cholera2.html>. A wealth of information about physician and scientist John Snow's life and works can be found at <http://johnsnow.matrix.msu.edu>.

To start the analysis, we read in the data to a Pandas DataFrame; the data is already formatted into CSV files, readable by Pandas:

```
deaths = pd.read_csv('data/cholera_deaths.txt')
pumps = pd.read_csv('data/cholera_pumps.txt')
```

Each file contains two columns, one for X coordinates and one for Y coordinates. Let's check what it looks like:

```
deaths.head()
```

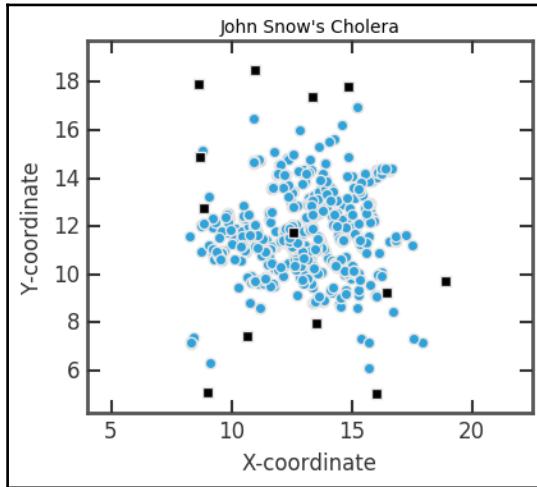
	X	Y
0	13.588010	11.095600
1	9.878124	12.559180
2	14.653980	10.180440
3	15.220570	9.993003
4	13.162650	12.963190

```
pumps.head()
```

	X	Y
0	8.651201	17.891600
1	10.984780	18.517851
2	13.378190	17.394541
3	14.879830	17.809919
4	8.694768	14.905470

With this information, we can now plot all the pumps and deaths in order to visualize the data:

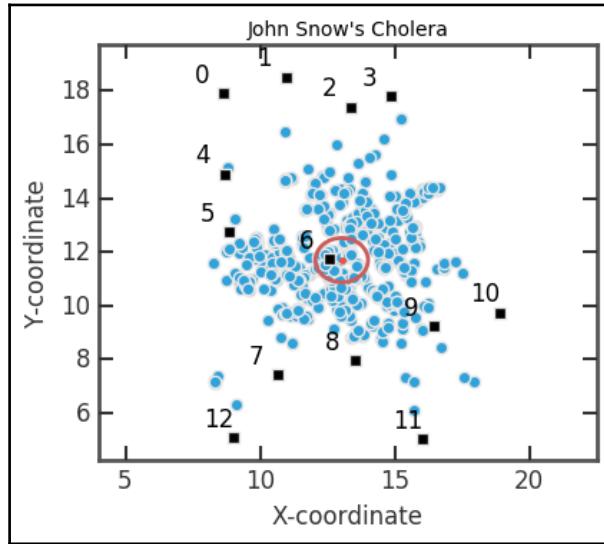
```
plt.figure(figsize=(4, 3.5))
plt.plot(deaths['X'], deaths['Y'],
          marker='o', lw=0, mew=1, mec='0.9', ms=6)
plt.plot(pumps['X'],pumps['Y'],
          marker='s', lw=0, mew=1, mec='0.9', color='k', ms=6)
plt.axis('equal')
plt.xlim((4.0,22.0));
plt.xlabel('X-coordinate')
plt.ylabel('Y-coordinate')
plt.title('John Snow's Cholera')
```



It is fairly easy to see that the pump in the middle is important. As a first data exploration, we will simply calculate the mean centroid of the distribution and plot this in the figure as an ellipse. We calculate the mean and standard deviation along the x and y axes as the centroid position:

```
fig = plt.figure(figsize=(4,3.5))
ax = fig.add_subplot(111)
plt.plot(deaths['X'], deaths['Y'],
          marker='o', lw=0, mew=1, mec='0.9', ms=6)
plt.plot(pumps['X'],pumps['Y'],
          marker='s', lw=0, mew=1, mec='0.9', color='k', ms=6)

from matplotlib.patches import Ellipse
ellipse = Ellipse(xy=(deaths['X'].mean(), deaths['Y'].mean()),
                  width=deaths['X'].std(), height=deaths['Y'].std(),
                  zorder=32, fc='None', ec='IndianRed', lw=2)
ax.add_artist(ellipse)
plt.plot(deaths['X'].mean(), deaths['Y'].mean(),
         '.', ms=10, mec='IndianRed', zorder=32)
for i in pumps.index:
    plt.annotate(s='{0}'.format(i), xy=(pumps[['X','Y']].loc[i]),
                 xytext=(-15,6), textcoords='offset points')
plt.axis('equal')
plt.xlim((4.0,22.5))
plt.xlabel('X-coordinate')
plt.ylabel('Y-coordinate')
plt.title('John Snow's Cholera')
```



Here, we also plotted the pump index, which we can get from DataFrame with the `pumps.index` method. The next step in the analysis is to see which pump is the closest to each point. We do this by calculating the distance from all pumps to all points. Then, we want to figure out for each point, which pump is the closest.

We save the closest pump to each point in a separate column of the deaths' DataFrame. With this dataset, the for-loop runs fairly quickly. However, the DataFrame `subtract` method chained with `sum()` and `idxmin()` methods takes a few seconds. I strongly encourage you to play around with various ways to speed this up. We also use the `.apply()` method of DataFrame to square and square root the values. The simple brute force first attempt of this took over a minute to run. The built-in functions and methods helped a lot:

```
deaths_tmp = deaths[['X', 'Y']].as_matrix()
idx_arr = np.array([], dtype='int')
for i in range(len(deaths)):
    idx_arr = np.append(idx_arr,
        (pumps.subtract(deaths_tmp[i])).apply(lambda
            x:x**2).sum(axis=1).apply(lambda x:x**0.5).idxmin())
deaths['C'] = idx_arr
```

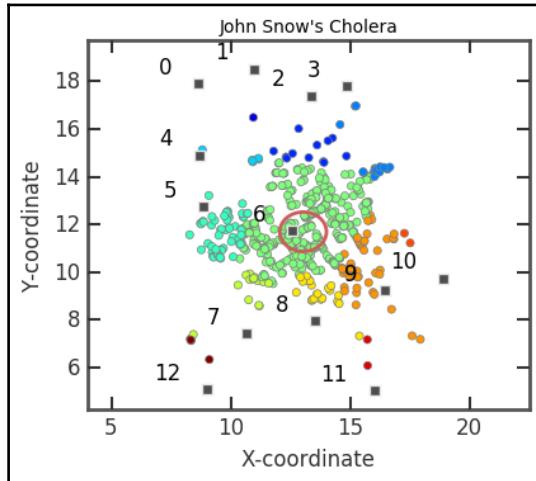
Just quickly check whether everything seems fine by printing out the top rows of the table:

```
deaths.head()
```

	X	Y	C
0	13.588010	11.095600	6
1	9.878124	12.559180	5
2	14.653980	10.180440	9
3	15.220570	9.993003	9
4	13.162650	12.963190	6

Now we want to visualize what we have. With colors, we can show which water pump we associate each death with. To do this, we use a colormap; in this case, the *jet* colormap. By calling the colormap with a value between 0 and 1, it returns a color; thus, we give it the pump indexes and then divide it with the total number of pumps – 12 in our case:

```
fig = plt.figure(figsize=(4,3.5))
ax = fig.add_subplot(111)
np.unique(deaths['C'].values)
plt.scatter(deaths['X'].as_matrix(), deaths['Y'].as_matrix(),
            color=plt.cm.jet(deaths['C']/12.),
            marker='o', lw=0.5, edgecolors='0.5', s=20)
plt.plot(pumps['X'],pumps['Y'],
marker='s', lw=0, mew=1, mec='0.9', color='0.3', ms=6)
for i in pumps.index:
    plt.annotate(s='{0}'.format(i), xy=(pumps[['X','Y']].loc[i]),
                 xytext=(-15,6), textcoords='offset points',
                 ha='right')
ellipse = Ellipse(xy=(deaths['X'].mean(), deaths['Y'].mean()),
                   width=deaths['X'].std(),
                   height=deaths['Y'].std(),
                   zorder=32, fc='None', ec='IndianRed', lw=2)
ax.add_artist(ellipse)
plt.axis('equal')
plt.xlim((4.0,22.5))
plt.xlabel('X-coordinate')
plt.ylabel('Y-coordinate')
plt.title('John Snow's Cholera')
```



The majority of deaths are dominated by the proximity of the pump in the center. This pump is located on Broad Street.

Now, remember that we have used fixed positions for the cluster centroids. In this case, we are basically working on the assumption that the water pumps are related to the cholera cases. Furthermore, the Euclidean distance is not really the real-life distance. People go along roads to get water and the road there is not necessarily straight. Thus, one would have to map out the streets and calculate the distance to each pump from that. Even so, already at this level, it is clear that there is something with the center pump related to the cholera cases. How would you account for the different distance? To calculate the distance, you would do what is called cost-analysis (c.f. when you hit directions on your sat-nav to go to a place). There are many different ways of doing cost analysis, and it also relates to the problem of finding the correct way through a maze.

In addition to these things, we do not have any data in the time domain, that is, the cholera would possibly spread to other pumps with time and the outbreak might have started at the Broad Street pump and spread to other, nearby pumps. Without time data, it is extra difficult to figure out what happened.

This is the general approach to cluster finding. The coordinates might be attributes instead, length and weight of dogs for example, and the location of the cluster centroid something that we would iteratively move around until we find the best position.

K-means clustering

The K-means algorithm is also referred to as vector quantization. What the algorithm does is finds the cluster (centroid) positions that minimize the distances to all points in the cluster. This is done iteratively; the problem with the algorithm is that it can be a bit greedy, meaning that it will find the nearest minima quickly. This is generally solved with some kind of basin-hopping approach where the nearest minima found is randomly perturbed and the algorithm restarted. Due to this fact, the algorithm is dependent on good initial guesses as input.

Suicide rate versus GDP versus absolute latitude

As mentioned in Chapter 4, *Regression*, we will analyze the data of suicide rates versus GDP versus absolute latitude or **Degrees From Equator** (DFE) for clusters. Our hypothesis from the visual inspection was that there were at least two distinct clusters, one with a higher suicide rate, GDP, and absolute latitude, and one with lower. We saved an HDF file in Chapter 4, *Regression*, that we now read in as a DataFrame. This time, we want to discard all the rows where one or more column entries are NaN or empty. Thus, we use the appropriate DataFrame method for this:

```
TABLE_FILE = 'data/data_ch4.h5'  
d2 = pd.read_hdf(TABLE_FILE)  
d2 = d2.dropna()
```

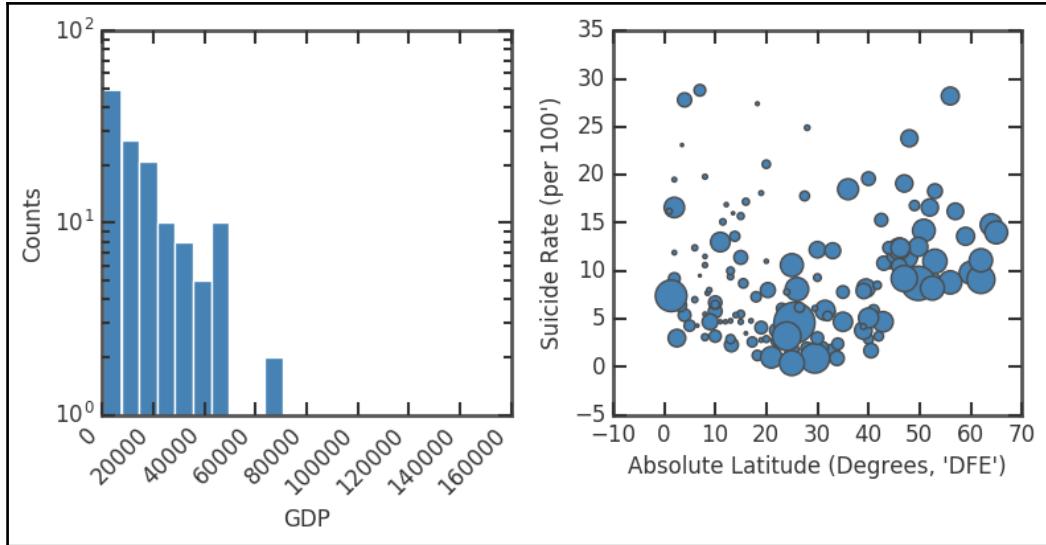
Next, while the DataFrame is a very handy format, which we will utilize later on, the input to the cluster algorithms in SciPy does not handle Pandas datatypes natively. Thus, we transfer the data to a NumPy array:

```
rates = d2[['DFE', 'GDP_CD', 'Both']].as_matrix().astype('float')
```

Next, to recap, we visualize the data with one histogram of the GDP and one scatterplot for all the data. We do this to aid us in the initial guesses of the cluster centroid positions:

```
plt.subplots(12, figsize=(8, 3.5))  
plt.subplot(121)  
plt.hist(rates.T[1], bins=20, color='SteelBlue')  
plt.xticks(rotation=45, ha='right')  
plt.yscale('log')  
plt.xlabel('GDP')  
plt.ylabel('Counts')  
plt.subplot(122)  
plt.scatter(rates.T[0], rates.T[2],  
           s=2e5 * rates.T[1] / rates.T[1].max(),  
           color='SteelBlue', edgecolors='0.3');
```

```
plt.xlabel('Absolute Latitude (Degrees, 'DFE') ')
plt.ylabel('Suicide Rate (per 100') ')
plt.subplots_adjust(wspace=0.25);
```

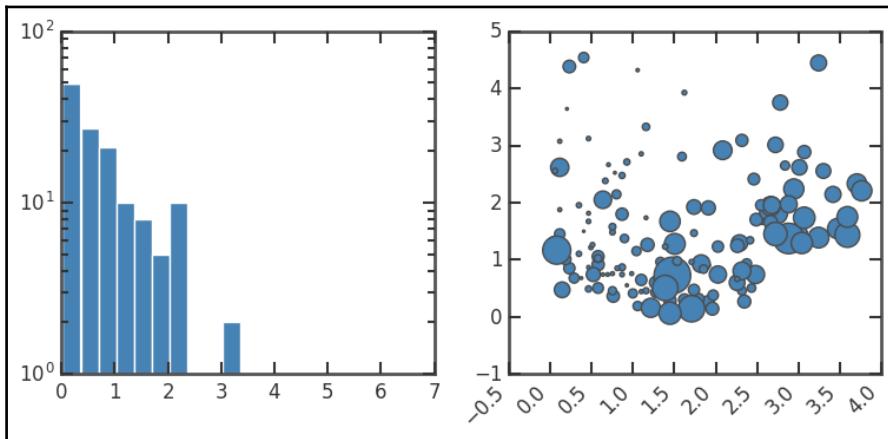


The scatter plot to the right shows the **Suicide Rate** on the y-axis and the **Absolute Latitude** on the x-axis. The size of each point is proportional to the country's GDP. The function to run the clustering k-means takes a special kind of normalized input. The data arrays (columns) have to be normalized by the standard deviation of the array. Although this is straightforward, there is a function included in the module called `whiten`. It will scale the data with the standard deviation:

```
w = vq.whiten(rates)
```

To show what it does to the data, we plot the preceding plots again, but with the output from the `whiten` function:

```
plt.subplots(12, figsize=(8, 3.5))
plt.subplot(121)
plt.hist(w[:,1], bins=20, color='SteelBlue')
plt.yscale('log')
plt.subplot(122)
plt.scatter(w.T[0], w.T[2], s=2e5*w.T[1]/w.T[1].max(),
           color='SteelBlue', edgecolors='0.3')
plt.xticks(rotation=45, ha='right');
```



As you can see, all the data is scaled from the previous figure. However, as mentioned, the scaling is just the standard deviation. So let's calculate the scaling and save it to the `sc` variable:

```
sc = rates.std(axis=0)
```

Now we are ready to estimate the initial guesses for the cluster centroids. Reading off the first plot of the data, we guess the centroids to be at 20 DFE, 200,000 GDP, and 10 suicides, and the second at 45 DFE, 100,000 GDP, and 15 suicides. We put this in an array and scale it with our scale parameter to the same scale as the output from the `whiten` function. This is then sent to the `kmeans2` function of SciPy:

```
init_guess = np.array([[20, 20E3, 10], [45, 100E3, 15]])
init_guess /= sc
z2_cb, z2_lbl = vq.kmeans2(w, init_guess, minit='matrix',
                             iter=500)
```

There is another function, `kmeans` (without the 2), which is a less complex version and does not stop iterating when it reaches a local minima; it stops when the changes between two iterations goes below some level. Thus, the standard k-means algorithm is represented in SciPy by the `kmeans2` function. The function outputs the centroids' scaled positions (here, `z2_cb`) and a lookup table (`z2_lbl`) telling us which row belongs to which centroid. To get the centroid positions in units we *understand*, we simply multiply with our scaling value:

```
z2_cb_sc = z2_cb * sc
```

At this point, we can plot the results. The following section is rather long and contains many different parts, so we will go through them section by section. However, the code should be run in one cell of the Notebook:

```
# K-means clustering figure START
plt.figure(figsize=(6, 4))
plt.scatter(z2_cb_sc[0, 0], z2_cb_sc[0, 2],
            s=5e2*z2_cb_sc[0, 1]/rates.T[1].max(),
            marker='+', color='k',
            edgecolors='k', lw=2, zorder=10, alpha=0.7);
plt.scatter(z2_cb_sc[1, 0], z2_cb_sc[1, 2],
            s=5e2*z2_cb_sc[1, 1]/rates.T[1].max(),
            marker='+', color='k', edgecolors='k', lw=3,
            zorder=10, alpha=0.7);
```

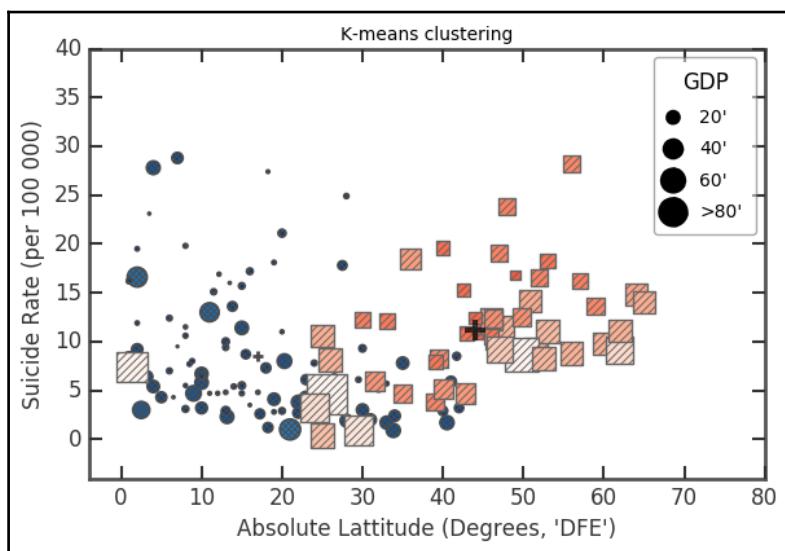
The first steps are quite simple; we set up the figure size and plot the points of the cluster centroids. We hypothesized about two clusters, thus we plot them with two different calls to `plt.scatter`. Here, `z2_cb_sc[1, 0]` gets the second cluster x coordinate (DFE) from the array, then switching 0 for 1 gives us the y coordinate (rate). We set the size of the marker to scale with the value of the third data axis, the GDP. We also do this further down for the data, just as in previous plots, so that it is easier to compare and differentiate the clusters. The `zorder` keyword gives the order in-depth of the elements that are plotted; a high `zorder` will put them on top of everything else, and a negative `zorder` will send them to the back.

```
s0 = abs(z2_lbl==0).astype('bool')
s1 = abs(z2_lbl==1).astype('bool')
pattern1 = 5*'x'
pattern2 = 4*'/'
plt.scatter(w.T[0][s0]*sc[0],
            w.T[2][s0]*sc[2],
            s=5e2*rates.T[1][s0]/rates.T[1].max(),
            lw=1,
            hatch=pattern1,
            edgecolors='0.3',
            color=plt.cm.Blues_r(
                rates.T[1][s0]/rates.T[1].max()));
plt.scatter(rates.T[0][s1],
            rates.T[2][s1],
            s=5e2*rates.T[1][s1]/rates.T[1].max(),
            lw=1,
            hatch=pattern2,
            edgecolors='0.4',
            marker='s',
            color=plt.cm.Reds_r(
                rates.T[1][s1]/rates.T[1].max()+0.4))
```

In this section, we plot the points of the clusters. First, we get the selection arrays. They are simply Boolean arrays, which are arrays where the values that correspond to either cluster 0 or 1 are True. Thus s0 is True where cluster id is 0, and s1 is True where cluster id is 1. Next, we define the hatch pattern for the scatterplot markers, which we later give the plotting function as input. The multiplier for the hatch pattern gives the density of the pattern. The scatterplots for the points are created in a similar fashion to the centroids, except the markers are a bit more complex. They are both colorcoded, as in the previous example with cholera deaths, but in a gradient instead of the exact same colors for all points. The gradient is defined by the GDP, which also defines the size of the points. The x and y data sent to the plot is different between the clusters, but they access the same data in the end because we multiply with our scaling factor.

```
p1 = plt.scatter([],[], hatch='None',
                 s=20E3*5e2/rates.T[1].max(),
                 color='k', edgecolors='None',)
p2 = plt.scatter([],[], hatch='None',
                 s=40E3*5e2/rates.T[1].max(),
                 color='k', edgecolors='None',)
p3 = plt.scatter([],[], hatch='None',
                 s=60E3*5e2/rates.T[1].max(),
                 color='k', edgecolors='None',)
p4 = plt.scatter([],[], hatch='None',
                 s=80E3*5e2/rates.T[1].max(),
                 color='k', edgecolors='None',)
labels = ["20'", "40'", "60'", ">80'"]
plt.legend([p1, p2, p3, p4], labels, ncol=1,
           frameon=True, fontsize=12,
           handlelength=1, loc=1,
           borderpad=0.75, labelspacing=0.75,
           handletextpad=0.75, title='GDP', scatterpoints=1.5)
plt.ylim((-4,40))
plt.xlim((-4,80))
plt.title('K-means clustering')
plt.xlabel('Absolute Latitude (Degrees, 'DFE') ')
plt.ylabel('Suicide Rate (per 100 000)');
```

The last tweak to the plot is made by creating a custom legend. We want to show the different sizes of the points and what GDP they correspond to. As there is a continuous gradient from low to high, we cannot use the plotted points. Thus we create our own, but leave the x and y input coordinates as empty lists. This will not show anything in the plot but we can use them to register in the legend. The various tweaks to the legend function control different aspects of the legend layout. I encourage you to experiment with it to see what happens:



As for the final analysis, two different clusters are identified. Just as our previous hypothesis, there is a cluster with a clear linear trend with relatively higher GDP, which is also located at a higher absolute latitude. Although the identification is rather weak, it is clear that the two groups are separated. Countries with low GDP are clustered closer to the equator. What happens when you add more clusters? Try to add a cluster for the low DFE high-rate countries, visualize it, and think about what this could mean for the conclusion(s).

Hierarchical clustering analysis

Hierarchical clustering is connectivity-based clustering. It assumes that the clusters are connected, or in another word, linked. For example, we can classify animals and plants based on this assumption. We have all developed from something common. This makes it possible for us to assume that every observation is its own cluster on one hand and, on the other, all observations are in one and the same group. This also forms the basis for two approaches to hierarchical clustering algorithms, agglomerative and divisive:

- **Agglomerative clustering** starts out with each point in its own cluster and then merges the two clusters with the lowest dissimilarity, that is, the bottom-up approach
- **Divisive clustering** is, as the name suggests, a top-down approach where we start out with one single cluster that is divided into smaller and smaller clusters

In contrast to k-means, it gives us a way to identify the clusters without initial guesses of the number of clusters or cluster positions. For this example, we will run an agglomerative cluster algorithm in SciPy.

Reading in and reducing the data

Galaxies in the Universe are not randomly distributed, they form clusters and filaments. These structures hint at the complex movement and history of the Universe. There are many different catalogs of galaxy clusters, although the techniques to classify clusters vary and there are several views on this. We will use the Updated Zwicky Catalog, which contains 19,367 galaxies (Falco et al. 1999, PASP 111, 438). The file can be downloaded from <http://tdc-www.harvard.edu/uzc/index.html>. The first Zwicky Catalog of Galaxies and Clusters of Galaxies was released in 1961 (Zwicky et al. 1961-1968. Catalog of Galaxies and Clusters of Galaxies, Vols. 1-6. CalTech).

To start, we import some required packages, read in the file to a DataFrame, and investigate what we have:

```
import astropy.coordinates as coord
import astropy.units as u
import astropy.constants as c
```

Astropy is a community-developed astronomy package to help Astronomers analyze and create powerful software to handle their data (<http://www.astropy.org/>). We import the coordinates package that can handle astronomical coordinates (**World Coordinate System–WCS**) and transforms. The units and constants packages are packages to handle physical units (conversions and so on) and constants (with units); both are extremely handy to do calculations where the units matter:

```
uzcat = pd.read_table('data/uzcJ2000.tab/uzcJ2000.tab',
                      sep='\t', header=16, dtype='str',
                      names=['ra', 'dec', 'Zmag', 'cz', 'cze', 'T', 'U', 'Ne',
                             'Ne', 'Zname', 'C', 'Ref', 'Oname', 'M', 'N'],
                      skiprows=[17])
```

Let's look at the data with the head method:

```
uzcat.head()
```

	ra	dec	Zmag	cz	cze	T	U	Ne	Zname	C	Ref	Oname	M	N
0	000237.9	+163838	14.9	6350	19	A	1	0	000000+16220	F		I5378S		
1	000246.3	+185310	14.8	7864	47	A	0	0	000012+18370	Z	0650	00002+1837		
2	000257.0	+041231	15.5	8695	40	E	0	0	000030+03560	Z	2700	00005+0356		
3	000302.9	+185221	15.5	8007	39	E	0	0	000030+18360	Z	0650	00005+1836		
4	000305.6	-015450	14.3	7298	42	B	0	0	000036-02110	Z	2218	00006-0211		

The first two columns, `ra` and `dec`, are coordinates in the equatorial coordinate system. Basically, if you imagine Earth's latitude and longitude system expanded, we are on the inside. RA, or Right Ascension, is the longitude and Dec, or declination, is the latitude. A consequence of this is that, as we are on the inside of it, East is West and West is East. The third column is the Z magnitude, which is a measure of how bright the galaxy is (in logarithmic units) measured at a certain wavelength of light. The fourth column is the redshift distance in units of km/s (fifth is the uncertainty) with respect to our Sun (that is, Heliocentric distance). This odd unit is the redshift multiplied by the speed of light ($v = cz$, z : redshift). Due to the simplicity of this, the `v` parameter can have speeds that go above the speed of light, that is, non-physical speeds. It assumes that the radial speed of every galaxy in the universe is dominated by the expansion of the universe. Recollecting that in Chapter 3, *Learning About Models*, we looked at Hubble's Law, the expansion of the universe increases linearly with distance. While the Hubble constant is constant for short distances, today we know that the expansion speed of the universe (that is, Hubble constant, H) changes at large distances, the change depending on what cosmology is assumed. We will convert this distance to something more graspable later on.

The rest of the columns are described either in the accompanying README file or online at <http://tdc-www.harvard.edu/uzc/uzcjformat.html>.

First, we want to translate the coordinates into something more readable than a string (that is, `ra` and `dec` columns). Equatorial coordinates are given for RA in hours, minutes, and seconds and Dec in degrees, minutes, and seconds. To get degrees from hours, you simply multiply hours by 15 (that is, 360 degrees divided by 24 hours). One of the first reasons to choose this dataset as an example is that because of the coordinate system, the distance is not the Euclidean distance. To be able to use it, we have to translate the coordinates into Cartesian coordinates, which we will do soon. As explained, we now fix the first thing with the coordinates; we convert them into understandable strings:

```
df['ra'] = df['ra'].apply(lambda x: '{0}h{1}m{2}s'.format(
                           x[:2], x[2:4], x[4:]))
df['dec'] = df['dec'].apply(lambda x: '{0}d{1}m{2}s'.format(
                            x[:3], x[3:5], x[5:]))
df.head()
```

	ra	dec	Zmag	cz	cze	T	U	Ne	Zname	C	Ref	Oname	M	N
0	00h02m37.9s	+16d38m38s	14.9	6350	19	A	1	0	000000+16220	F		I5378S		
1	00h02m46.3s	+18d53m10s	14.8	7864	47	A	0	0	000012+18370	Z	0650	00002+1837		
2	00h02m57.0s	+04d12m31s	15.5	8695	40	E	0	0	000030+03560	Z	2700	00005+0356		
3	00h03m02.9s	+18d52m21s	15.5	8007	39	E	0	0	000030+18360	Z	0650	00005+1836		
4	00h03m05.6s	-01d54m50s	14.3	7298	42	B	0	0	000036-02110	Z	2218	00006-0211		

Next, we need to put `np.nan` where the entry is empty (we are checking whether it is an empty string with spaces). With `apply`, you can apply a function to a certain column/row, and `applymap` applies a function to every table entry:

```
uzcat = uzcat.applymap(lambda x: np.nan if
                       isinstance(x, str) and
                       x.isspace() else x)
uzcat['cz'] = uzcat['cz'].astype('float')
```

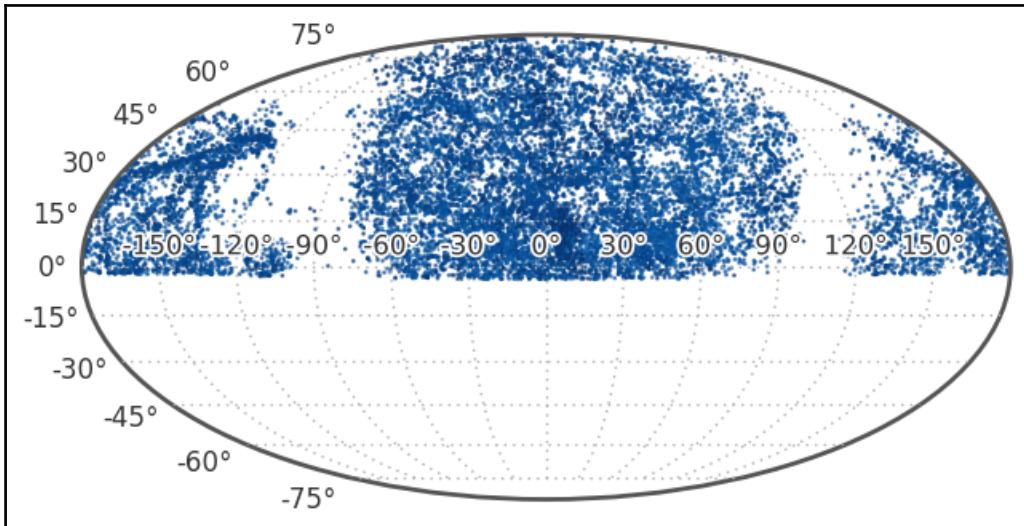
We also convert the magnitude column to floats by running `mycat.Zmag = mycat.Zmag.astype('float')`. To do an initial visualization of the data, we need to convert the coordinates to radians or degrees, something that `matplotlib` understands. To do this, we use the convenient `Astropy Coordinates` package:

```
coords_uzc = coord.SkyCoord(uzcat['ra'], uzcat['dec'], frame='fk5',
                             equinox='J2000')
```

We can now access the coordinates in one object and convert them to various units. For example, `coords_uzc.ra.deg.min()` will return the minimum RA coordinate in units of degrees; replacing `deg` with `rad` will return it in radians. Visualizing it at this level has several reasons; one reason for this is that we want to check what the coordinates cover; what part of the sky we are looking at. To do this, we use a projection method; otherwise, the coordinates do not make sense as they are not common x,y,z coordinates (in this case, the Mollweide projection), so we are looking at the whole sky flattened out:

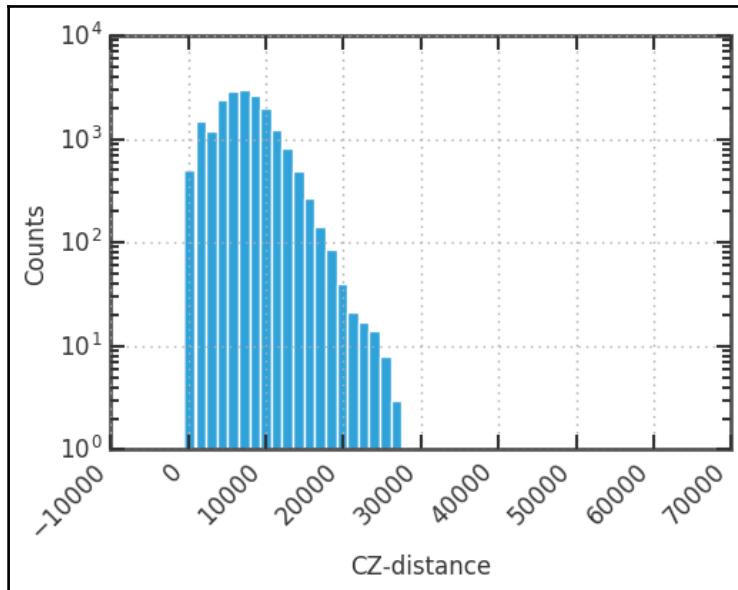
```
color_czs = (uzcat['cz']+abs(uzcat['cz'].min())) /  
             (uzcat['cz'].max()+abs(uzcat['cz'].min()))  
from matplotlib.path_effects import withStroke  
whitebg = withStroke(foreground="w", linewidth=2.5)  
fig = plt.figure(figsize=(8,3.5))  
ax = fig.add_subplot(111, projection="mollweide")  
ax.scatter(coords_uzc.ra.radian-np.pi, coords_uzc.dec.radian,  
           color=plt.cm.Blues_r(color_czs), alpha=0.6,  
           s=4, marker='.', zorder=-1)  
plt.grid()  
for tick in ax.get_xticklabels():  
    tick.set_path_effects([whitebg])
```

As the scatter points are dark, I have also modified the tick labels with path effects, which was introduced in matplotlib 1.4. This makes it much easier to distinguish the coordinate labels:



We can see that we have data for the upper part of the sky only. We also see the extent of the Milky Way, its gas and dust is blocking our view beyond it and no galaxies are found there in the dataset. To minimize the data that we look at, we will cut it along the Dec between 15 and 30 degrees. Let's check the distribution of distances that are covered:

```
uzcat['cz'].hist(bins=50)
plt.yscale('log')
plt.xlabel('CZ-distance')
plt.ylabel('Counts')
plt.xticks(rotation=45, ha='right');
```



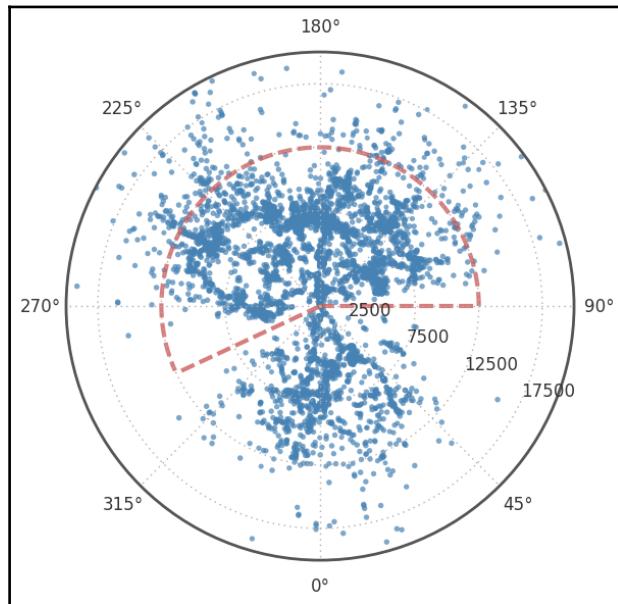
The peak is around 10,000 km/s, and we cut it off at 12,500 km/s. Let's visualize this cut from top-down. Instead of looking at both RA and Dec, we will look at RA and cz. First, we create the selection:

```
uzc_czs = uzcat['cz'].as_matrix()
uzcat['Zmag'] = uzcat['Zmag'].astype('float')
decmin = 15
decmax = 30
ramin = 90
ramax = 295
czmin = 0
czmax = 12500
selection_dec = (coords_uzc.dec.deg>decmin) *
```

```
(coords_uzc.dec.deg<decmax)
selection_ra = (coords_uzc.ra.deg>ramin) *
               (coords_uzc.ra.deg<ramax)
selection_czs = (uzc_czs>czmin) * (uzc_czs<czmax)
selection= selection_dec * selection_ra * selection_czs
```

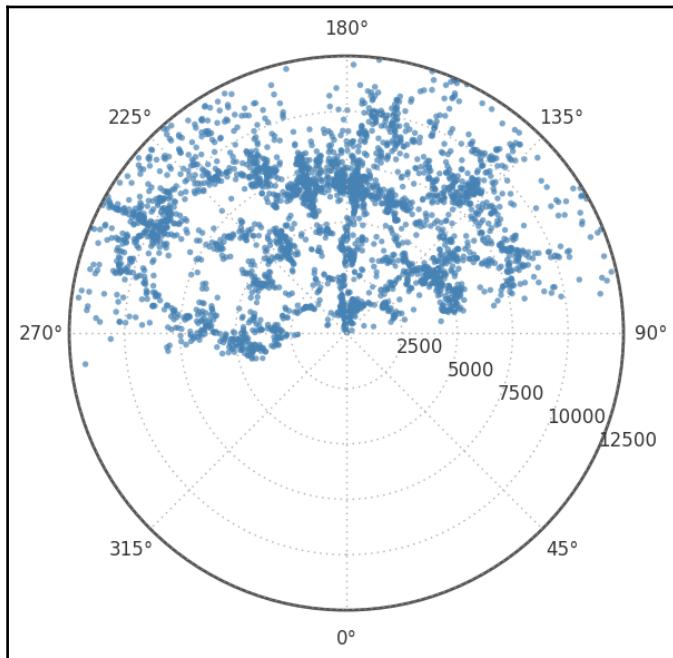
We export the `cz` column from the DataFrame just for convenience; we create a separate Boolean array for each selection. This way, we can filter whatever we want. For example, calling `coords_uzc.ra.radian[selection_dec*selection_ra]` will only filter out the RA and Dec that we are after. Next, we plot it all with only the `Dec` filter, and then visualize where we are about to cut in `cz` and `RA`. I have not explained the cut in `RA` and `cz` chosen here, but it was done after looking at the following image:

```
fig = plt.figure( figsize=(6,6))
ax = fig.add_subplot(111, polar=True)
sct = ax.scatter(coords_uzc.ra.radian[selection_dec],
                  uzc_czs[selection_dec],
                  color='SteelBlue',
                  s=uzcat['Zmag'][selection_dec*selection_czs],
                  edgecolors="none",
                  alpha=0.7,
                  zorder=0)
ax.set_rlim(0,20000)
ax.set_theta_offset(np.pi/-2)
ax.set_rlabel_position(65)
ax.set_rticks(range(2500,20001,5000));
ax.plot([(ramin*u.deg).to(u.radian).value,
          (ramin*u.deg).to(u.radian).value], [0,12500],
          color='IndianRed', alpha=0.8, dashes=(10,4))
ax.plot([ramax*np.pi/180., ramax*np.pi/180.], [0,12500],
          color='IndianRed', alpha=0.8, dashes=(10,4))
theta = np.arange(ramin, ramax, 1)
ax.plot(theta*np.pi/180., np.ones_like(theta)*12500,
          color='IndianRed', alpha=0.8, dashes=(10,4))
```



Here, we use a polar plot by passing `polar=True` when we instantiate the subplot and applying the `selection_dec` filters to all `Dec` values above 30 and below 15 degrees. The coordinates need to be given in radians, hence we go over to radians by simply asking for the coordinates in radians as previously described. Next, we customize the plot, rotating the plot 90 degrees clockwise, which is $\pi/2$ in radians. To make it easier to read the radial distance axis labels, we set them to be drawn at 65 degrees and set the distance at which they should be drawn. The last two function calls draw the dashed region, the region which I set as `selection_ra` and `selection_czs`. Next, we plot only the selection points and zoom in a bit:

```
fig = plt.figure( figsize=(6,6) )
ax = fig.add_subplot(111, polar=True)
sct = ax.scatter(coords_uzc.ra.radian[selection],
                  uzc_czs[selection],
                  color='SteelBlue',
                  s=uzcat['Zmag'][selection],
                  edgecolors="none",
                  alpha=0.7,
                  zorder=0)
ax.set_rlim(0,12500)
ax.set_theta_offset(np.pi/-2)
ax.set_rlabel_position(65)
ax.set_rticks(range(2500,12501,2500));
```



It is important to notice that most coordinates are above the line that 90 to 270 degrees forms. This will later have an effect on the Cartesian coordinates. With a subsection of the total catalog, it is a good idea to create a separate DataFrame to store everything in it, including the coordinates in degrees:

```
mycat = uzcat.copy(deep=True).loc[selection]
mycat['ra_deg'] = coords_uzc.ra.deg[selection]
mycat['dec_deg'] = coords_uzc.dec.deg[selection]
```

Although RA, Dec, and cz is a perfectly understandable coordinate format for astronomers, it is not for most people (it is even hard to digest for astronomers). So we shall now convert these spherical coordinates (Celestial Equatorial Coordinate System) to x, y, z. To do this, we use some very convenient functions in the Astropy Coordinates package, which we have already worked with. First, we calculate the actual distance to the galaxies. To do this, we use the Distance function, which can be done with different geometries of the universe/cosmologies, but the default (current) is fine for us:

```
zs = (((mycat['cz'].as_matrix()*u.km/u.s) / c.c).decompose())
dist = coord.Distance(z=zs)
print(dist)
mycat['dist'] = dist
```

We now have everything to calculate the Cartesian coordinates of the galaxies:

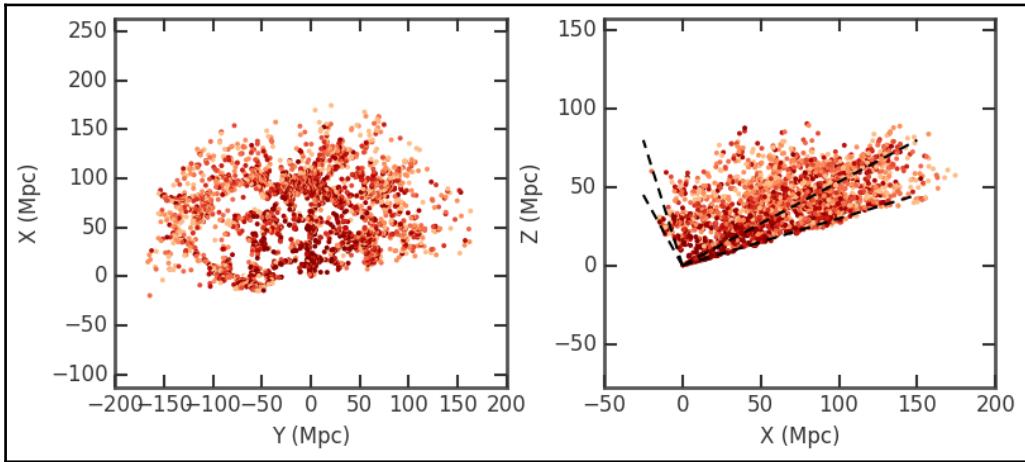
```
coords_xyz = coord.SkyCoord(ra=mycat['ra_deg']*u.deg,
                             dec=mycat['dec_deg']*u.deg,
                             distance=dist*u.Mpc,
                             frame='fk5',
                             equinox='J2000')
```

Now is a good time to save these Cartesian coordinates to our catalog:

```
mycat['X'] = coords_xyz.cartesian.x.value
mycat['Y'] = coords_xyz.cartesian.y.value
mycat['Z'] = coords_xyz.cartesian.z.value
```

I suggest running `head()` and `describe()` on the current DataFrame catalog that we have created (that is, `mycat`). Notice then that most X coordinates are negative. Why is this? Remember what RA coordinates most of our selection had? Go back and check the polar plots that we made. The RA is between 90 and 270 degrees; basically the opposite direction of 0 degrees, causing them to have negative X coordinates now. Now I want to plot this; as it is actually three-dimensional data, I will use two plots to visualize the three dimensions:

```
fig, axs = plt.subplots(1,2, figsize=(14,6))
plt.subplot(121)
plt.scatter(mycat['Y'], -1*mycat['X'], s=8,
            color=plt.cm.OrRd_r(10**-(mycat.Zmag
                           -mycat.Zmag.max()))),
            edgecolor='None')
plt.xlabel('Y (Mpc)'); plt.ylabel('X (Mpc)')
plt.axis('equal');
plt.subplot(122)
plt.scatter(-1*mycat['X'], mycat['Z'], s=8,
            color=plt.cm.OrRd_r(10**-(mycat.Zmag
                           -mycat.Zmag.max()))),
            edgecolor='None')
lststyle = dict(lw=1.5, color='k', dashes=(6,4))
plt.plot([0,150], [0,80], **lststyle)
plt.plot([0,150], [0,45], **lststyle)
plt.plot([0,-25], [0,80], **lststyle)
plt.plot([0,-25], [0,45], **lststyle)
plt.xlabel('X (Mpc)'); plt.ylabel('Z (Mpc)')
plt.axis('equal')
plt.subplots_adjust(wspace=0.25);
```



Visualizing it like this gives you a good overview, even if the data is three-dimensional. We could have made the selection after converting to x , y , and z coordinates and cut out a cube. Try this out and see how the results of this exercise differ. Furthermore, I recommend that you try to make a three-dimensional plot at this stage; refer to the previous chapter (Chapter 4, *Regression*) for the code. We have now reduced the catalog to cover the region that we are interested in and mapped some of the columns to values easily readable by functions. This is the time to save it in order to make it easier to start where we left off. Just like in the previous chapter, we use the HDF file format:

```
TABLE_FILE = 'data/data_ch5_clustering.h5'  
mycat.to_hdf(TABLE_FILE, 'ch5data', mode='w', table=True)
```

As an alternative, if you have problems with the HDF libraries or just want an alternative, you can also save it with the pickle module, which is a standard module in Python:

```
mycat.to_pickle('data/data_ch5_clustering.pick')
```

We will read this data in Chapter 7, *Supervised and Unsupervised Learning*. Now we have reduced the data so that we can run our clustering analysis on it.

Hierarchical cluster algorithm

The hierarchical agglomerative clustering algorithm is run in SciPy through the `linkage` function with this array as input. There are two main parameters to set in the `linkage` function, method and metric:

- **Method** defines the linkage algorithm to use, that is, how we estimate the dissimilarities between two clusters and thus define how clusters are formed
- **Metric** defines the distance metric; in this case, we are working with non-categorical variables, where distance makes sense

In our case, we have converted the data to Cartesian coordinates so that we can use the common Euclidean distance. It is possible to define your own distance function. Possible methods and metrics in the `linkage` function are listed in the SciPy documentation at <http://docs.scipy.org/>.

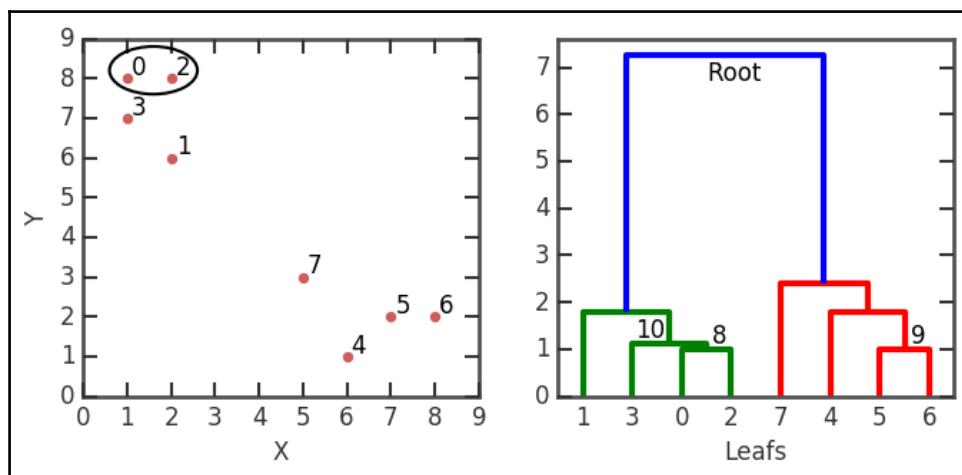
The `linkage` function takes an N by 2 array (N data points), so here we use only the X and Y coordinates:

```
galpos = np.array([mycat.X, mycat.Y]).T
z_centroid = hac.linkage(galpos, metric = 'euclidean',
                         method = 'centroid')
```

The output here is the linkage matrix. This is the result for the whole run; it contains four columns. To quickly illustrate what it contains, we run a controlled and much smaller example. To visualize the various group levels, we also plot a dendrogram, which is done with the `hac.dendrogram` function that takes the linkage output as input. It visualizes the clustering sequence in a handy way. The root level is the level where the whole dataset is in one cluster. At the other end is a lower level of clustering; however, they are connected to the root level. Each node represents a group of clusters and each node connects to two subnodes. If all levels are plotted, the end node (node at the lowest level) is called a leaf node and contains only one data point (observation). How these are connected is determined by the linkage definition (dissimilarity measuring algorithm):

```
x = np.array([1,2,2,1,6,7,8,5])
y = np.array([8,6,8,7,1,2,2,3])
a = np.array([x,y]).T
z = hac.linkage(a, metric = 'euclidean', method = 'centroid')
fig, axs = plt.subplots(1,2, figsize=(7,3))
axs[0].scatter(x,y, marker='o', s=40, c='IndianRed')
axs[0].set_xlabel('X'); axs[0].set_ylabel('Y');
for i in range(len(x)):
    axs[0].annotate(s=str(i), xy=(x[i]+0.1,y[i]+0.1))
ellipse1 = Ellipse(xy=(1.6,8.2),
```

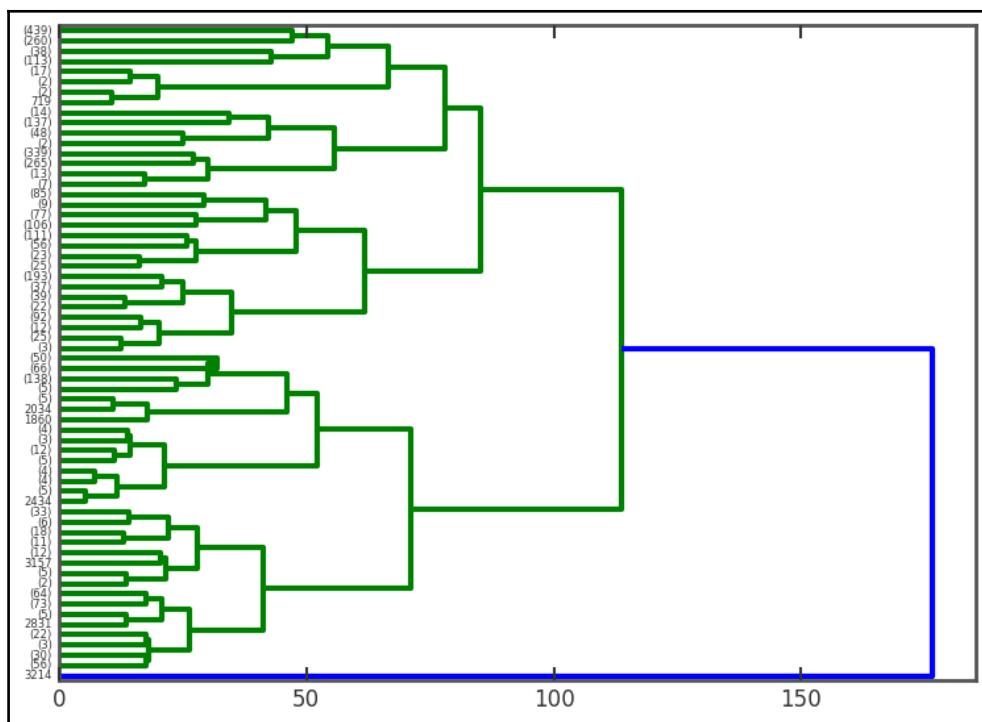
```
width=2., height=1.2,
      zorder=32, fc='None', ec='k', lw=1.5)
axs[0].add_artist(ellipse1)
d_temp = hac.dendrogram(z, ax=axs[1])
axs[1].annotate(s='8',
                 xy=(np.mean(d_temp['icoord'][0][1:-1]),
                      d_temp['dcoord'][0][1]),
                 xytext=(3,3), textcoords='offset points')
axs[1].annotate(s='9',
                 xy=(np.mean(d_temp['icoord'][3][1:-1]),
                      d_temp['dcoord'][3][1]),
                 xytext=(3,3), textcoords='offset points')
axs[1].annotate(s='10',
                 xy=(np.mean(d_temp['icoord'][1][1:-1])-2,
                      d_temp['dcoord'][1][1]),
                 xytext=(3,3), textcoords='offset points',
                 ha='right')
axs[1].annotate(s='Root',
                 xy=(np.mean(d_temp['icoord'][-1][1:-1]),
                      d_temp['dcoord'][-1][1]-0.3),
                 xytext=(5,5), textcoords='offset points',
                 va='top', ha='center')
axs[1].set_xlabel('Leafs')
```



In the linkage output z , the first row is $[0., 2., 1., 2.]$, which says the following: cluster index 0 and 2 form a group, their distance is 1, and the number of leafs (data points) in the group is 2. This group is encircled in the image. The group number is the number of original points plus the iteration number ($n+i$); in this case, we form cluster 8 (8+0). The third row (NB, not the second!) has the numbers $[3., 8., 1.11803399, 3.]$ in it. It combines cluster index 3, which in this case is just the point because every cluster number less than 8 is a point/leaf with cluster 8, just what we created in the first iteration. The distance between them (with the given metric and method) is 1.111803399 and it contains three leafs. It forms the 10th cluster, that is, 8 + 2 iterations. I have illustrated these nodes in the example dendrogram and also marked cluster group number 10. Think about how it fits into the linkage output described here.

With this knowledge, we can now plot the dendrogram for the main data analysis:

```
fig, ax = plt.subplots(1, figsize=(8,6))
d0 = hac.dendrogram(z_centroid, p=6, truncate_mode='level',
                      orientation='right', ax=ax)
```

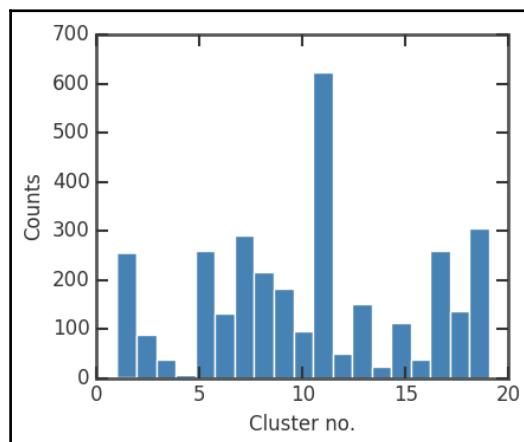


This time, I have added some parameters to the plotting function-we have tipped the whole dendrogram 90 degrees. The height between two nodes is proportional to the dissimilarity between the nodes (the distance method). Cutting this dendrogram across will give a certain amount of clusters. It's interesting to see that the root node divides only once into two clusters, where only one of them continues to divide. To get the clusters at a certain level, we use the `fcluster` function, also part of SciPy's clustering module. I suggest that you try different amount of clusters and plot them. In the following example, I have used 20 clusters:

```
nclust = 20
part_centroid = hac.fcluster(z_centroid, 20, criterion='maxclust')
```

Here, criterion sets the constraints on forming the clusters. Just to check the division of points in each cluster/group, we plot a histogram:

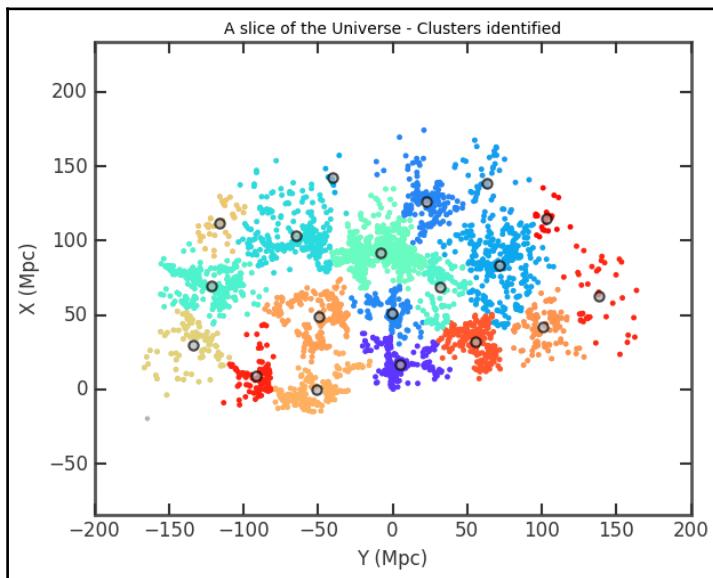
```
plt.figure(figsize=(7, 6))
otpt = plt.hist(part_centroid, color='SteelBlue', bins=nclust);
plt.xlabel('Cluster no.); plt.ylabel('Counts');
```



There are a lot of points in the 11th cluster. Of course, this does not tell us much, so now we plot all the clusters. As the position of each cluster, I just calculate the centroid with the mean method of the array object. This is just to mark the location of clusters more clearly:

```
plt.figure(figsize=(6, 5))
plt.subplot(111)
part = part_centroid
levels = np.arange(nclust)
colors = plt.cm.rainbow(np.random.rand(len(levels)))
for n, color in zip(levels, colors):
```

```
plt.scatter(mycat['Y'][part==n], -1*mycat['X'][part==n], s=12,
           color=color, edgecolor='None')
plt.plot(mycat['Y'][part==n].mean(),
         -1*mycat['X'][part==n].mean(),
         'o', c='0.7', mec='k', ms=6,
         ls='None', mew=1.5, alpha=0.7)
plt.xlabel('Y (Mpc)'); plt.ylabel('X (Mpc)')
plt.scatter(mycat['Y'], -1*mycat['X'], s=10,
            color='0.7', edgecolor='None', zorder=-1)
plt.title('A slice of the Universe - Clusters identified')
plt.axis('equal');
```



The cluster division looks solid. However, we determined the number of clusters by visually inspecting the effect of changing the number of clusters. This might not be the true or ideal number of clusters. In the previous example, we had a hypothesis of two clusters that we wanted to test, but even in that case the data might be better represented by, for example, three clusters or perhaps none (or one cluster with outliers). We want to have a reproducible way of determining the number of clusters. There are several approaches to this; there is even a dedicated Wikipedia article on how to find the number of clusters (http://en.wikipedia.org/wiki/Determining_the_number_of_clusters_in_a_data_set). The main problem with determining the optimum number of clusters is that we have to assume something about the cluster shapes. Hierarchical clustering circumvents this problem slightly by checking all the levels of a cluster with the assumption that each cluster is divided up into smaller clusters.

One approach measures the cluster compactness by calculating the normalized sum of the squared distances (that is, variance) for each cluster and then using this to estimate what percentage of the data that can be described by this variance in cluster size.

Incrementally increasing the number of clusters, you get a graph of the increase in the coverage of the cluster by calculating the variance. When the true number of clusters has been reached, this variance (or coverage) will stop increasing and flatten out. However, this does not fit all clump shapes; imagine, for example, very elongated ellipses close to each other. In our case, the clusters at medium distances are more filamentary than clumpy (or Gaussian-like).

Summary

We have now identified clusters using a range of methods, from calculating simple centroids manually to advanced hierarchical clustering algorithms in SciPy. There are of course many more packages in Python. We will look at one alternative, the machine learning package, Scikit-learn, in Chapter 7, *Supervised and Unsupervised Learning*, to identify clusters. SciPy has these two clustering frameworks, that is, vector quantization and hierarchical clustering, which lay the foundation for cluster analysis and are very useful in many general data analysis problems. In the next chapter, we will look at Bayesian analysis and how to use the PyMC Bayesian inference package in Python to characterize various things in data.

6

Bayesian Methods

Bayesian inference is a different paradigm for statistics; it is not a method or algorithm such as cluster finding or linear regression. It stands next to classical statistical analysis.

Everything that we have done so far in this book, and everything that you can do in classical (or frequentist) statistical analysis, you can do in Bayesian statistics. The main difference between frequentist (classical) and Bayesian statistics is that while frequentist assumes that the model parameters are fixed, Bayesian assumes that they have a range, a distribution. Thus, from the frequentist approach, it is easy to create point estimates—mean, variance, or fixed model parameters—directly from the data. The point estimates are unique to the data; each new dataset needs new point estimates.

In this chapter, we will cover the following topics:

- Examples of Bayesian analysis: one where we try to identify a switch point in a time series and another with linear regression, where we compare the methods from Chapter 4, *Regression*
- How to assess the MCMC run from Bayesian analysis
- A very short introduction on plotting coordinates on maps, which can be very important when presenting and investigating data

The Bayesian method

From the Bayesian approach, data is seen as fixed. Once we have measured things, the values are fixed. On the other hand, parameters can be described by probability distributions. The probability distribution describes how much is known about a certain parameter. This description might change if we get new data, but the model itself will not change. There is lots of literature on this, and there is no rule of thumb for when to use frequentist or when to use Bayesian analysis.

For simple and fairly well-behaved data, I would say that the frequentist approach is fine when you need a quick estimate. To get more insights and for more constrained problems, that is, when we know more about our parameters and can estimate the prior distributions with more than a simple uniform prior, it is better to use the Bayesian approach. Due to the slightly more intuitive handling of things in Bayesian analysis, it is easier to build more complex models and answer complex questions.

Credible versus confidence intervals

A nice and common way to highlight the differences is to compare the confidence interval of frequentists with the corresponding notion in Bayesian statistics, credible interval. Confidence interval is from the frequentist's approach, where the parameter is fixed. The confidence interval is based on the repetition of the observations. A 98% confidence interval means that repeating the experiment to measure the parameter a large number of times and calculating the interval for each experiment, 98% of the intervals will contain the value of the parameter. This goes back to the fact that the data is random.

Credible (or probability) interval stems from probabilities, that is, the Bayesian approach. This means that the parameter is random and we can say that, given the data, there is a 98% chance that the true value of the parameter is in the interval.

Bayes formula

Bayesian analysis boils down to Bayes formula; thus, a chapter about Bayesian analysis without mentioning Bayes formula would not be worth much. In Bayesian analysis, everything done can be expressed in a probability statement:

$$P(a|b)$$

This is read as probability of a given b, where b is the data and a is the parameter that you are trying to estimate. With Bayesian analysis, we build models that we can test against data. Bayesian analysis (inference) uses probability distribution(s) as input to the model (hypothesis) that we are building (testing).

With some prior knowledge of statistics in our luggage, we write out Bayes formula:

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)}$$

The Bayes formula follows from conditional probabilities. Describing it in words, the posterior probability is the probability of the observations (given the parameter) times the prior of the parameter divided by an integral over the whole parameter space. It is this denominator that causes trouble with the analysis because, to calculate it, we need to employ some advanced analysis; in these examples, the **Markov Chain Monte Carlo (MCMC)** algorithm is used. We assume that you are familiar with the basics of Bayes formula. We will see how to implement the analysis in Python with the help of the PyMC package.

Python packages

One popular Bayesian analysis package in Python is PyMC (<http://pymc-devs.github.io/pymc/>). PyMC is an actively developed package to make Bayesian models and fitting in Python accessible and straightforward. One of the fitting algorithms available is MCMC, which is also one of the most employed. There are other packages such as the feature-rich emcee package (<http://dan.iel.fm/emcee/current/>). We will use PyMC in this book. To install PyMC in Anaconda, open an Anaconda command prompt (or terminal window) and use the conda command to install PyMC:

```
conda install pymc
```

It will check the Anaconda Python package index and download and install/upgrade the necessary dependencies for PyMC. Next, we can start a new Jupyter Notebook and put in the default imports.

U.S. air travel safety record

In this example, we will look at a dataset from the U.S. **National Transportation Safety Board (NTSB)**. The NTSB has an open database that can be downloaded from their web page, <http://www.ntsb.gov>. One important thing about the data is that it contains *civil aviation accidents and selected incidents within the United States, its territories and possessions, and in international waters*, that is, it is not for the whole world. Basically, it is for U.S.-related accidents only, which makes sense for a U.S. national organization. There are databases that contain the whole world, but with less fields in them. For example, the NTSB dataset contains information about minor injuries for the accident in question. For comparison, and as an opening for exercises, after the Bayesian analysis of the NTSB data, we shall load and have a quick look at a dataset from OpenData by Socrata (<https://opendata.socrata.com>) that covers the whole world. The question that we want to investigate in this section is whether there are any jumps in the statistics for plane accidents with time. An alternative source is the Aviation Safety Network (<https://aviation-safety.net>). One important point before starting the analysis is that we shall once again read in the real raw data and clean it from unwanted parts so that we can focus on the analysis. This takes a few lines of coding, but it is very important to cover this as this shows you what is really happening and you will understand the results and data much better than if I would give you the finished cleaned and reduced data (or even created data with *random noise*).

Getting the NTSB database

To download the data from NTSB, go to their web page (<http://www.ntsb.gov>), click on **Aviation Accident Database**, and select **Download All (Text)**. The data file, `AviationData.txt`, should now be downloaded and saved for you to read in.

The dataset contains date stamps for when a specific accident took place. To be able to read in the dates into a format that Python understands, we need to parse the date strings with the `datetime` package. The `datetime` package is a standard package that comes with Python (and the Anaconda distribution):

```
from datetime import datetime
```

Now let's read in the data. To spare you some time, I have redefined the column names so that they are more easily accessible. By now, you should be familiar with the very useful Pandas `read_csv` function:

```
aadata = pd.read_csv('data/AviationData.txt',
                     delimiter='|',
                     skiprows=1,
                     names=['id', 'type', 'number', 'date',
                            'location', 'country', 'lat', 'long', 'airport_code',
                            'airport_name', 'injury_severity', 'aircraft_damage',
                            'aircraft_cat', 'reg_no', 'make', 'model',
                            'amateur_built', 'no_engines', 'engine_type', 'FAR_desc',
                            'schedule', 'purpose', 'air_carrier', 'fatal',
                            'serious', 'minor', 'uninjured',
                            'weather', 'broad_phase', 'report_status',
                            'pub_date', 'none'])
```

Listing the column names shows that there is a wealth of data here, such as location, latitude, and longitude of the accident, the airport code and name, and so on:

```
aadata.columns
```

```
Index(['id', 'type', 'number', 'date', 'location', 'country', 'lat', 'long',
       'airport_code', 'airport_name', 'injury_severity', 'aircraft_damage',
       'aircraft_cat', 'reg_no', 'make', 'model', 'amateur_built',
       'no_engines', 'engine_type', 'FAR_desc', 'schedule', 'purpose',
       'air_carrier', 'fatal', 'serious', 'minor', 'uninjured', 'weather',
       'broad_phase', 'report_status', 'pub_date', 'none'],
      dtype='object')
```

After looking at the data and trying some of the following things, you will discover that some date entries are empty, that is, just containing whitespace (). We could, as before, find the entries with whitespace using the `apply(-map)` function(s) and replace the values. In this case, I will just quickly filter them out with a Boolean array produced by the matching expression, `!=`, that is, not equal to (we only want the rows with dates). This is because, as mentioned in the beginning, we want to see how the accidents vary with time:

```
selection = aadata['date'] != ''
aadata = aadata[selection]
```

Now the actual date strings are a bit tricky. They have the MONTH/DAY/YEAR format (for me as a European, this is illogical). The standard `datetime` module can do the job as long as we tell it what format the dates are in. In practice, Pandas can parse this when reading the data with the `parse_dates=X` flag, where X is either a column index integer or column name string.

However, sometimes it does not work well without putting in a lot of work, as in our case, so we shall parse it on our own. Here, we parse it into the new column, `datetime`, where each date is converted to a `datetime` object with the `strptime` function. One of the reasons for this is that the date is given with whitespace around it. So it is given as 02/18/2016 instead of 02/18/2016, which is also why we give the date format specification as `%m/%d/%Y`, that is, with whitespace around it. This way, the `strptime` function knows what it looks like:

```
aadata['datetime'] = [datetime.strptime(x, ' %m/%d/%Y ') for x in  
aadata['date']]
```

Now that we have `datetime` objects, Python knows what the dates mean. Only checking just the year or month could be interesting later on. To have it handy, we save those in separate columns. With a moderate dataset like this, we can create many columns without slowing down things noticeably:

```
aadata['month'] = [int(x.month) for x in aadata['datetime']]  
aadata['year'] = [int(x.year) for x in aadata['datetime']]
```

Now we also want the dates as decimal year, so we can bin them year by year and calculate yearly statistics. To do this, we want to see what fraction of a year has passed for a certain date. Here, we write a small function, the solution inspired in part by various answers online. We call the `datetime` function to create objects representing the start and end of the year. We need to put `year+1` here, if we do not do this, `month=12` and `day=31` would yield in nonsensical values around the new year (for example, 2017 when it is still 2016). If you search on Google for this problem, there are a lot of different, good answers and ways of doing this (some requiring additional packages installed):

```
def decyear(date):  
    start = datetime(year=date.year, month=1, day=1)  
    end = datetime(year=date.year+1, month=1, day=1)  
    decimal = (date-start)/(end-start)  
    return date.year+decimal
```

With this function, we can apply it to each element in the `datetime` column of our table. As all the column rows contain a `datetime` object already, the function that we just created happily accepts the input:

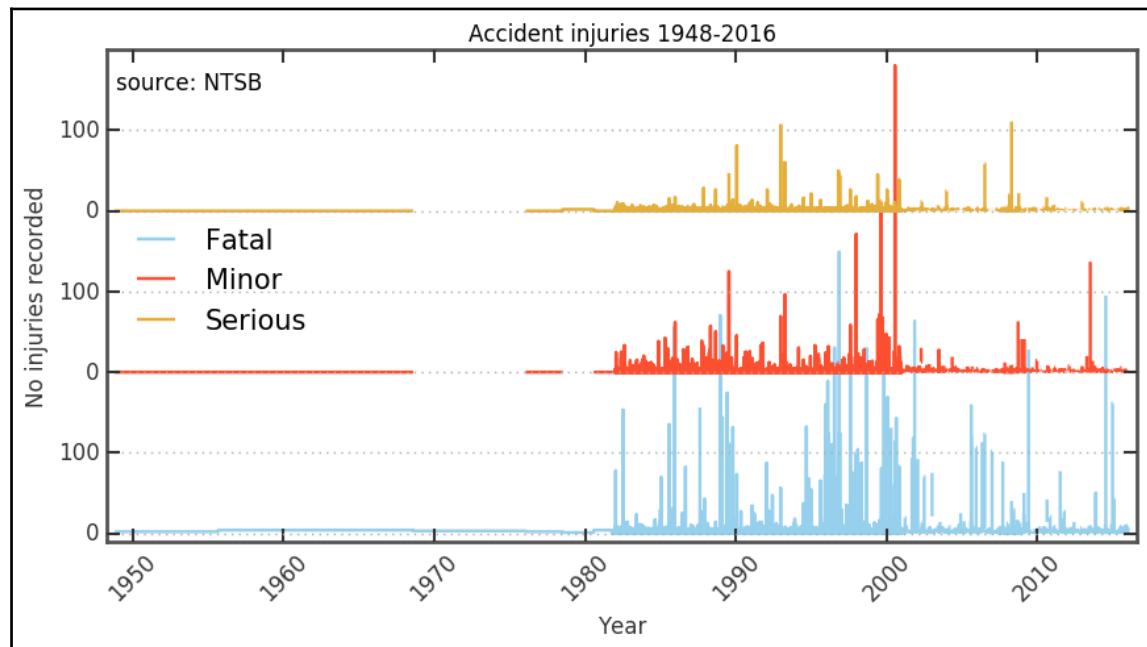
```
aadata['decyear'] = aadata['datetime'].apply(decyear)
```

Now, the columns `Latitude`, `Longitude`, `uninjured`, `fatalities`, and `serious` and `minor` injuries should all be floats and not strings for easy calculations and other operations. So we convert them to floats with the `applymap` method. The following code will convert the empty strings to `Nan` values and numbers to floats:

```
cols = ['lat', 'long',
        'fatal',
        'serious',
        'minor',
        'uninjured']
aadata[cols] = aadata[cols].applymap(
    lambda x: np.nan if isinstance(x, str)
               and x.isspace() else float(x))
```

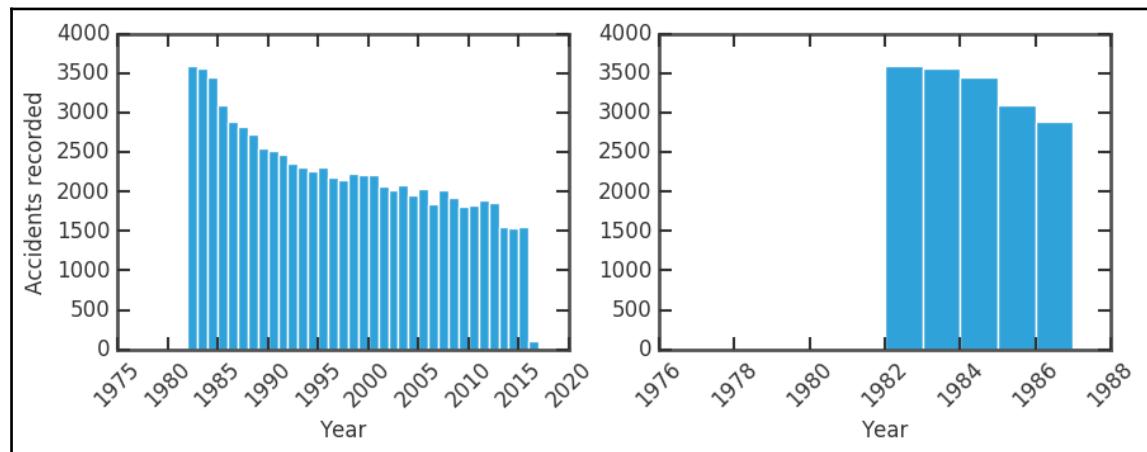
We are just applying a lambda function, which converts the values to floats and empty strings to `NaN`, to all the entries in the given columns. Now let's plot the data to see if we need to trim or process it further:

```
plt.figure(figsize=(9, 4.5))
plt.step(aadata['decyear'], aadata['fatal'],
          lw=1.75, where='mid', alpha=0.5, label='Fatal')
plt.step(aadata['decyear'], aadata['minor']+200,
          lw=1.75, where='mid', label='Minor')
plt.step(aadata['decyear'], aadata['serious']+200*2,
          lw=1.75, where='mid', label='Serious')
plt.xticks(rotation=45)
plt.legend(loc=(0.01,.4), fontsize=15)
plt.ylim((-10,600))
plt.grid(axis='y')
plt.title('Accident injuries {0}-{1}'.format(
            aadata['year'].min(), aadata['year'].max()))
plt.text(0.15,0.92,'source: NTSB', size=12,
        transform=plt.gca().transAxes, ha='right')
plt.yticks(np.arange(0,600,100), [0,100,0,100,0,100])
plt.xlabel('Year')
plt.ylabel('No injuries recorded')
plt.xlim((aadata['decyear'].min()-0.5,
          aadata['decyear'].max()+0.5));
```



The available data before around 1980 is very sparse and not ideal for statistical interpretation. Before deciding what to do, let's check the data for these entries. To see how many accidents were recorded, we plot the histogram around that time. Here, we employ a filter by combining two Boolean arrays. We also change the `bins` parameter by giving it the years, 1975 to 1990; in this way, we know that the bins will be per year:

```
plt.figure(figsize=(9, 3))
plt.subplot(121)
year_selection = (adata['year']>=1975) & (adata['year']<=2016)
plt.hist(adata[year_selection]['year'].as_matrix(),
         bins=np.arange(1975, 2016+2, 1), align='mid')
plt.xlabel('Year'); plt.grid(axis='x')
plt.xticks(rotation=45);
plt.ylabel('Accidents recorded')
plt.subplot(122)
year_selection = (adata['year']>=1976) & (adata['year']<=1986)
plt.hist(adata[year_selection]['year'].as_matrix(),
         bins=np.arange(1976, 1986+2, 1), align='mid')
plt.xlabel('Year')
plt.xticks(rotation=45);
```



The absolute number of accidents have roughly halved in 35 years; given that the number of passengers has definitely increased, this is very good. The right-hand plot shows that before 1983, there are very few recorded accidents by the NTSB. Listing the table with this criteria shows six recorded accidents:

```
aadata[aadata['year']<=1981]
```

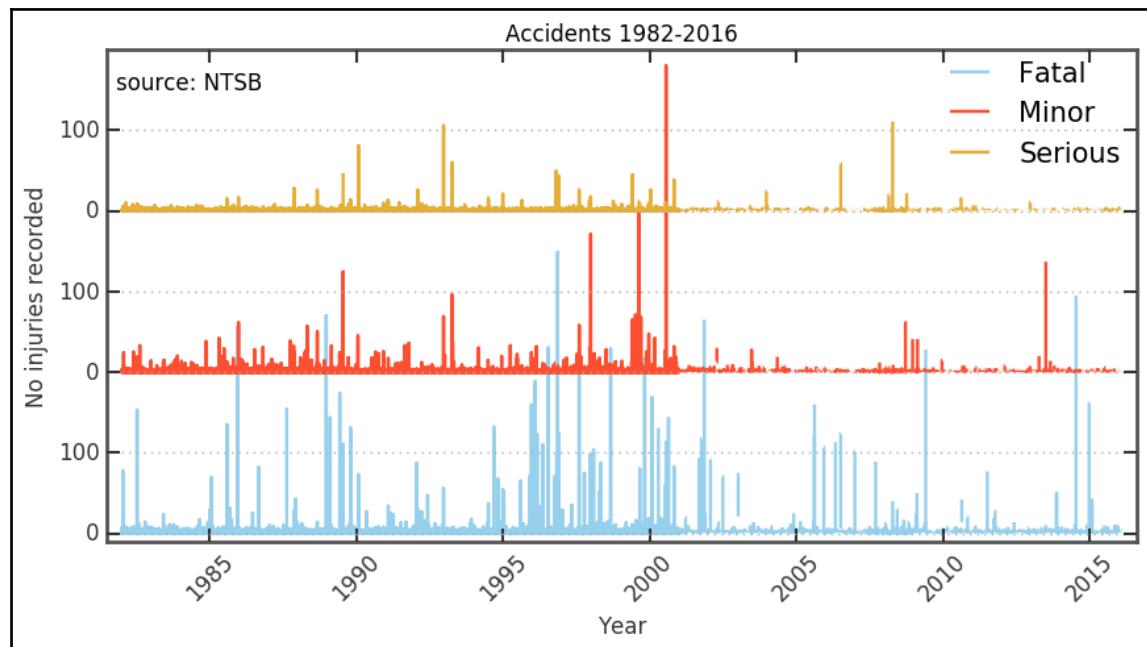
After this thorough inspection, I think it is safe to remove the entries before and including 1981. I do not know the reason for the lack of data; perhaps the NTSB was founded at this point? Their mandate was reformulated to include storing a database of the incidents? In any case, let's exclude these entries:

```
aadata = aadata[ aadata['year']>1981 ]
```

Creating the same figure as before, this is what we can see:

```
plt.figure(figsize=(10,5))
plt.step(aadata['decyear'], aadata['fatal'],
          lw=1.75, where='mid', alpha=0.5, label='Fatal')
plt.step(aadata['decyear'], aadata['minor']+200,
          lw=1.75, where='mid', label='Minor')
plt.step(aadata['decyear'], aadata['serious']+200*2,
          lw=1.75, where='mid', label='Serious')
plt.xticks(rotation=45)
plt.legend(loc=(0.8,0.74), fontsize=15)
plt.ylim((-10,6000))
plt.grid(axis='x')
plt.title('Accidents {0}-{1}'.format(
           aadata['year'].min(), aadata['year'].max()))
```

```
plt.text(0.135,0.95,'source: NTSB', size=12,
         transform=plt.gca().transAxes, ha='right')
plt.yticks(np.arange(0,600,100), [0,100,0,100,0,100])
plt.xlabel('Year')
plt.ylabel('No injuries recorded')
plt.xlim((adata['decyear'].min()-0.5,
          aadata['decyear'].max()+0.5));
```



We now have a cleaned dataset that we can work with. It is still hard to distinguish any trends, but around the millennium shift, there is a change in the trend. However, to further see what might be happening, we want to bin the data and look at various key numbers.

Binning the data

In this section, we will bin the data by year. This is so that we can get a better overview of overall trends in the data, which takes us to the next step in the characterization and analysis.

As mentioned before, we want to bin the data per year to look at the trends per year. We did binning before in Chapter 4, *Regression*; we use the `groupby` method of our Pandas DataFrame. There are two ways to define the bins here, we can use NumPy's `digitize` function or we can use Pandas `cut` function. I have used the `digitize` function here as it is more generally useful; you might not always use Pandas for your data (for some reason):

```
bins = np.arange(aadata.year.min(), aadata.year.max()+1, 1 )
yearly_dig = aadata.groupby(np.digitize(aadata.year, bins))
```

We can now calculate statistics for each bin. We can get the sum, maximum, mean, and so on:

```
yearly_dig.mean().head()
```

	lat	long	fatal	serious	minor	uninjured	month	year	decyear
1	30.757778	-88.355555	0.443978	0.203699	0.279474	2.317168	6.488450	1982	1982.495213
2	47.080556	-117.368611	0.358996	0.190059	0.296213	4.258810	6.652137	1983	1983.508864
3	NaN	NaN	0.356749	0.202322	0.303919	3.621739	6.553659	1984	1984.502076
4	NaN	NaN	0.534198	0.198379	0.359507	3.663855	6.477390	1985	1985.494267
5	NaN	NaN	0.410435	0.215454	0.338097	4.138531	6.480556	1986	1986.495181

Additionally, ensure that you get the years out:

```
np.floor(yearly_dig['year'].mean()).as_matrix()
```

```
array([ 1982.,  1983.,  1984.,  1985.,  1986.,  1987.,  1988.,  1989.,
       1990.,  1991.,  1992.,  1993.,  1994.,  1995.,  1996.,  1997.,
       1998.,  1999.,  2000.,  2001.,  2002.,  2003.,  2004.,  2005.,
       2006.,  2007.,  2008.,  2009.,  2010.,  2011.,  2012.,  2013.,
       2014.,  2015.,  2016.])
```

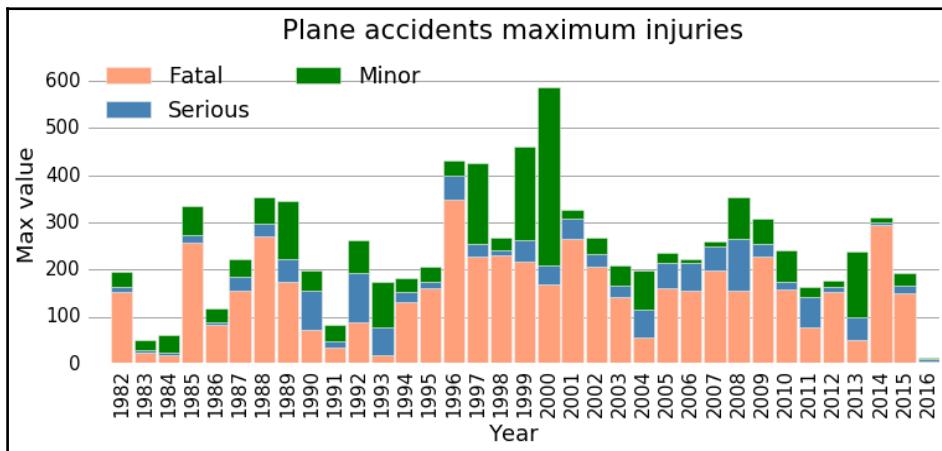
More importantly, we can visualize it. The following function will plot a bar plot and stack the various fields on top of each other. As input, it takes the Pandas groups object, a list of field names (< 3), and which field name to use as x axis. Then, there are some customizations and tweaks to make it look better. Noteworthy among them is the `fig.autofmt_xdate(rotation=90, ha='center')` function, which will format the dates for you automatically. You can send it various parameters; in this case, we use it to rotate and horizontally align the x tick labels (as dates):

```
def plot_trend(groups, fields=['Fatal'], which='year', what='max'):  
    fig, ax = plt.subplots(1,1,figsize=(9,3.5))  
    x = np.floor(groups.mean()[which.lower()]).as_matrix()  
    width = 0.9  
    colors = ['LightSalmon', 'SteelBlue', 'Green']  
    bottom = np.zeros( len(groups.max())[fields[0].lower()] )  
    for i in range(len(fields)):  
        if what=='max':  
            ax.bar(x, groups.max()[fields[int(i)].lower()],  
                    width, color=colors[int(i)],  
                    label=fields[int(i)], align='center',  
                    bottom=bottom, zorder=4)  
        bottom += groups.max()[  
            fields[int(i)].lower()  
        ].as_matrix()  
    elif what=='mean':  
        ax.bar(x, groups.mean()[fields[int(i)].lower()],  
                width, color=colors[int(i)],  
                label=fields[int(i)],  
                align='center', bottom=bottom, zorder=4)  
        bottom += groups.mean()[  
            fields[int(i)].lower()  
        ].as_matrix()  
    ax.legend(loc=2, ncol=2, frameon=False)  
    ax.grid(b=True, which='major',  
            axis='y', color='0.65', linestyle='-', zorder=-1)  
    ax.yaxis.set_ticks_position('left')  
    ax.xaxis.set_ticks_position('bottom')  
    for tic1, tic2 in zip(  
        ax.xaxis.get_major_ticks(),  
        ax.yaxis.get_major_ticks()  
    ):  
        tic1.tick1On = tic1.tick2On = False  
        tic2.tick1On = tic2.tick2On = False  
    for spine in ['left','right','top','bottom']:  
        ax.spines[spine].set_color('w')  
    xticks = np.arange(x.min(), x.max()+1, 1)  
    ax.set_xticks(xticks)  
    ax.set_xticklabels([str(int(x)) for x in xticks])
```

```
fig.autofmt_xdate(rotation=90, ha='center')
ax.set_xlim((xticks.min()-1.5, xticks.max()+0.5))
ax.set_ylim((0,bottom.max()*1.15))
if what=='max':
    ax.set_title('Plane accidents maximum injuries')
    ax.set_ylabel('Max value')
elif what=='mean':
    ax.set_title('Plane accidents mean injuries')
    ax.set_ylabel('Mean value')
ax.set_xlabel(str(which))
return ax
```

Now, let's use it to plot the maximum fatal, serious, and minor injuries for each year in the timespan, 1982 to 2016:

```
ax = plot_trend(yearly_dig, fields=['Fatal','Serious','Minor'],
                 which='Year')
```



The dominant bars are the ones for fatal—at least from this visual inspection—so when things go really bad in an airplane accident, most people contract fatal injuries. What is curious here is that there seems to be a change in the maximum value, a jump somewhere between 1991 and 1996. After that, it looks like the mean of the maximum number of fatalities is higher. This is what we will try to model with Bayesian inference in the next section.

Bayesian analysis of the data

Now we can dive into the Bayesian part of this analysis. You should always inspect the data in the manner that we have done in the previous exercises. These steps are commonly skipped in text and even fabricated data is used; this paints a simplified picture of data analysis. We have to work with the data to get somewhere and understand what analysis is feasible. First, we need to import the PyMC package and plot function from the Matplot submodule. This function plots a summary of the parameters' posterior distribution, trace (that is, each iteration), and autocorrelation:

```
import pymc
from pymc import Matplot as mcplt
```

To start the analysis, we store the x and y values, year, and maximum fatalities in arrays:

```
x = np.floor(yearly_dig.mean()['year']).as_matrix()
y = yearly_dig.max()['fatal'].as_matrix()
```

Now we develop our model by defining a function with all the parameters and data. It is a discrete process, so we use a Poisson distribution. Furthermore, we use the exponential distribution for the early and late mean rate; this is appropriate for this stochastic process. Try plotting a histogram of the fatalities to see what distribution it has. For the year where the jump/switch takes place, we use a discrete uniform distribution, that is, it is flat for all the values between the lower and upper bounds and zero elsewhere. The variable that is determined by the various stochastic variables, late, early mean, and the switch point is the mean value before and after the jump. As it depends on (stochastic) variables, it is called a deterministic variable. In the code, this is marked by the `@pymc.deterministic()` decorator. This is the process that we are trying to model. There are several distributions built into PyMC, but you can also define your own. However, for most problems, the built-in ones should do the trick. The various available distributions are in the `pymc.distributions` submodule:

```
def model_fatalities(y=y):
    s = pymc.DiscreteUniform('s', lower=5, upper=18, value=14)
    e = pymc.Exponential('e', beta=1.)
    l = pymc.Exponential('l', beta=1.)
    @pymc.deterministic(plot=False)
    def m(s=s, e=e, l=l):
        meanval = np.empty(len(y))
        meanval[:s] = e
        meanval[s:] = l
        return meanval
    D = pymc.Poisson('D', mu=m, value=y, observed=True)
    return locals()
```

The `return locals()` is a somewhat simply way to send all the local variables back. As we have a good overview of what those are, this is not a problem to use. We have now defined the model; to use it in the MCMC sampler, we give the mode as input to the MCMC class:

```
np.random.seed(1234)
MDL = pymc.MCMC(model_fatalities(y=y))
```

To use the standard sampler, we can simply call the `MDL.sample(N)` method, where `N` is the number of iterations to run. There are additional parameters as well; you can give it a burn-in period, a period where no results are considered. This is part of the MCMC algorithm, where it can sometimes be good to let it run for a few iterations that are discarded so that it can start converging. Second, we can give a `thin` argument; this is how often it should save the outcome of the iteration. In our case, I run it 50,000 times, with 5,000 iterations as burn-in and thin by two. Try running with various numbers to see if the outcome changes, how it changes, and how well the parameters are estimated:

```
MDL.sample(5e4, 5e3, 2)
```

```
[-----100%-----] 50000 of 50000 complete in 8.6 sec
```

The step method, that is, how to move around in the parameter space, can also be changed. To check what step method we have, we run the following command:

```
MDL.step_method_dict
```

```
{<pymc.distributions.new_dist_class.<locals>.new_class 'l' at 0x7f841e56b470>: [<pymc.StepMethods.Metropolis at 0x7f841e56b390>],
 <pymc.distributions.new_dist_class.<locals>.new_class 'e' at 0x7f841e56b358>: [<pymc.StepMethods.Metropolis at 0x7f841e56b208>],
 <pymc.distributions.new_dist_class.<locals>.new_class 's' at 0x7f841e56b3c8>: [<pymc.StepMethods.DiscreteMetropolis at 0x7f841e56b2e8>]}
```

To change the step method, perhaps to the Adaptive Metropolis algorithm, which uses an adaptive step size (length), we would import it and run the following:

```
from pymc import AdaptiveMetropolis
MDL = pymc.MCMC(model_fatalities(y=y))
MDL.use_step_method(AdaptiveMetropolis, MDL.e)
MDL.use_step_method(AdaptiveMetropolis, MDL.l)
MDL.sample(5e4, 5e3, 2)
```

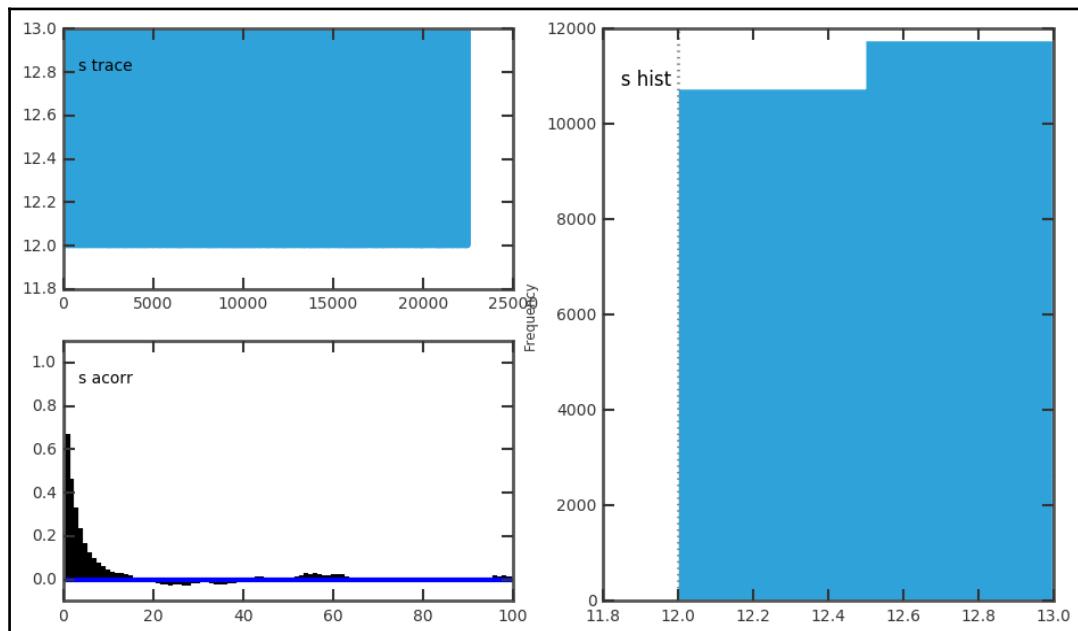
We are not going to do this here though; this is for problems where the variables are highly correlated. I leave this as an exercise for you to test, together with different prior parameter distributions.

Now we have the whole run in the `MDL` object. From this object, we can estimate the parameters and plot their posterior distribution. There are convenient functions to do all of this. The following code shows you how to pull out the mean of the posterior distribution and standard deviation. This is where credible interval comes in; we have credible intervals for the parameters, not confidence intervals:

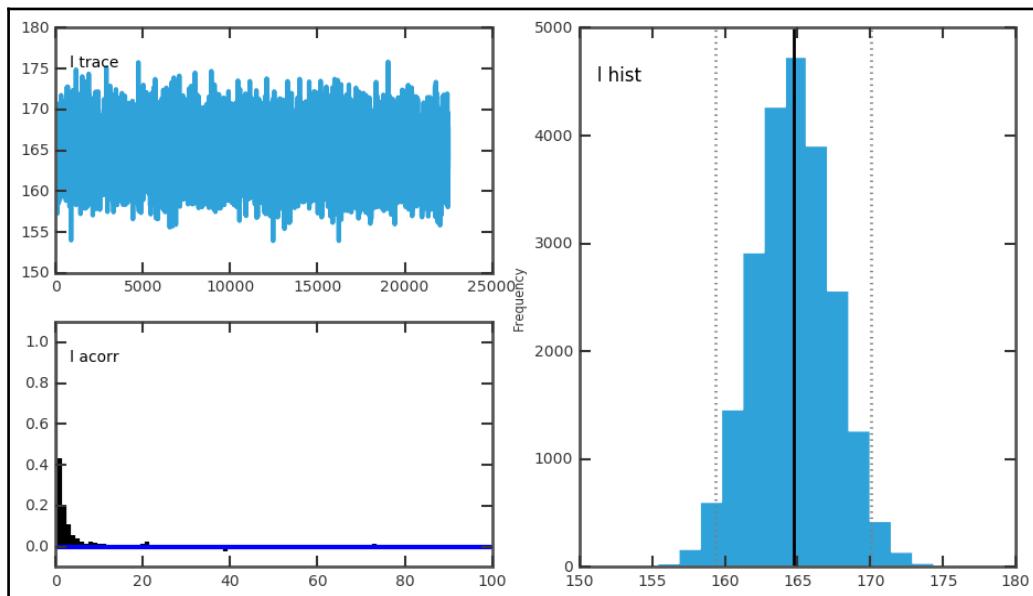
```
early = MDL.stats()['e']['mean']
earlyerr = MDL.stats()['e']['standard deviation']
late = MDL.stats()['l']['mean']
lateerr = MDL.stats()['l']['standard deviation']
spt = MDL.stats()['s']['mean']
spterr = MDL.stats()['s']['standard deviation']
```

Before plotting the results and all the numbers, we must check the results of the MCMC run, to do this we plot the trace, posterior distribution, and autocorrelation for all the stochastic parameters. We do this with the `plot` function from the `pymc.Matplot` module that we imported in the beginning:

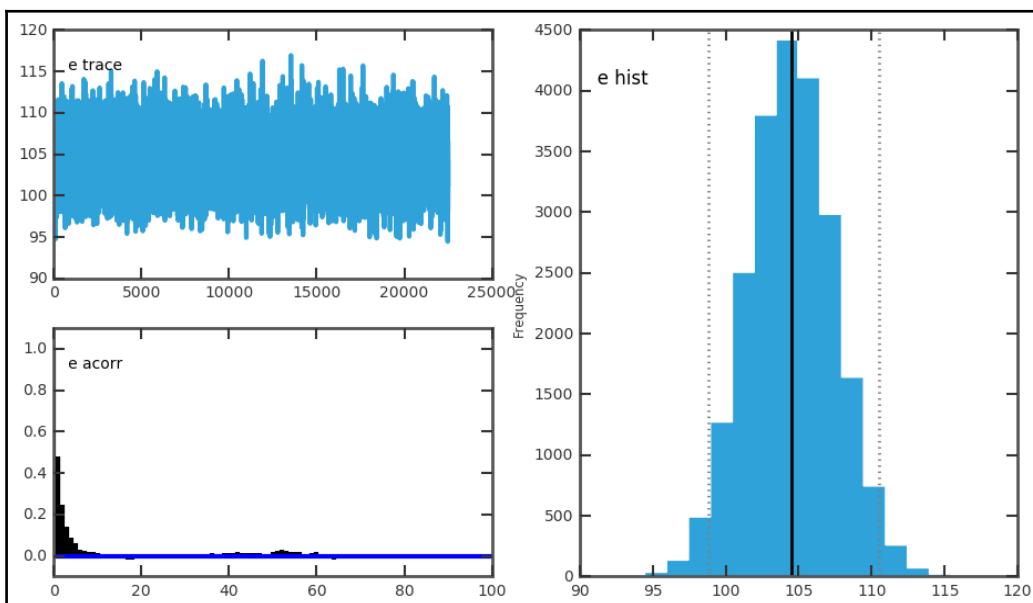
```
mcplt.plot(MDL)
```



The function plots all three things in one figure. This is very convenient for a quick assessment of the results. It is important to look at all the plots; they give clues to how well the run went:

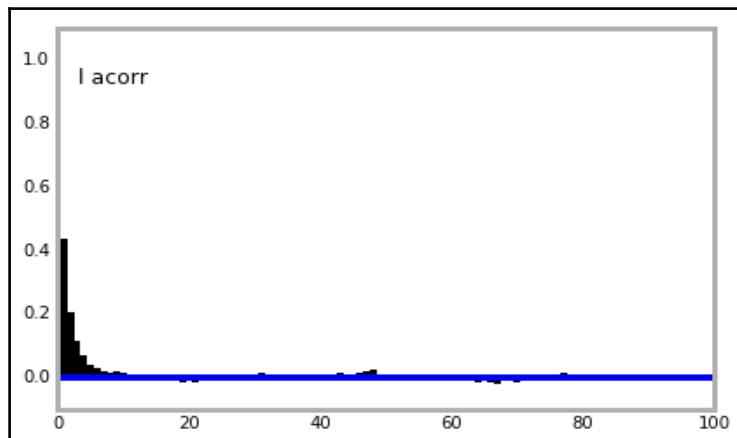


For each stochastic variable, the trace, autocorrelation, and posterior distribution is plotted:



In our case, this gives one figure each for `e`, `l`, and `s`. In each figure, the top left-hand plot shows the trace or time series. The trace is the value for each iteration. It can be accessed with `MDL.trace('l')[:]` for the `late` parameter. Try getting the trace and plotting the trace versus iteration and histogram for it; they should look the same as these. For a good model setup, the trace should fluctuate randomly around the best estimate, just like the trace for the early and `later` parameter. The autocorrelation plot should have a peak at 0; if the plot shows a significant amount of values at higher `x` values, it is a sign that you need to increase the `thin` variable. The trace and posterior distribution for the jump/switch point looks different. However, because it is a discrete variable and is constrained within one year, it just shows one peak and the trace follows this. Thus, the switch point is well-constrained with a very small credible interval. Each of the plots in the figures can be drawn individually with related functions in the `pymc.Matplot` module (`mcpplt` here). The autocorrelation plot for the `l` variable can be produced with the following command:

```
mcpplt.autocorrelation(MDL.l)
```



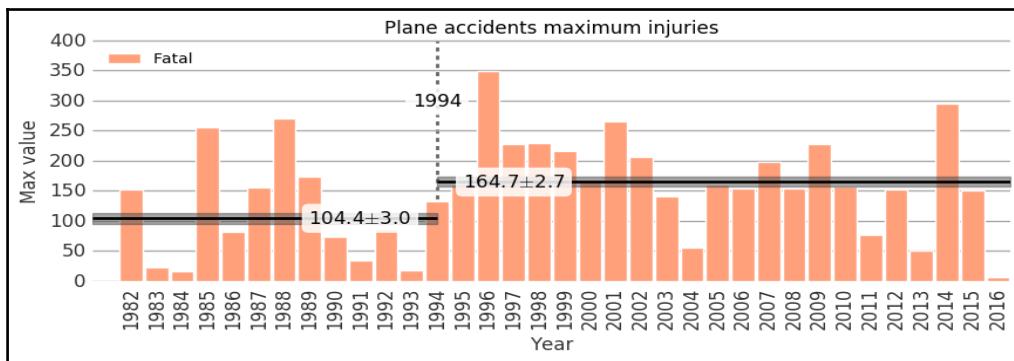
Now that we have constructed a model, run the sampler, and extracted the best estimate parameters, we can plot the results. The jump/switch point identified by the model is simply an index in the array of years, so we need to find which year the index/position corresponds to. To do this, we use NumPy's `floor` function to round down, and then we can convert it to an integer and slice the `x` array, which is our year array created in the beginning:

```
s = int(np.floor(spt))
print(spt, spterr, x[s])
```

The preceding code gives 12.524, 0.499423667841, and 1994.0 as an output.

To construct the plot with the results, we can once again use the function defined previously, but this time we are only looking at the fatal injuries, so we give a different field parameter. To plot the credible intervals, I use the `fill_between` function here; it is a very handy function and does exactly what it says. Furthermore, instead of the `text` function, which just places text in the figure, I use the more powerful `annotate` function, where we can customize it with a nice box around it:

```
ax = plot_trend(yearly_dig, fields=['Fatal'], which='Year')
ax.plot([x[0]-1.5,x[s]], [early,early], 'k', lw=2)
ax.fill_between([x[0]-1.5,x[s]],
                [early-3*earlyerr,early-3*earlyerr],
                [early+3*earlyerr,early+3*earlyerr],
                color='0.3', alpha=0.5, zorder=2)
ax.plot([x[s],x[-1]+0.5], [late,late], 'k', lw=2)
ax.fill_between([x[s],x[-1]+0.5],
                [late-3*lateerr,late-3*lateerr],
                [late+3*lateerr,late+3*lateerr],
                color='0.3', alpha=0.5, zorder=2)
ax.axvline(int(x[s]), color='0.4', dashes=(3,3), lw=2)
bbox_args = dict(boxstyle="round", fc="w", alpha=0.85)
ax.annotate('{0:.1f}±{1:.1f}'.format(early, earlyerr),
            xy=(x[s]-1,early),
            bbox=bbox_args, ha='right', va='center')
ax.annotate('{0:.1f}±{1:.1f}'.format(late, lateerr),
            xy=(x[s]+1,late),
            bbox=bbox_args, ha='left', va='center')
ax.annotate('{0}'.format(int(x[s])), xy=(int(x[s]),300),
            bbox=bbox_args, ha='center', va='center');
```



Given the data, the parameters are in the ranges, 104.5 ± 3.0 and 164.7 ± 2.7 , with a 95% credibility (that is, ± 1 sigma). The mean maximum number of fatalities in a year have increased and made a jump around 1994 from 104.5 to 164.7 people. Although from the plot it looks around the year 2004 and 2012-2013, the maximum values do not follow the same trend. Even if this is a simple model, it is hard to argue for a more complex model and the results are significant. The goal should, of course, be to have zero fatalities for every year.

We analyzed what the statistics look like on a per year basis; what if we look at the statistics over the year? To do this, we need to bin per month. So in the next section, this is what we will do.

Binning by month

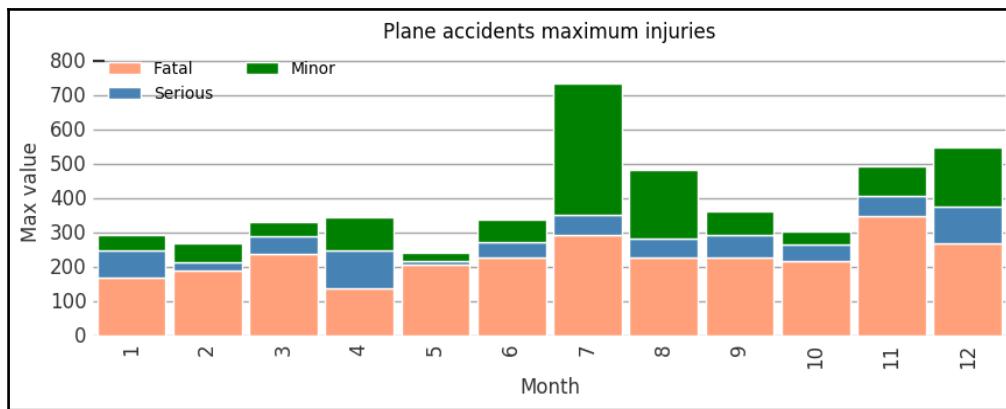
Repeat the binning procedure that we have, but by month:

```
bins = np.arange(1, 12+1, 1 )
monthly_dig = aadata.groupby(np.digitize(aadata.month, bins))
monthly_dig.mean().head()
```

	lat	long	fatal	serious	minor	uninjured	month	year	decyear
1	35.355070	-91.799283	1.024984	0.351452	0.482369	7.505215	1	1996.595841	1996.637186
2	35.211711	-92.921433	0.890277	0.295262	0.486586	6.861646	2	1996.190641	1996.313377
3	36.439443	-93.719725	0.673851	0.281289	0.428362	6.491056	3	1996.299847	1996.503531
4	37.187038	-93.330245	0.651183	0.304870	0.489868	5.455664	4	1996.292600	1996.579883
5	37.816921	-94.954229	0.671199	0.272957	0.445699	5.194848	5	1996.382683	1996.754983

Now we can do the same but per month; just send the right parameters to the `plot_trend` function that we have created:

```
ax = plot_trend(monthly_dig, fields=['Fatal', 'Serious', 'Minor'],
                 which='Month')
ax.set_xlim(0.5,12.5);
```



While there is no strong trend, we can note that months 7 and 8, July and August, together with months 11 and 12, November and December, have higher values. Summer and Christmas are popular times of the year to travel, so it is probably a reflection of the yearly variation in the amount of travelers. More travelers means an increase in the number of accidents (with constant risk), thus a greater chance for high fatality accidents. Another question is what the mean variation looks like; what we have worked with so far is the maximum. I have added a parameter in the plotting function, `what`, and set it to `mean`, which will cause the mean to be plotted. I leave this as an exercise; you will see something peculiar with the mean. Try to create different plots to investigate! Also, do not forget to check the mean per month and year.

This last plot highlights something else as well—the total number of passengers might affect the outcome. To get the total number of passengers in the U.S., you can run the following:

```
from pandas.io import wb
airpasstot = wb.download(indicator='IS.AIR.PSGR',
                         country=['USA'], start=1982, end=2014)
```

However, looking at this data and comparing it with the data that we have worked with here is left as an exercise for you.

To give you a taste of some of the powerful visualizations that are possible with Python, I want to quickly plot the coordinates of each accident on a map. This is of course not necessary in this case, but it is good to visualize results sometimes, and in Python, it is not obvious how to do it. So this part is not necessary for this chapter, but it is good to know the basics of how to plot things on a map in Python, as many things that we study depend on location on the Earth.

Plotting coordinates

To plot the coordinates, we (unfortunately) have to install packages. Even more unfortunate is that it depends on what operating system and Python distribution you are running. The two packages that we will quickly cover here are the basemap module of the mpl_toolkits package and the cartopy package.

The first one, mpl_toolkits, does not work on Windows (as of April 2016). To install basemap from mpl_toolkits in Anaconda on Mac and Linux, run `conda install -c https://conda.anaconda.org/anaconda basemap`. This should install basemap and all its dependencies.

The second package, cartopy, depends on the GEOS and proj.4 libraries, so they need to be installed first. This can be a bit tedious, but once the GEOS (version greater than 3.3.3) and proj.4 libraries (version greater than 4.8.0) are installed, cartopy can be installed with the pip command-line tool, `pip install cartopy`. Once again, in Windows, the prebuilt binaries of proj.4 is version 4.4.6, making it very difficult to install cartopy as well.

For this quick exercise, we grab the latitudes and longitudes of each accident in separate arrays as the plotting commands might be sensitive with the input format and might not support the Pandas Series:

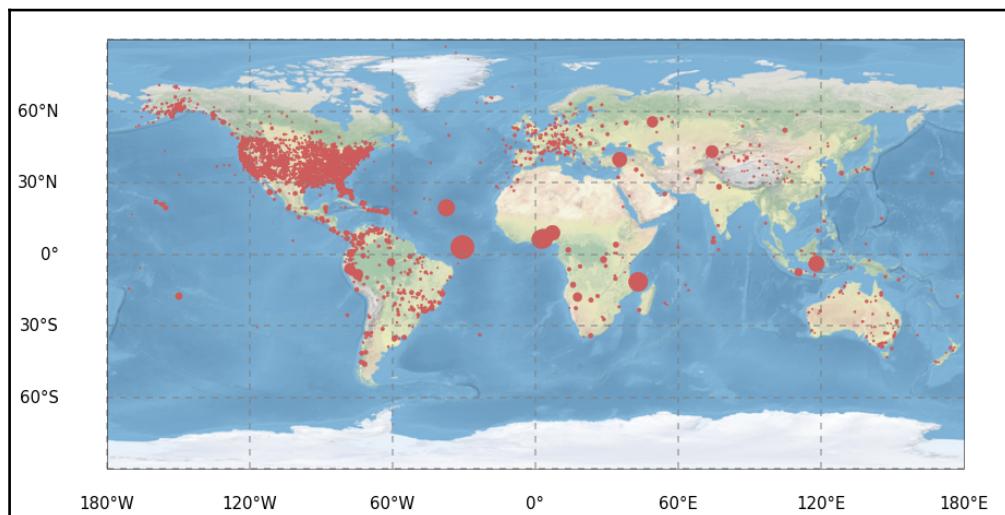
```
lats, lons = aadata['lat'].as_matrix(), aadata['long'].as_matrix()
```

Cartopy

First out is cartopy, where we first create a figure, then add axes to it, where we have to specify the lower left and top right edges of the axes inside the figure and additionally, give it a projection, which is taken from the CRS module of cartopy. When importing the CRS module, we also import the formatters for longitude and latitude, which will just add N, S, E, and W to the x and y tick labels. Importing the matplotlib ticker module, we can specify the exact locations of the tick labels. We could probably do this in the same manner with the ticks function.

As we are plotting the coordinates of the accidents, it is nice to have a background showing the Earth. Therefore, we load an image of the Earth with the `ax.stock_img()` command. It is possible to load coastlines, countries, and other things. To view examples and other possibilities including different projections, see the cartopy website (<http://scitools.org.uk/cartopy>). We then create a scatter plot with latitude and longitude as coordinates and scale the size of the markers proportional to the total fatalities. Then, we plot the gridlines, meridians, and latitude great circles. After this, we just customize the tick locations and labels with the imported formatters and tick location setters:

```
import cartopy.crs as ccrs
from cartopy.mpl.gridliner import LONGITUDE_FORMATTER, LATITUDE_FORMATTER
import matplotlib.ticker as mticker
fig = plt.figure(figsize=(12,10))
ax = fig.add_axes([0,0,1,1], projection=ccrs.PlateCarree())
ax.stock_img()
ax.scatter(aadata['long'],aadata['lat'],
           color='IndianRed', s=aadata['fatal']*2,
           transform=ccrs.Geodetic())
gl = ax.gridlines(crs=ccrs.PlateCarree(), draw_labels=True,
                   linewidth=2, color='gray',
                   alpha=0.5, linestyle='--')
gl.xlabel_top = False
gl.ylabel_right = False
gl.xlocator = mticker.FixedLocator(np.arange(-180,180+1,60))
gl.xformatter = LONGITUDE_FORMATTER
gl.yformatter = LATITUDE_FORMATTER
```

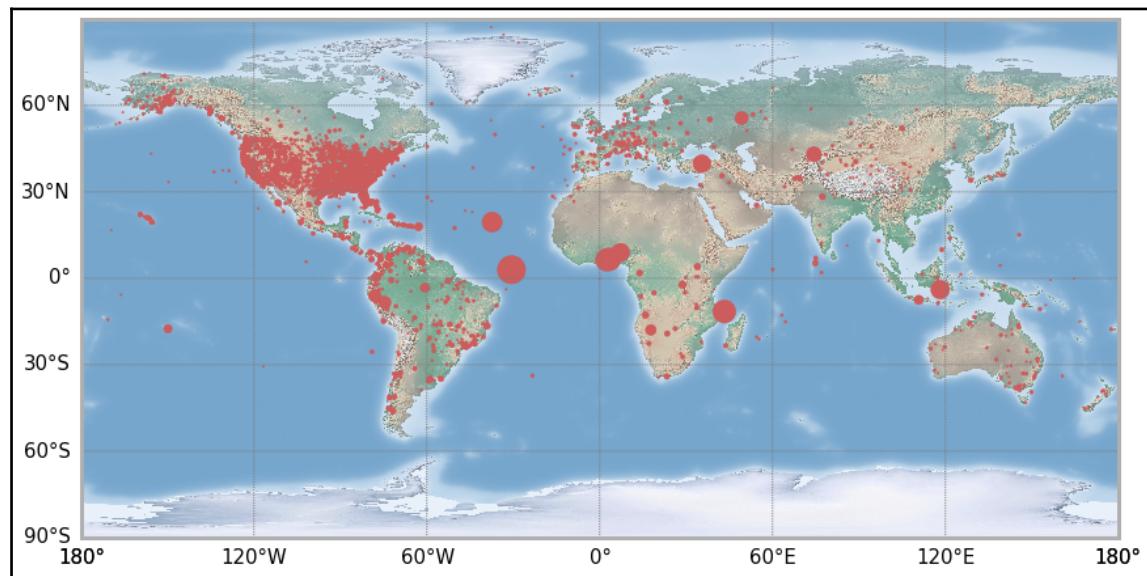


It seems that most of the registered accidents occurred over land. I leave the further tinkering of the figure to you. Looking at statistics over the world, it is useful to be able to plot the spatial distribution of the parameters.

Mpl toolkits – basemap

As promised, we now produce the exact same plot in the basemap module of mpl_toolkits. Here, we only need to import the basemap module and no other tick-modifying functions. The projection is not set in creating the axes, instead it is created by calling the basemap function with the wanted projection. Previously, we called the Carree projection; this is simply the Equidistant Cylindrical projection, and in basemap, this is obtained by giving it the projection string, cyl, for cylindrical. The resolution parameter is given as c, for coarse. To get the same background image, call the shadedrelief command. There are other backgrounds, such as Earth by night, for example. It is possible to draw only coastlines or country borders. In basemap, there are built-in functions for a lot of things; thus, instead of calling matplotlib functions, we now call methods of the map object to create meridians and latitudes. I have also included the functions to draw coastlines and country borders, but commented them out. Try uncommenting them and perhaps comment out the drawing of the background image:

```
from mpl_toolkits.basemap import Basemap
fig = plt.figure(figsize=(11,10))
ax = fig.add_axes([0,0,1,1])
map = Basemap(projection='cyl', resolution='c')
map.shadedrelief()
#map.drawcoastlines()
#map.drawcountries()
map.drawparallels(np.arange(-90, 90, 30), labels=[1,0,0,0],
                  color='grey')
map.drawmeridians(np.arange(map.lonmin, map.lonmax+30, 60),
                  labels=[0,0,0,1], color='grey')
x, y = map(lons, lats)
map.scatter(x, y, color='IndianRed', s=aadata['fatal']*2);
```



I leave any further tinkering of the coordinate plotting to you. However, you have the first steps here. As the main goal of this chapter is Bayesian analysis, we shall now continue with the data analysis examples.

Climate change – CO₂ in the atmosphere

With Bayesian analysis, we can fit any model; anything that we can do with frequentist or classical statistics, we can do with Bayesian statistics. In this next example, we will perform linear regression with both Bayesian inference and frequentist approaches. As we have covered the model creation and date parsing, we will go through things a little bit more quickly in this example. The data that we are going to use is the atmospheric CO₂ over a span of about 1,000 years and the growth rate over the past 40 years, and then fit a linear function to the growth rate over the past 50-60 years.

Getting the data

The data for the last 50-60 years is from **National Oceanic and Atmospheric Administration (NOAA)** marine stations, **surface sites**. It can be found at <http://www.esrl.noaa.gov/gmd/ccgg/trends/global.html>, where you can download two datasets, growth rates, and annual means. The direct links to the data tables are ftp://aftp.cmdl.noaa.gov/products/trends/co2/co2_gr_gl.txt for the growth rates (that is, gr) and ftp://aftp.cmdl.noaa.gov/products/trends/co2/co2_annmean_gl.txt for the global means (annual in this case). The data reference is Ed Dlugokencky and Pieter Tans, NOAA/ESRL (<http://www.esrl.noaa.gov/gmd/ccgg/trends/>).

To go further back, we need ice core samples from the South Pole, the SIPLE station ice core that goes about 200 years into the past. At <http://cdiac.ornl.gov/trends/co2/siple.html>, there is more information and a direct link to the data, <http://cdiac.ornl.gov/ftp/trends/co2/siple2.013>. The data reference is *Neftel,A., H. Friedli, E. Moor, H. Lütscher, H. Oeschger, U. Siegenthaler, and B. Stauffer, 1994. Historical CO₂ record from the Siple Station ice core. In Trends: A Compendium of Data on Global Change. Carbon Dioxide Information Analysis Center, Oak Ridge National Laboratory, U.S. Department of Energy, Oak Ridge, Tenn., U.S.A.* For data even further back, a 1,000 years, we use ice cores from the Law Dome; more information can be found at <http://cdiac.ornl.gov/trends/co2/lawdome.html> with a direct link to the data, <http://cdiac.ornl.gov/ftp/trends/co2/lawdome.smoothed.yr75>, and the reference is *D.M. Etheridge, L.P. Steele, R.L. Langenfelds, R.J. Francey, J.-M. Barnola, and V.I. Morgan, 1998. Historical CO₂ records from the Law Dome DE08, DE08-2, and DSS ice cores. In Trends: A Compendium of Data on Global Change. Carbon Dioxide Information Analysis Center, Oak Ridge National Laboratory, U.S. Department of Energy, Oak Ridge, Tenn., U.S.A.*

As we have done a few times now, we read in the data with the Pandas csv reader:

```
co2_gr = pd.read_csv('data/co2_gr_gl.txt',
                     delim_whitespace=True,
                     skiprows=62,
                     names=['year', 'rate', 'err'])
co2_now = pd.read_csv('data/co2_annmean_gl.txt',
                      delim_whitespace=True,
                      skiprows=57,
                      names=['year', 'co2', 'err'])
co2_200 = pd.read_csv('data/siple2.013.dat',
                      delim_whitespace=True,
                      skiprows=36,
                      names=['depth', 'year', 'co2'])
co2_1000 = pd.read_csv('data/lawdome.smoothed.yr75.dat',
                       delim_whitespace=True,
```

```
skiprows=22,  
names=['year', 'co2'])
```

There are some additional comments in the last rows of the SIPLE ice core file:

```
co2_200.tail()
```

	depth	year	co2
20	86.80	1943	307.9
21	81.22	1953	312.7
22	Data	in	the
23	table	were	published
24	CO2	concentrations	are

We remove them by slicing the data frame excluding the last three rows:

```
co2_200 = co2_200[:-3]
```

As the Pandas csv reader could not parse the last three rows into floats/integers, the `dtype` is not right; as it also reads in text, it will use the most generic and accepting data type. First, check the data type of all the datasets to see that we do not have to fix any of the others:

```
print( co2_200['year'].dtype, co2_1000['co2'].dtype,  
      co2_now['co2'].dtype, co2_gr['rate'].dtype)
```

```
object float64 float64 float64
```

The `co2_200` DataFrame has the wrong `dtype`, as expected. We change it with Pandas' `to_numeric` function and check whether it works:

```
co2_200['year'] = pd.to_numeric(co2_200['year'])  
co2_200['co2'] = pd.to_numeric(co2_200['co2'])  
co2_200['co2'].dtype, co2_200['year'].dtype
```

```
(dtype('float64'), dtype('int64'))
```

64-bit integer and float is now the new data type of the year and `co2` columns respectively, which is exactly what we need.

Creating and sampling the model

Now let's visualize all of this. The dataset can be separated in two, as explained previously—one for the absolute CO₂ concentration and one for the growth rate. The unit of the CO₂ concentration is expressed as a mole fraction in dry air (the South Pole is one of the driest places on Earth); in this case, the fraction is expressed in parts per million:

```
fig, axs = plt.subplots(1,2, figsize=(10,4))

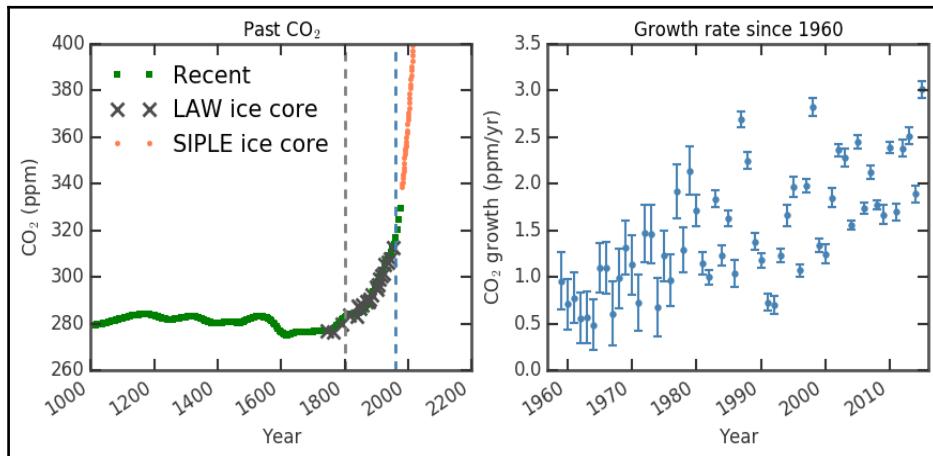
ax2 = axs[0]
ax2.errorbar(co2_now['year'], co2_now['co2'],
             yerr=co2_now['err'],
             color='SteelBlue',
             ls='None',
             elinewidth=1.5,
             capthick=1.5,
             marker='.',
             ms=6)
ax2.plot(co2_1000['year'], co2_1000['co2'],
          color='Green',
          ls='None',
          marker='.',
          ms=6)
ax2.plot(co2_200['year'], co2_200['co2'],
          color='IndianRed',
          ls='None',
          marker='.',
          ms=6)
ax2.axvline(1800, lw=2, color='Gray', dashes=(6,5))
ax2.axvline(co2_gr['year'][0], lw=2,
            color='SteelBlue', dashes=(6,5))
print(co2_gr['year'][0])
ax2.legend(['Recent',
            'LAW ice core',
            'SIPPLE ice core'], fontsize=15, loc=2)
labels = ax2.get_xticklabels()
plt.setp(labels, rotation=33, ha='right')
ax2.set_ylabel('CO$_2$ (ppm)')
ax2.set_xlabel('Year')
ax2.set_title('Past CO$_2$')

ax1 = axs[1]
ax1.errorbar(co2_gr['year'], co2_gr['rate'],
             yerr=co2_gr['err'],
             color='SteelBlue',
             ls='None',
             elinewidth=1.5,
```

```

capthick=1.5,
marker='.',
ms=8)
labels = ax1.get_xticklabels()
plt.setp(labels, rotation=33, ha='right')
ax1.set_ylabel('CO2 growth (ppm/yr)')
ax1.set_xlabel('Year')
ax1.set_xlim((1957,2016))
ax1.set_title('Growth rate since 1960');

```

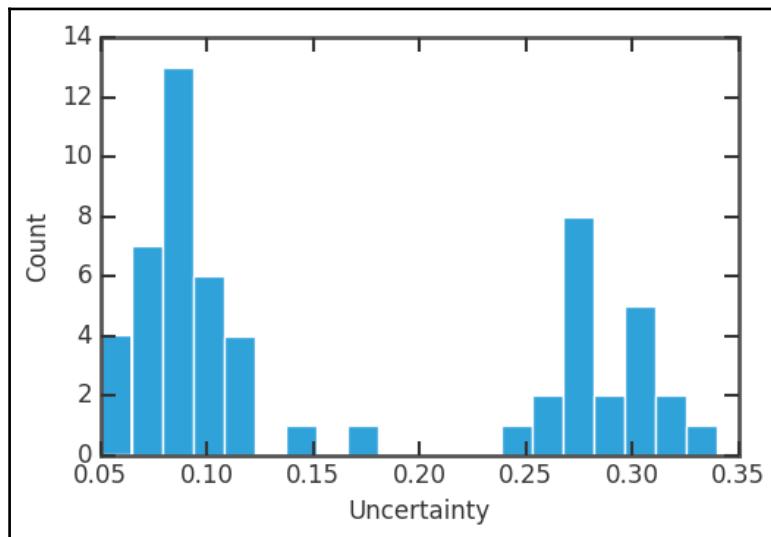


The left-hand plot showing the absolute CO₂ level shows that the ice cores connect to the present day measurements very well. The first vertical dashed line marks the rough start of the industrial revolution (the year 1800). Even though it took some 50 years to really get the steam going (pun intended), this shows where the curve starts an exponential increase that has continued to this day. This correlation between the sharp changes from rather stable measurements to an exponential increase precisely after introducing coal burning steam engines is a strong evidence for manmade climate change. The second vertical line shows where the modern, direct measurements start, that is, 1959. They fit very well with the historical record extracted from ice cores. In the right-hand plot, we basically have a zoom in of that modern time period, from 1959 until today; however, it shows the growth of CO₂ in the atmosphere in ppm (as covered earlier). The data seems to have some distribution in the uncertainties (signifying an upgrade in the measuring technique/instrument). Out of curiosity, let's check this first:

```

_ = plt.hist(co2_gr['err'], bins=20)
plt.xlabel('Uncertainty')
plt.ylabel('Count');

```



Indeed, it has two peaks where the oldest values are the most uncertain. Just like our previous example, let's first convert the values that we want to NumPy arrays:

```
x = co2_gr['year'].as_matrix()
y = co2_gr['rate'].as_matrix()
y_error = co2_gr['err'].as_matrix()
```

Now that we have done this, we define our model of a linear slope with the same method as for the airplane accidents. It is simply a function that returns the stochastic and deterministic variables. In our case, it is a linear function, taking slope and intercept; this time, we assume that they are normally distributed, which is not an unreasonable assumption. The normal distribution takes a minimum of two parameters, `mu` and `tau` (from PyMC documentation), which is just the position and width of the Gaussian normal distribution:

```
def model(x, y):
    slope = pymc.Normal('slope', 0.1, 1.)
    intercept = pymc.Normal('intercept', -50., 10.)
    @pymc.deterministic(plot=False)
    def linear(x=x, slope=slope, intercept=intercept):
        return x * slope + intercept
    f = pymc.Normal('f', mu=linear,
                    tau=1.0/y_error, value=y, observed=True)
    return locals()
```

As before, we initiate the model with a call to MCMC and then sample from it half a million times, a burn-in of 50,000 and thin by 100 this time. I encourage you to first run with something low, such as four, or even omit it (that is, `MDL.sample(5e5, 5e4)`), plot the diagnostics (as follows), and compare the results:

```
MDL = pymc.MCMC(model(x,y))
MDL.sample(5e5, 5e4, 100)
```

```
[-----100%-----] 500000 of 500000 complete in 41.5 sec
```

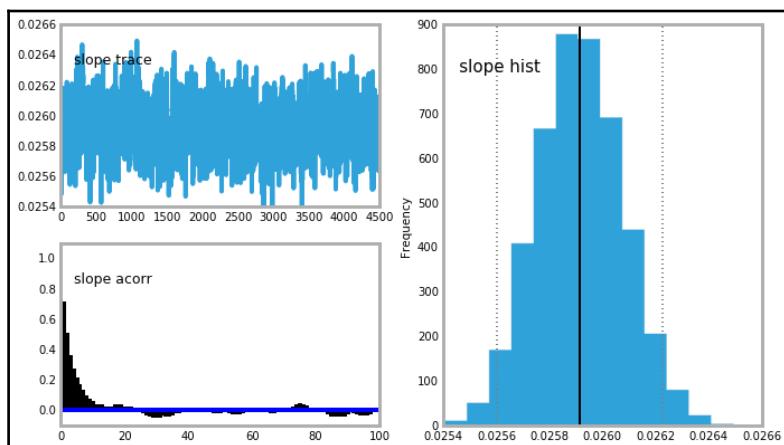
You just ran half a million iterations in less than a minute! Due to the thinning, the post-sampling analysis is a bit quicker:

```
y_min = MDL.stats()['linear']['quantiles'][2.5]
y_max = MDL.stats()['linear']['quantiles'][97.5]
y_fit = MDL.stats()['linear']['mean']

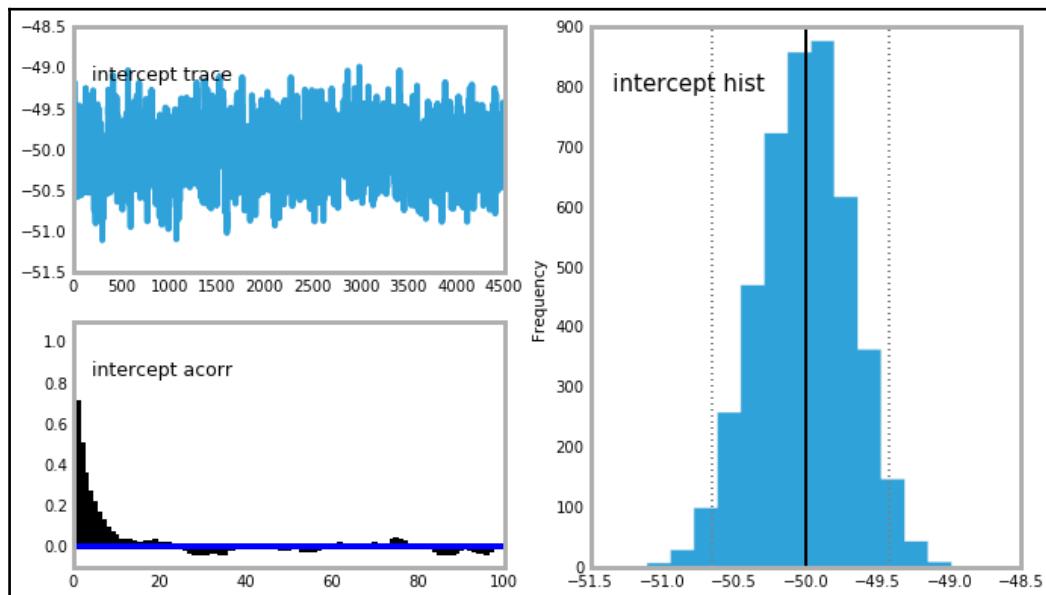
slope = MDL.stats()['slope']['mean']
slope_err = MDL.stats()['slope']['standard deviation']
intercept = MDL.stats()['intercept']['mean']
intercept_err = MDL.stats()['intercept']['standard deviation']
```

We should also create the plots for the time series, posterior distribution, and autocorrelation:

```
mcplt.plot(MDL)
```



The trace, posterior distribution, and autocorrelation plots all look very good—sharp, well-defined peaks and stable time series. This is also true for the intercept variable:



Before we plot the results, I also want us to use one of the other packages, the statsmodels ordinary least square fitting. Just as before, we import the formula package so that we can simply give Pandas the column names that we want to find the relationship between:

```
import statsmodels.formula.api as smf
from statsmodels.sandbox.regression.predstd import wls_prediction_std
ols_results = smf.ols("rate ~ year", co2_gr).fit()
```

We then grab the best fitting parameters and their uncertainty. Here, I flip the parameter tuples for convenience:

```
prstd, iv_l, iv_u = wls_prediction_std(ols_results)
ols_params = np.flipud(ols_results.params)
ols_err = np.sqrt(np.diag(ols_results.cov_params()))**.5
```

We can now compare the two methods, least square (frequentist) and Bayesian model fitting:

```
print('OLS: slope:{0:.3f}, intercept:{1:.2f}'.format(*ols_params))
print('Bay: slope:{0:.3f}, intercept:{1:.2f}'.format(slope, intercept))
```

OLS: slope:0.027, intercept:-51.81
Bay: slope:0.026, intercept:-50.01

The Bayesian method seems to find the best estimates of the parameters just below the ordinary least square method. The parameters are close enough for us to call it even, but consistently closer to zero—an interesting observation. We will get back to this same dataset when we look at machine learning algorithms in the next chapter. I also want to look at the confidence versus credible intervals. We do this for the OLS and Bayesian model fit with the following method:

```
ols_results.conf_int(alpha=0.05)
```

	0	1
Intercept	-66.531103	-37.092365
year	0.019425	0.034240

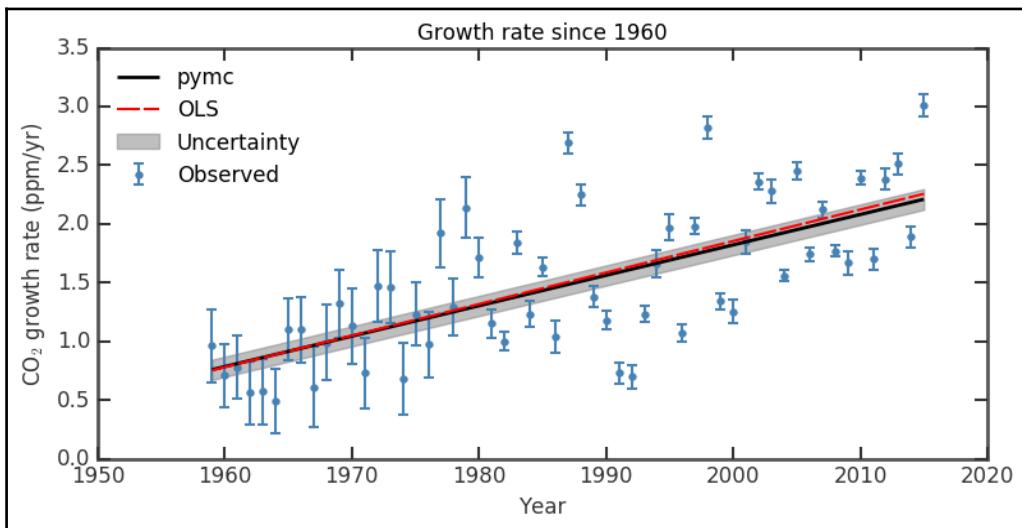
The `alpha=0.05` gives the confidence interval level, where it means the 95% confidence interval (that is, $1 - 0.05 = 0.95$):

```
MDL.stats(['intercept', 'slope'])
```

```
{'intercept': {'95% HPD interval': array([-50.64990877, -49.42168758]),  
  'mc error': 0.011931753405578262,  
  'mean': -50.008066085637815,  
  'n': 4500,  
  'quantiles': {2.5: -50.646760270081515,  
    25: -50.228993605017173,  
    50: -50.004119841092979,  
    75: -49.789140434449351,  
    97.5: -49.412790669964195},  
  'standard deviation': 0.31970980437231761},  
{'slope': {'95% HPD interval': array([ 0.02560229,  0.02622579]),  
  'mc error': 6.0141748688389682e-06,  
  'mean': 0.0259157602742307,  
  'n': 4500,  
  'quantiles': {2.5: 0.025609913276748567,  
    25: 0.025803373826353462,  
    50: 0.02591258446734945,  
    75: 0.026027359901668552,  
    97.5: 0.026237852465415799},  
  'standard deviation': 0.00016186303140381465}}}
```

So the confidence level for the intercept is $[-66.5, -37.1]$ for the OLS fit, and the credible interval from the Bayesian fit is $[-50.6, -49.4]$. This highlights the difference between the two methods. Let's now finally plot the results, and I want to draw both the fits in the same plot:

```
plt.figure(figsize=(10,6))  
plt.title('Growth rate since 1960');  
plt.errorbar(x,y,yerr=y_error,  
             color='SteelBlue', ls='None',  
             linewidth=1.5, capthick=1.5,  
             marker='.', ms=8,  
             label='Observed')  
plt.xlabel('Year')  
plt.ylabel('CO$_2$ growth rate (ppm/yr)')  
plt.plot(x, y_fit,  
         'k', lw=2, label='pymc')  
plt.fill_between(x, y_min, y_max,  
                 color='0.5', alpha=0.5,  
                 label='Uncertainty')  
plt.plot([x.min(), x.max()],  
        [ols_results.fittedvalues.min(), ols_results.fittedvalues.max()],  
        'r', dashes=(13,2), lw=1.5, label='OLS', zorder=32)  
plt.legend(loc=2, numpoints=1);
```



Here, I use the matplotlib function, `fill_between`, to show the credible interval of the function. The fit of the data looks good, and after looking at it a few times, you might realize that it looks as if it is divided into two—two linear segments with some offset divided around 1985. One exercise for you is to test this hypothesis: create a function with two linear segments with a break in a certain year, and then try to constrain the model and compare the results. Why could this be? Perhaps they changed the instrument; if you remember, the two parts of the data also have different uncertainties, so a difference in systematic error is not completely unlikely.

Summary

We have discussed how to test models and hypotheses with Bayesian analysis using the Python package, PyMC. It is a powerful package that gives out more intuitive results, where you see how the parameters are characterized. Not all posterior distributions are shaped like Gaussian, but the trace and autocorrelation should look similar for well-constrained parameters.

In the next chapter, we will dive into some of the machine learning algorithms available in Python and look at how they can identify clusters, classify data, and do linear regression. As in this chapter, we will compare the linear fit with that of Bayesian analysis and OLS. We will compare the cluster findings with the analysis that we did of galaxies in the universe in Chapter 5, *Clustering*.

7

Supervised and Unsupervised Learning

The amount of data collected for various purposes in society has increased enormously in the last few decades. Machine learning is a way of making sense of all this data by leveraging what we know about the data. In the generalized picture of machine learning, the computer first learns from a given dataset (training) and creates a generalized model to represent it. With this model, it is possible to predict various outcomes, results, and groupings (classes). In this chapter, we will cover the following topics:

- Linear regression with machine learning algorithms
- Clustering with machine learning algorithms
- Feature selection—a preprocessing method to select what is most important
- Classification with different machine learning algorithms and kernels

Before getting started, I will give you a brief introduction to machine learning and the package that we will use: Scikit-learn.

Introduction to machine learning

There are three main categories of machine learning: supervised, unsupervised, and reinforced. Given a simple dataset with input x and output y , supervised learning is when both x and y have known labels. The algorithm maps x to y and after training, it can predict y values with x as input. Contrary to this, unsupervised learning is when only x is labeled and the algorithm finds a label for y itself. Reinforced learning is when the computer learns without the need to map the input to an outcome and instead responds to the input. This is how algorithms that play chess or other games work.

They try to predict how to react to input without a clearly quantifiable outcome, instead seeking reinforcement; one example being to play the game continuously until it ends without making a mistake (that is, win). One feature-rich and popular package for machine learning in Python is Scikit-learn.

Scikit-learn

Scikit-learn is part of the SciPy Toolkits, which are packages that are affiliated with SciPy. More information on SciPy toolkits and a list of available ones can be found at <https://www.scipy.org/scikits.html>. The first release of Scikit-learn came in 2007, but the first publication presenting the package in 2011 was Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011. For a wealth of examples, documentation, and reading, see the Scikit-learn web page (<http://scikit-learn.org>). The package is well-maintained and the documentation is excellent and very extensive in coverage.

After this brief introduction, we do as we did in the previous chapters—we start a Jupyter Notebook and run the standard imports.

Now we also, of course, want to import Scikit-learn. The following code imports it and also prints out the version number of the installed Scikit-learn:

```
import sklearn  
sklearn.__version__
```

Next, I have created a function that removes the right and top axis in a plot or a grid of plots. It comes in handy when producing figures; it is important to be able to present the data and results from the analysis in a clear and focused way. Removing unnecessary lines in a plot is part of this and also saves text space. The name of the `despine` function is inspired by the equivalent function in the excellent package, Seaborn, which can help you make nice figures as well (<https://stanford.edu/~mwaskom/software/seaborn/>):

```
def despine(axs):  
    # to be able to handle subplot grids  
    # it assumes the input is a list of  
    # axes instances, if it is not a list,  
    # it puts it in one  
    if type(axs) != type([]):  
        axs = [axs]  
    for ax in axs:  
        ax.yaxis.set_ticks_position('left')  
        ax.xaxis.set_ticks_position('bottom')  
        ax.spines['bottom'].set_position(('outward', 10))  
        ax.spines['left'].set_position(('outward', 10))
```

Linear regression

There are many different linear regression models built-in in Scikit-learn, **Ordinary Least Squares (OLS)** and **Least Absolute Shrinkage and Selection Operator (LASSO)** to name two. The difference between these two can be approximated by different loss functions, which is the function that is worked on by the machine learning algorithm. In LASSO, there is an added penalty going away from the fitted function, whereas OLS is simply the least square equation. However, the routine is still different from the OLS that we covered earlier; the underlying algorithm to reach the answer is a machine learning algorithm. One such common algorithm is gradient decent. Here, we shall take the climate data from the previous chapter and fit a linear function to it with two methods, then we will compare the results from the OLS model with that of PyMC's Bayesian inference (Chapter 6, *Bayesian Methods*) and statsmodels' OLS (Chapter 4, *Regression*).

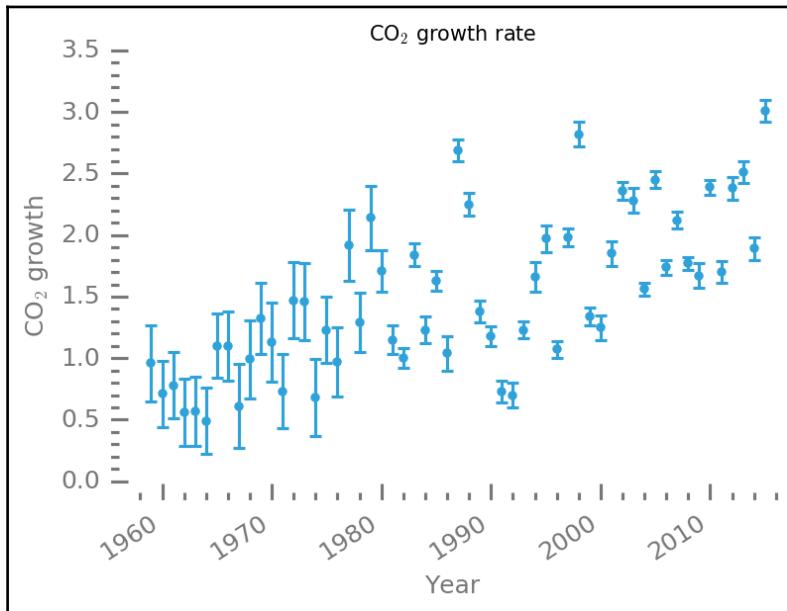
Climate data

We begin by reading in the data on the CO₂ growth rate for the past 60 years:

```
co2_gr = pd.read_csv('data/co2_gr_gl.txt',
                      delim_whitespace=True,
                      skiprows=62,
                      names=['year', 'rate', 'err'])
```

To refresh your memory, we plot the data again. We now use our `despine` function to remove two of the axes:

```
fig, ax = plt.subplots(1,1)
ax.errorbar(co2_gr['year'], co2_gr['rate'],
            yerr=co2_gr['err'],
            ls='None',
            elinewidth=1.5,
            capthick=1.5,
            marker='.',
            ms=8)
despine(ax)
plt.minorticks_on()
labels = ax.get_xticklabels()
plt.setp(labels, rotation=33, ha='right')
ax.set_ylabel('CO$_2$ growth')
ax.set_xlabel('Year')
ax.set_xlim((1957,2016))
ax.set_title('CO$_2$ growth rate');
```



The data has a lot of spread in the middle of the range, roughly between 1980 and 2000. In one way, it looks like two lines with a jump at 1985 could be fitted. However, it is not clear why this would be the case; perhaps the two distributions of uncertainties have something to do with it?

In Scikit-learn, the learning is done by first initiating the estimator, an object where we call the fit method to train the dataset. This means that we first have to import the estimator. In this example, I will show you two different estimators and the resulting fits that they produce. The first is the simple linear model and the second is the LASSO estimator. There are many different linear models in Scikit-learn: RANSAC, Theil-Sen, and linear models based on **stochastic gradient descent (SGD)** learning, to name a few. After going through this example, you should look at another estimator and try it out. We first import the ones that we will work with, and also the `cross_validation` function, which we will use to separate the dataset into two parts—training and testing:

```
from sklearn.linear_model import LinearRegression, Lasso  
from sklearn import cross_validation
```

As we want to be able to validate our fit and see how good it is, we do not use all of the data. We use the `train_test_split` function in `cross_validation` to put 25% of the data in a testing set and 75% in the training set. Play around with different values and see how the fit results change in the end. Then, we store the `x` and `y` values in appropriate structures. The `x` values have to have one extra axis. This could also be done with the `x_train.reshape(-1, 1)` code; the way we do it here gives the same effect. We also create an array to later plot the fit with `x` values that we know span the whole range and a bit more:

```
x_test, x_train, y_test, y_train = cross_validation.train_test_split(  
    co2_gr['year'], co2_gr['rate'],  
    test_size=0.75,  
    random_state=0)  
x_train = x_train[:, np.newaxis]  
x_test = x_test[:, np.newaxis]  
line_x = np.array([1955, 2025])
```

In the training and test data split, we also make use of the `random_state` parameter so that the random seed is the same and we get the same division of the training and testing set by running it multiple times (for exact reproducibility). We are now ready to train the data, the first is the simple linear regression model. To run the machine learning algorithms on the training set, we first create an estimator object/class and then we simply train the model by calling the `fit` method with the training `x` and `y` values as input:

```
est_lin = LinearRegression()  
est_lin.fit(X_train, y_train)  
lin_pred = est_lin.predict(line_x.reshape(-1, 1))
```

Here, I also added a calculation of the predicted `y` values from the array that we created earlier, and to show an alternative way of restructuring the input array, I used the `reshape` method. Next is the LASSO model, and we do exactly the same, except that in creating the model object we now have the option of giving it extra parameters. The `alpha` parameter is basically what separates this model from the preceding simple linear regression model. If it is set to zero, the model becomes the same as the linear model. The LASSO model `alpha` input modifies the loss function and the default value is 1. Try different values, although 0 is not good to choose as the model is not made to operate without the added penalty to the loss function:

```
est_lasso = Lasso(alpha=0.7)  
est_lasso.fit(X_train, y_train)  
lasso_pred = est_lasso.predict(line_x.reshape(2, 1))
```

To see the results, we first print out the estimates of the coefficients, mean square discrepancy or error (mean squared residuals), and variance score. The variance score is a method in the Scikit-learn model (estimator) that we create. While a variance score of 1 means that it is able to predict the values perfectly, a score of 0 means that no values were predicted and there is no (linear) relationship between the variables. The coefficients are accessed with `estimator.coef_` and `estimator.intercept_`. To get the mean square error, we simply take the difference between the predicted and observed values, which are calculated with `estimator.predict(x)`, where `x` is `x` values where you want to predict `y` values. This should be calculated on the test data and not the training set. We first create a function to calculate this and print the relevant diagnostics:

```
def printstuff(estimator, A, b):
    name = estimator.__str__().split('(')[0]
    print('+'*6, name, '+'*6)
    print('Slope: {:.3f} Intercept:{:.2f} '.format(
        estimator.coef_[0], estimator.intercept_))
    print("Mean squared residuals: {:.2f}".format(
        np.mean((estimator.predict(A) - b)**2)))
    print('Variance score: {:.2f}'.format(
        estimator.score(A, b)))
```

With this function, we can now print the results of the fitting in a way that gives us the estimated slope and intercept, mean squared residuals, and variance score:

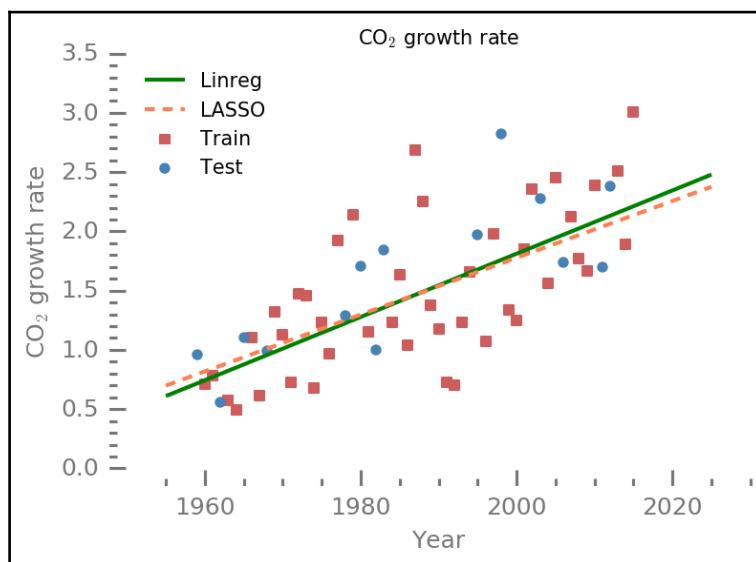
```
printstuff(est_lin, X_test, y_test)
printstuff(est_lasso, X_test, y_test)
```

```
+++++ LinearRegression ++++++
Slope: 0.027 Intercept:-51.60
Mean squared residuals: 0.17
Variance score: 0.56
+++++ Lasso ++++++
Slope: 0.024 Intercept:-46.16
Mean squared residuals: 0.17
Variance score: 0.56
```

The LASSO model estimates lower values to the slope and intercept, but gives a similar mean squared residuals and variance score to the linear regression model. The spread in the data is too much to conclude whether any of them produce more reliable estimates. We can, of course, plot all of these results in a figure together with the data:

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.scatter(X_train, y_train, marker='s',
           label='Train', color='IndianRed')
```

```
ax.scatter(X_test, y_test, label='Test',
           color='SteelBlue')
ax.plot(line_x, lin_pred, color='Green',
        label='Linreg', lw=2)
ax.plot(line_x, lasso_pred, color='Coral',
        dashes=(5,4), label='LASSO', lw=2)
ax.set_xlabel('Year')
ax.set_ylabel('CO2 growth rate')
ax.legend(loc=2, fontsize=10, numpoints=1)
despine(ax)
plt.minorticks_on()
ax.locator_params(axis='x', nbins=5)
ax.locator_params(axis='y', nbins=7)
ax.set_xlim(1950,2030)
ax.set_title('CO2 growth rate');
```



The fits differ slightly, but are definitely within each other's uncertainty. The spread in the data points is relatively large. The squares show the training set and the circles show the testing (validation) set. Extrapolating outside this range would, however, predict significantly different values.

We can calculate the R² score, just as before with classical OLS regression:

```
from sklearn.metrics import r2_score
r2_lin = r2_score(co2_gr['rate'],
                  est_lin.predict(
                      co2_gr['year'].reshape(-1,1)))
r2_lasso = r2_score(co2_gr['rate'],
                     est_lasso.predict(
                         co2_gr['year'].reshape(-1,1)))
print('LinearSVC: {:.2f}\nLASSO:\n\t{:.2f}'.format(r2_lin, r2_lasso))
```

LinearSVC: 0.49
LASSO: 0.48

The R² values are relatively high, despite the significant spread in the data points and limited size of the data.

Checking with Bayesian analysis and OLS

We will quickly make a comparison with both the statsmodels' OLS regression and Bayesian inference with a linear model. The Bayesian inference and OLS fits are the same as in Chapter 6, *Bayesian Methods*, and a small version is repeated here:

```
import pymc
x = co2_gr['year'].as_matrix()
y = co2_gr['rate'].as_matrix()
y_error = co2_gr['err'].as_matrix()
def model(x, y):
    slope = pymc.Normal('slope', 0.1, 1.)
    intercept = pymc.Normal('intercept', -50., 10.)
    @pymc.deterministic(plot=False)
    def linear(x=x, slope=slope, intercept=intercept):
        return x * slope + intercept
    f = pymc.Normal('f', mu=linear, tau=1.0/y_error,
                    value=y, observed=True)
    return locals()
MDL = pymc.MCMC(model(x,y))
MDL.sample(5e5, 5e4, 100)
y_fit = MDL.stats()['linear']['mean']
slope = MDL.stats()['slope']['mean']
intercept = MDL.stats()['intercept']['mean']
```

For the OLS model, we again use the formula framework to express the relationship between our variables:

```
import statsmodels.formula.api as smf
from statsmodels.sandbox.regression.predstd import wls_prediction_std
ols_results = smf.ols("rate ~ year", co2_gr).fit()
ols_params = np.flipud(ols_results.params)
```

Now that we have the results from all three methods, let's print out the slope and intercept for them:

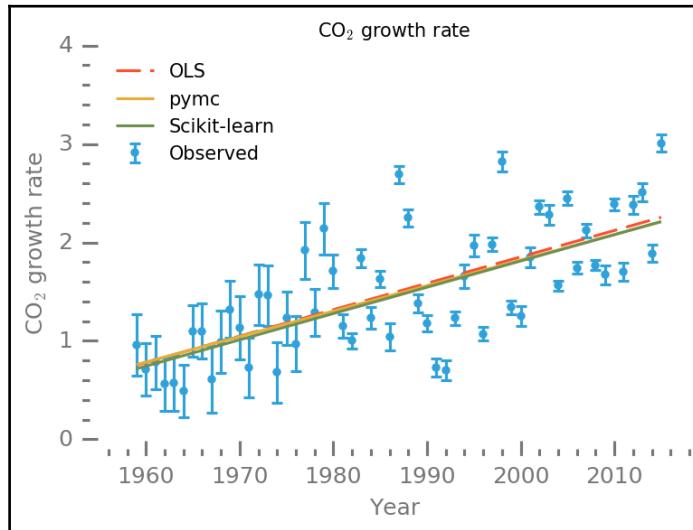
```
print('      Slope Intercept \nML : \
{0:.3f} {1:.3f} \nOLS: {2:.3f} \
{3:.3f} \nBay: {4:.3f} \
{5:.3f}'.format(est_lin.coef_[0], est_lin.intercept_,
                 ols_params[0], ols_params[1],
                 slope, intercept) )
```

	Slope	Intercept
ML :	0.027	-51.597
OLS:	0.027	-51.812
Bay:	0.026	-49.998

While the overall results are similar, Bayesian inference seems to estimate absolute values that are lower than the other methods. We can now visualize these different estimates together with the data:

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.errorbar(x, y, yerr=y_error, ls='None',
            elinewidth=1.5, capthick=1.5,
            marker='.', ms=8, label='Observed')
ax.set_xlabel('Year')
ax.set_ylabel('CO2 growth rate')
ax.plot([x.min(), x.max()],
        [ols_results.fittedvalues.min(),
         ols_results.fittedvalues.max()],
        lw=1.5, label='OLS',
        dashes=(13,5))
ax.plot(x, y_fit, lw=1.5,
        label='pymc')
ax.plot([x.min(), x.max()],
        est_lin.predict([[x.min(), ], [x.max(), ]]),
        label='Scikit-learn', lw=1.5)
despine(ax)
ax.locator_params(axis='x', nbins=7)
```

```
ax.locator_params(axis='y', nbins=4)
ax.set_xlim((1955,2018))
ax.legend(loc=2, numpoints=1)
ax.set_title('CO2 growth rate');
```



This might look like a very small difference; however, if we use these different results to extrapolate 30, 50, or even 100 years into the future, these will yield significantly different results. This example shows you how simple it is to try out different methods and models in Scikit-learn. It is also possible to create your own. Next up, we will look at one cluster identification model in Scikit-learn, DBSCAN.

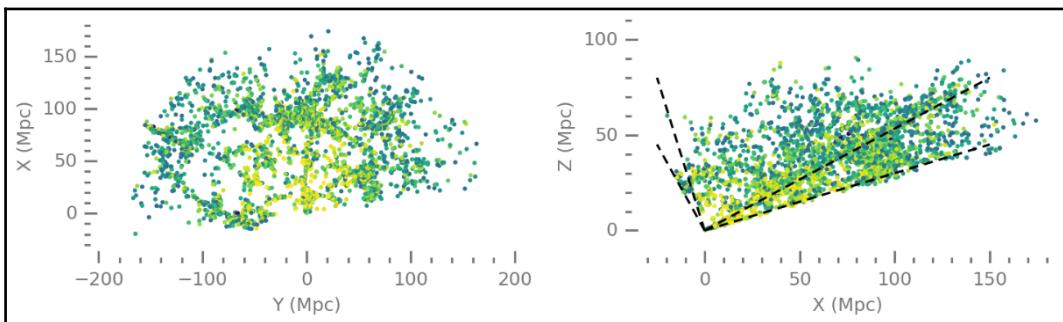
Clustering

In this example, we will look at a cluster finding algorithm in Scikit-learn called **DBSCAN**. DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise, and is a clustering algorithm that favors groups of points and can identify points outside any of these groups (clusters) as noise (outliers). As with the linear machine learning methods, Scikit-learn makes it very easy to work with it. We first read in the data from Chapter 5, *Clustering*, with Pandas' `read_pickle` function:

```
TABLE_FILE = 'data/test.pickle'
mycat = pd.read_pickle(TABLE_FILE)
```

As with the previous dataset, to refresh your memory, we plot the data. It contains a slice of the mapped nearby Universe, that is, galaxies with determined positions (direction and distance from us). As before, we scale the color with the Z-magnitude, as found in the data table:

```
fig,ax = plt.subplots(1,2, figsize=(10,2.5))
plt.subplot(121)
plt.scatter(mycat['Y'], -1*mycat['X'],
            s=8,
            color=plt.cm.viridis_r(
                10** (mycat.Zmag-myCat.Zmag.max()) ),
            edgecolor='None')
plt.xlabel('Y (Mpc)'); plt.ylabel('X (Mpc)')
ax = plt.gca()
despine(ax)
ax.locator_params(axis='x', nbins=5)
ax.locator_params(axis='y', nbins=5)
plt.axis('equal')
plt.subplot(122)
c_arr = 10** (mycat.Zmag-myCat.Zmag.max())
plt.scatter(-1*mycat['X'], mycat['Z'],
            s=8,
            color=plt.cm.viridis_r(c_arr),
            edgecolor='None')
lstyle = dict(lw=1.5, color='k', dashes=(6,4))
ax = plt.gca()
despine(ax)
ax.locator_params(axis='x', nbins=5)
ax.locator_params(axis='y', nbins=5)
plt.plot([0,150], [0,80], **lstyle)
plt.plot([0,150], [0,45], **lstyle)
plt.plot([0,-25], [0,80], **lstyle)
plt.plot([0,-25], [0,45], **lstyle)
plt.xlabel('X (Mpc)'); plt.ylabel('Z (Mpc)')
plt.subplots_adjust(wspace=0.3)
plt.axis('equal');
plt.ylim((-10,110));
```



The data spans extremely large scales and many galaxies in the given directions. To start using machine learning for cluster finding, we import the relevant objects from Scikit-learn:

```
from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.preprocessing import StandardScaler
```

The first import is simply the DBSCAN method, and the second import is the metrics module with which we can calculate various statistics on the clustering algorithm.

The StandardScaler class is simply to scale the data, as in Chapter 5, *Clustering*. Next, we set up the input data; every row should contain the coordinates of a feature/point scaled. This scaled coordinate list is then input into the DBSCAN method:

```
A = np.array([mycat['Y'], -1*mycat['X'], mycat['Z']]).T
A_scaled = StandardScaler().fit_transform(A)
dabout = DBSCAN(eps=0.15, min_samples=5).fit(A_scaled)
```

The DBSCAN object is instantiated with several parameters. The `eps` parameter limits the size of the cluster in terms of the distance within which at least `min_samples` have to lie for it to be a cluster (remember, in scaled units). The `dabout` object now stores all the results of the fit. The `dabout.labels_` array contains all the labels for each point; points not in any cluster are given a `-1` label. Let's check whether we have any:

```
(dabout.labels_ == -1).any()
```

It prints out `True`, so we have noise. Another important method that the output object has is `core_sample_indices_`. It contains the core samples from which each cluster is expanded and formed. It is almost like the centroid positions in k-means clustering. We now create a Boolean array for the core sample indices and also a list of the unique labels in the results. This is the recommended way according to the Scikit-learn documentation.

```
csmask = np.zeros_like(dabout.labels_, dtype=bool)
csmask[dabout.core_sample_indices_] = True
unique_labels = set(dabout.labels_)
```

Without the true labels of the clusters, it is tricky to measure the success of the cluster finding. Normally, you would calculate the silhouette score, which is a score that scales with the distance between the centroid and samples in the same cluster and nearby clusters. The higher the silhouette score, the better the cluster finding was at defining the cluster. However, this assumes clusters that are centered around one point, not a filamentary structure. To show you how to calculate and interpret the silhouette score, we go through it for this example, but keep in mind that it might not be a representative method in this case. We will calculate the silhouette score and also print out the number of clusters found. Remember that the labels array also contains the noise labels (that is, -1):

```
n_clusters = len(set(labels)) - [0,1][-1 in labels]
print('Estimated number of clusters: %d' % n_clusters)
print("Silhouette Coefficient: %0.3f"
      % metrics.silhouette_score(A_scaled, dabout.labels_))
```

Estimated number of clusters: 8
Silhouette Coefficient: -0.143

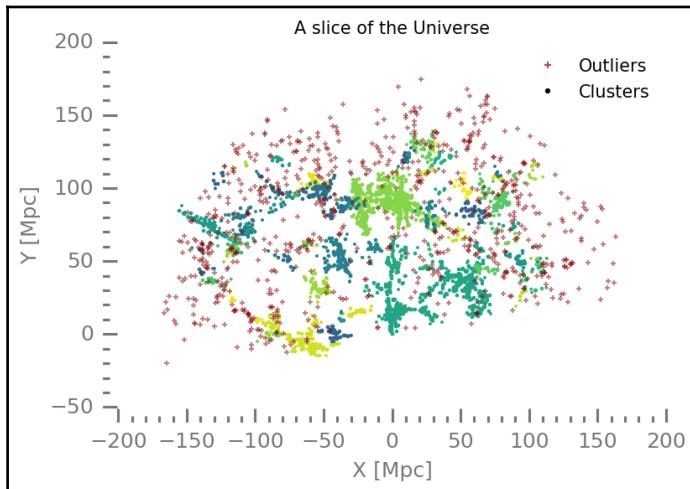
Silhouette score values close to zero indicate that the clusters overlap. Now we will plot all the results and check what it looks like. I have tried to plot the core samples differently by increasing their marker size and decreasing the size of non-core samples. I also shuffled the colors in an attempt at making different clusters stand out against their neighbors:

```
colors = plt.cm.viridis(np.linspace(0.3, 1, len(unique_labels)))
np.random.seed(0)
np.random.shuffle(colors)
for lbl, col in zip(unique_labels, colors):
    if lbl == -1:
        # Black used for noise.
        col = 'DarkRed'; m1=m2= '+'; s = 10; a = 0.5
    else:
        m1='.'; m2='.'; s=5; a=1
    cmmask = (dabout.labels_ == lbl)
    xy = A[cmmask & csmask]
    plt.scatter(xy[:, 0], xy[:, 1], color=col,
                marker=m1,
                s=s+1,
                alpha=a)
    xy = A[cmmask & ~csmask]
    plt.scatter(xy[:, 0], xy[:, 1], color=col,
```

```

marker=m2,
s=s-2,
alpha=a)
despine(plt.gca())
noiseArtist = plt.Line2D((0,1),(0,0),
                           color='DarkRed',
                           marker='+',
                           linestyle='',
                           ms=4, mew=1,
                           alpha=0.7)
clusterArtist = plt.Line2D((0,1),(0,0),
                           color='k',
                           marker='.',
                           linestyle='',
                           ms=4, mew=1)
plt.legend([noiseArtist, clusterArtist],
           ['Outliers', 'Clusters'],
           numpoints=1)
plt.title('A slice of the Universe')
plt.xlabel('X [Mpc]')
plt.ylabel('Y [Mpc]');

```



The algorithm also finds noise (outliers), which are plotted in red crosses here. Try to tweak the displaying of the core and non-core samples so that they stand out more. Furthermore, you should try different parameters for the DBSCAN method and see how the outcome is affected. Another thing would be to go back to Chapter 5, *Clustering*, and put 66 clusters in the hierarchical cluster algorithm that we tried there with the same dataset and compare.

Seeds classification

We will now look at three main groups of classification (learning) models: **Support Vector Machine (SVM)**, Nearest Neighbor, and Random Forest. SVM simply divides the space in N regions, separated by a boundary. The boundary can be allowed to have different shapes, for example, there is a linear boundary or quadratic boundary. The Nearest Neighbor classification identifies the k -nearest neighbors and classifies the current data point depending on what class the k -nearest neighbors belong to. The Random Forest classifier is a decision tree learning method, which, in simple terms, creates rules from the given training data to be able to classify new data. A set of if-statements in a row is what gives it the name decision tree.

The data that we are going to use comes from the UCI Machine Learning repository (Lichman, M. (2013)-<http://archive.ics.uci.edu/ml>. (Irvine, CA: University of California, School of Information and Computer Science). The dataset contains several measured attributes of three different types of wheat grains (M. Charytanowicz, J. Niewczas, P. Kulczycki, P.A. Kowalski, S. Lukasik, S. Zak, *A Complete Gradient Clustering Algorithm for Features Analysis of X-ray Images*, in: Information Technologies in Biomedicine, Ewa Pietka, Jacek Kawa (eds.), Springer-Verlag, Berlin-Heidelberg, 2010, pp. 15-24.).

We want to create a classifier, something that, if we measure specific parameters of a seed, can tell what type of seed it is. With the dataset, a description of the columns are supplied, which can also be found on the UCI web page for the dataset. There are eight columns, seven for the parameters and one for the known type of the seed (that is, the label). I have created a text file for this; in case you are running a Linux-based system, you can list the contents with Jupyter magic:

```
%%bash  
less data/seeds.desc
```

- 1. area A,
- 2. perimeter P,
- 3. compactness C = $4\pi A/P^2$,
- 4. length of kernel,
- 5. width of kernel,
- 6. asymmetry coefficient
- 7. length of kernel groove.
- 8. group



If you are running Microsoft Windows, you can use the `more` command, that is, `more data/seeds.desc`, which gives you the same output but in a pop-up window, not as convenient but still useful.

Now that we know what columns are there, we can read it into a Pandas DataFrame:

```
seeds = pd.read_csv('data/seeds_dataset.txt',
                     delim_whitespace=True,
                     names=['A', 'P', 'C', 'lkern', 'wkern',
                            'asym', 'lgro', 'gr'])
```

As always, list the contents of what was read:

```
seeds.head()
```

	A	P	C	lkern	wkern	asym	lgro	gr
0	15.26	14.84	0.8710	5.763	3.312	2.221	5.220	1
1	14.88	14.57	0.8811	5.554	3.333	1.018	4.956	1
2	14.29	14.09	0.9050	5.291	3.337	2.699	4.825	1
3	13.84	13.94	0.8955	5.324	3.379	2.259	4.805	1
4	16.14	14.99	0.9034	5.658	3.562	1.355	5.175	1

Visualizing the data

We could just run the whole classification process on all seven parameters in the dataset. This is computationally expensive, and the costs increase very quickly when increasing the amount of data. To make a first attempt at selecting only the attributes that matter for classification, I want to visually inspect the distribution of the values for the attributes for the different types of grains. To do this, we first create a selection filter for the different groups:

```
gr1 = seeds.gr == 1
gr2 = seeds.gr == 2
gr3 = seeds.gr == 3
```

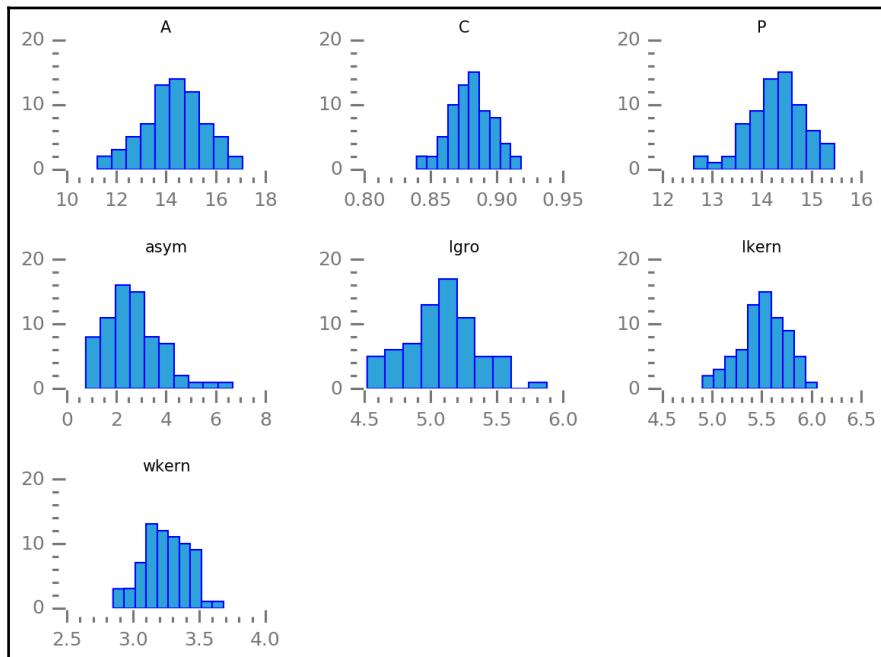
To only plot the relevant parameters, we also create a list of the ones that we want to see (that is, not the type of grain):

```
pars = ['A', 'C', 'P', 'asym', 'lgro', 'lkern', 'wkern']
```

With Pandas' built-in histogram function, we can plot the attributes for each group. I have added some extra commands to make the figures look a bit better and more clutter-free:

```
axes = seeds[pars][gr1].hist(figsize=(8, 6))
despine(list(axes.flatten()))
_ = [ax.grid() for ax in list(axes.flatten())]
_ = [ax.locator_params(axis='x', nbins=4) for ax in
     list(axes.flatten())]
_ = [ax.locator_params(axis='y', nbins=2) for ax in
     list(axes.flatten())]
plt.subplots_adjust(wspace=0.5, hspace=0.7)
```

Once again, we use the `despine` function to make the plots clearer. The preceding code will plot all of the attributes for the first group, `gr1`. The histogram will show you how the values are distributed:

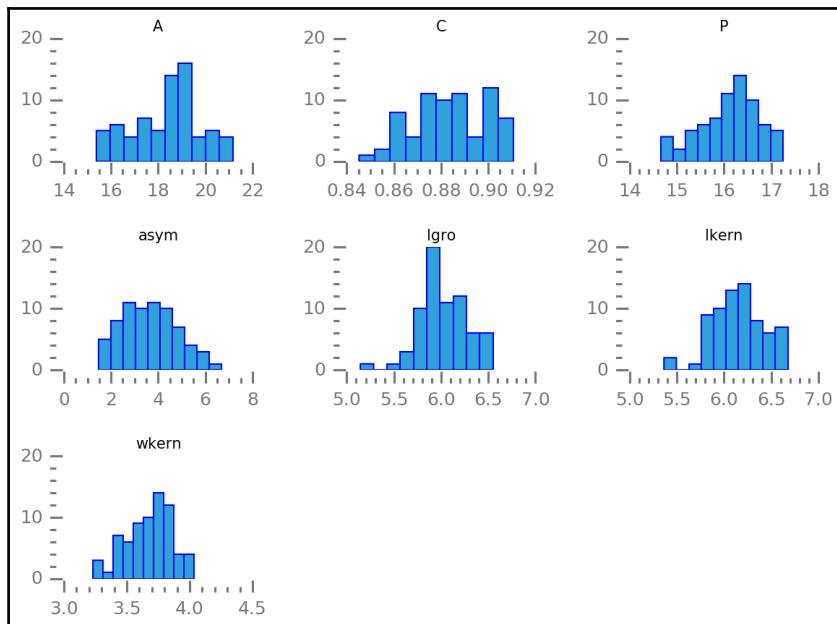


With the selection filters, it is easy to plot the other groups:

```
axes = seeds[pars][gr2].hist(figsize=(8, 6))
despine(list(axes.flatten()))
_ = [ax.grid() for ax in list(axes.flatten())]
_ = [ax.locator_params(axis='x', nbins=4) for ax in
```

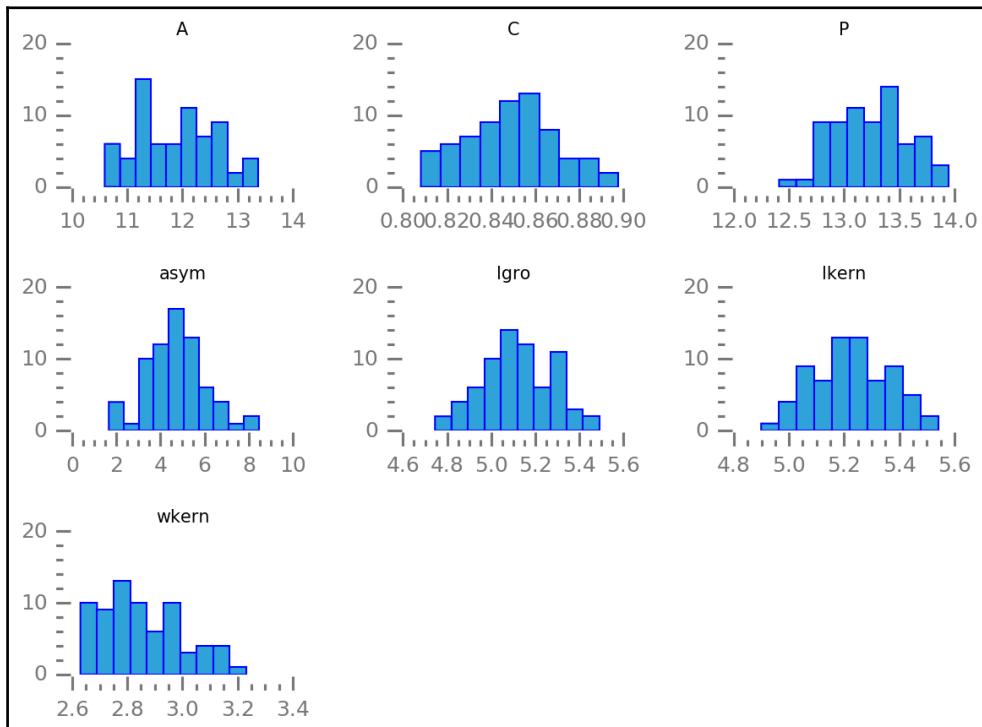
```
list(axes.flatten())
_ = [ax.locator_params(axis='y', nbins=2) for ax in
     list(axes.flatten())]
plt.subplots_adjust(wspace=0.5, hspace=0.7)
```

By plotting all the groups, we can look at how the distribution of values for the attributes differ. We are trying to identify the ones that differ the most between groups:



```
axes = seeds[pars][gr3].hist(figsize=(8,6))
despine(list(axes.flatten()))
_ = [ax.grid() for ax in list(axes.flatten())]
_ = [ax.locator_params(axis='x', nbins=5) for ax in
     list(axes.flatten())]
_ = [ax.locator_params(axis='y', nbins=2) for ax in
     list(axes.flatten())]
plt.subplots_adjust(wspace=0.5, hspace=0.7)
```

After plotting this last group, we can look at each attribute and find the attribute where the distribution is separate for each group. This way, we can use it to distinguish the various groups (types of seeds), that is, classify them:

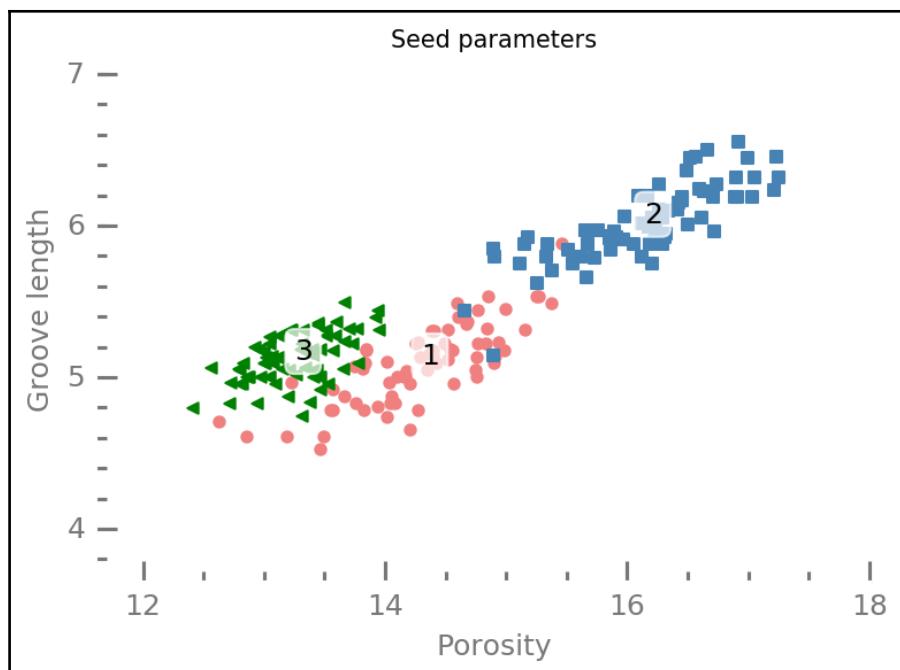


From this, I would argue that porosity and groove length are good parameters due to their fairly well-defined and, for the three groups, separated peaks. To check this, we plot them against each other. We also want to mark the various groups of grains:

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.scatter(seeds.P[gr1], seeds.lgro[gr1],
           color='LightCoral')
ax.scatter(seeds.P[gr2], seeds.lgro[gr2],
           color='SteelBlue', marker='s')
ax.scatter(seeds.P[gr3], seeds.lgro[gr3],
           color='Green', marker='<');
ax.text(seeds.P[gr1].mean(), seeds.lgro[gr1].mean(),
       '1', bbox=dict(color='w', alpha=0.7,
                      boxstyle="Round"))
ax.text(seeds.P[gr2].mean(), seeds.lgro[gr2].mean(),
       '2', bbox=dict(color='w', alpha=0.7,
                      boxstyle="Round"))
ax.text(seeds.P[gr3].mean(), seeds.lgro[gr3].mean(),
       '3', bbox=dict(color='w', alpha=0.7,
                      boxstyle="Round"))
```

```
'2', bbox=dict(color='w', alpha=0.7,
                 boxstyle="Round"))
ax.text(seeds.P[gr3].mean(), seeds.lgro[gr3].mean(),
        '3', bbox=dict(color='w', alpha=0.7,
                       boxstyle="Round"))

ax.set_xlabel('Porosity')
ax.set_ylabel('Groove length')
ax.set_title('Seed parameters')
despine(ax)
plt.minorticks_on()
ax.locator_params(axis='x', nbins=5)
ax.locator_params(axis='y', nbins=4)
ax.set_xlim(11.8,18)
ax.set_ylim(3.8,7.1);
```



Now, this is only two of the parameters; we have several more. However, from this, it seems that it is the hardest to separate groups 1 and 3, that is, the circles and triangles.

Feature selection

Built-in into Scikit-learn are several ways of determining the best parameters to look at. This is sometimes called feature selection, which is trying to determine which parameters have the biggest differences between each other and are best suited to describe the various groups as exactly that-distinct groups. Here, we use one where we can give a number, K (not to be confused with K in K-means), which determines up to what number of features it should select the best ones.

First, we store the seeds table as a matrix, a NumPy array, and then we separate the data and labels:

```
X_raw = seeds.as_matrix()  
X_pre, labels = X_raw[:, :-1], X_raw[:, -1]
```

Now we can import the selection algorithm and run it on the data. Note that we also import the chi2 estimator and supply it to the selection object. This means that chi-squared minimization will be used to determine the best parameters:

```
from sklearn.feature_selection import SelectKBest  
from sklearn.feature_selection import chi2  
X_best = SelectKBest(chi2, k=2).fit_transform(X_pre, labels)
```

It has now selected two columns; to check which columns, we print out the first few rows of the selection and the raw data:

```
X_best[:5]
```

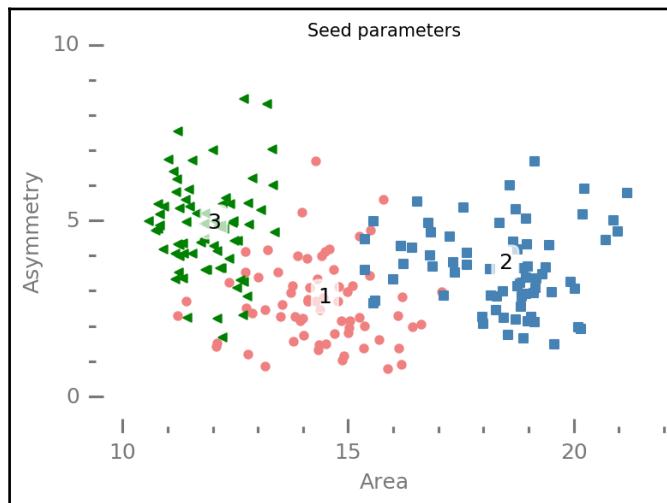
```
array([[ 15.26 ,  2.221],  
       [ 14.88 ,  1.018],  
       [ 14.29 ,  2.699],  
       [ 13.84 ,  2.259],  
       [ 16.14 ,  1.355]])
```

```
seeds.head()
```

	A	P	C	lkern	wkern	asym	lgro	gr
0	15.26	14.84	0.8710	5.763	3.312	2.221	5.220	1
1	14.88	14.57	0.8811	5.554	3.333	1.018	4.956	1
2	14.29	14.09	0.9050	5.291	3.337	2.699	4.825	1
3	13.84	13.94	0.8955	5.324	3.379	2.259	4.805	1
4	16.14	14.99	0.9034	5.658	3.562	1.355	5.175	1

The area (A) and asymmetry (asym) coefficients are the two best parameters to work with according to this selection algorithm. Before we run it through one of the machine learning algorithms for classification, we plot all of the data again, but this time the features are selected by the algorithm:

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.scatter(seeds.A[gr1], seeds.asym[gr1],
           color='LightCoral')
ax.text(seeds.A[gr1].mean(), seeds.asym[gr1].mean(),
        '1', bbox=dict(color='w', alpha=0.7,
                       boxstyle="Round"))
ax.scatter(seeds.A[gr2], seeds.asym[gr2],
           color='SteelBlue',
           marker='s')
ax.text(seeds.A[gr2].mean(), seeds.asym[gr2].mean(),
        '2', bbox=dict(color='w', alpha=0.7,
                       boxstyle="Round"))
ax.scatter(seeds.A[gr3], seeds.asym[gr3],
           color='Green',
           marker='<')
ax.text(seeds.A[gr3].mean(), seeds.asym[gr3].mean(),
        '3', bbox=dict(color='w', alpha=0.7,
                       boxstyle="Round"))
ax.set_xlabel('Area')
ax.set_ylabel('Asymmetry')
ax.set_title('Seed parameters')
despine(ax)
plt.minorticks_on()
ax.locator_params(axis='x', nbins=5)
ax.locator_params(axis='y', nbins=3)
ax.set_xlim(9.6,22)
ax.set_ylim(-0.6,10);
```



Compared with the previous similar plot, the points are more spread out, and perhaps the circles and triangles are more separated with less overlap—it is hard to assess.

Classifying the data

To start classifying the data, we prepare a few things first. We import the SVM module and K-nearest neighbor along with the random forest estimators. Within the SVM module is the **Support Vector Classification (SVC)** estimator, the main estimator for SVM. SVC can be run with different kernels; we will cover the linear, radial basis function, and polynomial kernels. I will give a short explanation of them before we run them.

To visualize the classification, I want to plot the boundaries, and we will use contour lines to do this. For this, we need to create a grid of points and evaluate them with our trained classifier:

```
from sklearn import svm
from sklearn.neighbors import KNeighborsClassifier

res = 0.01
#X, y = X_best[::2], labels[::2]
X, y = X_best, labels
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, res),
                     np.arange(y_min, y_max, res))
```

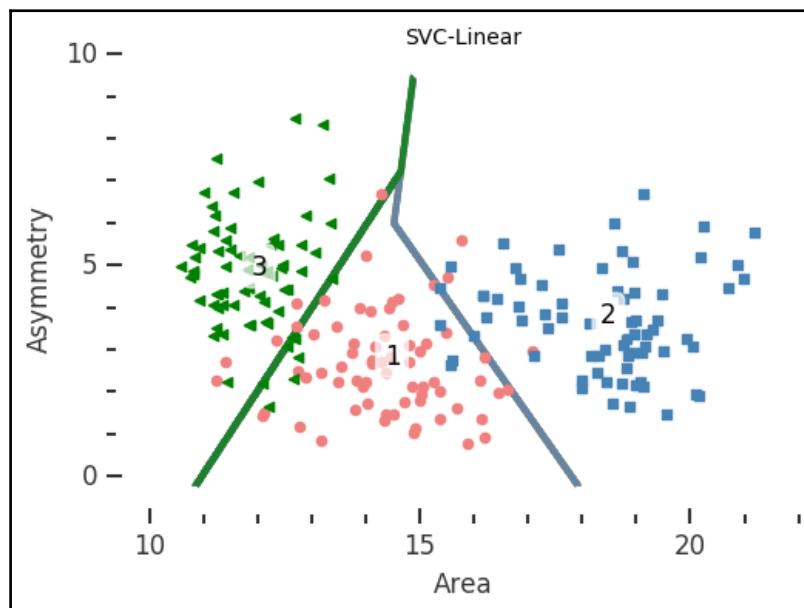
Here, we write a function to draw the results. To draw the boundaries, the x and y grid that we created previously is used. It is passed to the estimator's `predict (xxyy)` method. Here, input is the estimator, output from the machine learning classification model (that is, different SVMs, K-Nearest Neighbor, and Random Forest), and title of the plot. The contour plot draws the boundaries, and you can change `ax.contour` to `ax.contourf` to get filled contours. Now that we have a function to take care of the visualization, we can focus on testing the different models (called kernels):

```
def plot_results(clf, title):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    plt.subplots_adjust(wspace=0.2, hspace=0.4)
    xxyy = np.vstack((xx.flatten(), yy.flatten())).T
    Z = clf.predict(xxyy)
    Z = Z.reshape(xx.shape)
    ax.contour(xx, yy, Z,
               colors=['Green', 'LightCoral', 'SteelBlue'],
               alpha=0.7, zorder=-1)
    ax.scatter(seeds.A[gr1], seeds.asym[gr1],
               color='LightCoral')
    ax.scatter(seeds.A[gr2], seeds.asym[gr2],
               color='SteelBlue', marker='s')
    ax.scatter(seeds.A[gr3], seeds.asym[gr3],
               color='Green', marker='<')
    ax.text(seeds.A[gr1].mean(), seeds.asym[gr1].mean(),
            '1', bbox=dict(color='w', alpha=0.7,
                           boxstyle="Round"))
    ax.text(seeds.A[gr2].mean(), seeds.asym[gr2].mean(),
            '2', bbox=dict(color='w', alpha=0.7,
                           boxstyle="Round"))
    ax.text(seeds.A[gr3].mean(), seeds.asym[gr3].mean(),
            '3', bbox=dict(color='w', alpha=0.7,
                           boxstyle="Round"))
    despine(ax)
    plt.minorticks_on()
    ax.locator_params(axis='x', nbins=5)
    ax.locator_params(axis='y', nbins=3)
    ax.set_xlabel('Area')
    ax.set_ylabel('Asymmetry')
    ax.set_title(title, size=10)
    ax.set_xlim(9.6,22)
    ax.set_ylim(-0.6,10);
```

The SVC linear kernel

One simple kernel in SVC is the linear kernel, which assumes linear boundaries. As input, it takes C , the parameter determining the sensitivity to noisy data; with very noisy data, you can decrease this parameter. To get the linear kernel, run the classification on our data, and plot the results with our function, we run the following:

```
svc = svm.SVC(kernel='linear', C=1.).fit(X, y)
plot_results(svc, 'SVC-Linear')
```

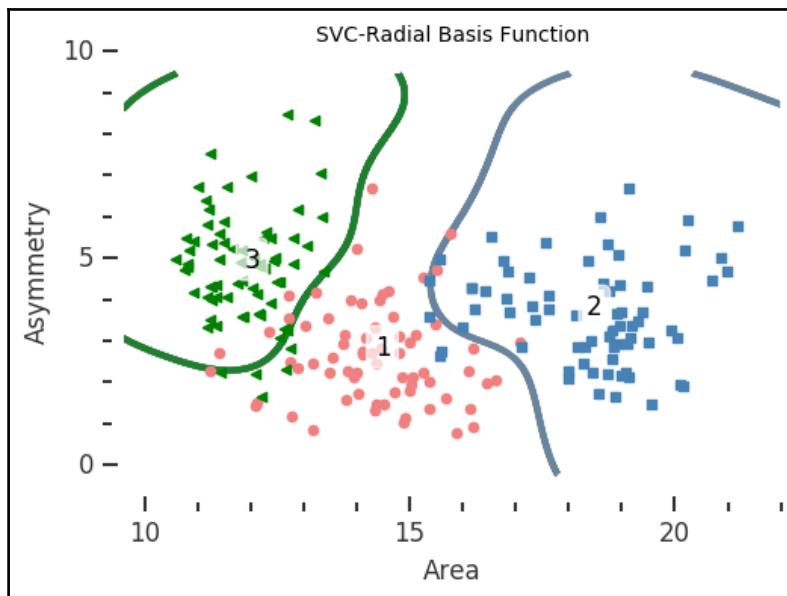


As you can see, the boundaries are linear, and it does a reasonable job in dividing the various points into groups that correspond to the ones created by the researchers.

The SVC Radial Basis Function

The next kernel, the **Radial Basis Function (RBF)**, is the kernel used if no input kernel is given to the SVC call; it is basically a Gaussian kernel. The result is a kernel (region) that is built up by a linear combination of Gaussians. In addition to the C parameter, the gamma parameter can be given here; it is the inverse width of the Gaussian(s), so it gives the steepness of the boundary:

```
rbf_svc = svm.SVC(kernel='rbf', gamma=0.4, C=1.).fit(X, y)
plot_results(rbf_svc, 'SVC-Radial Basis Function')
```

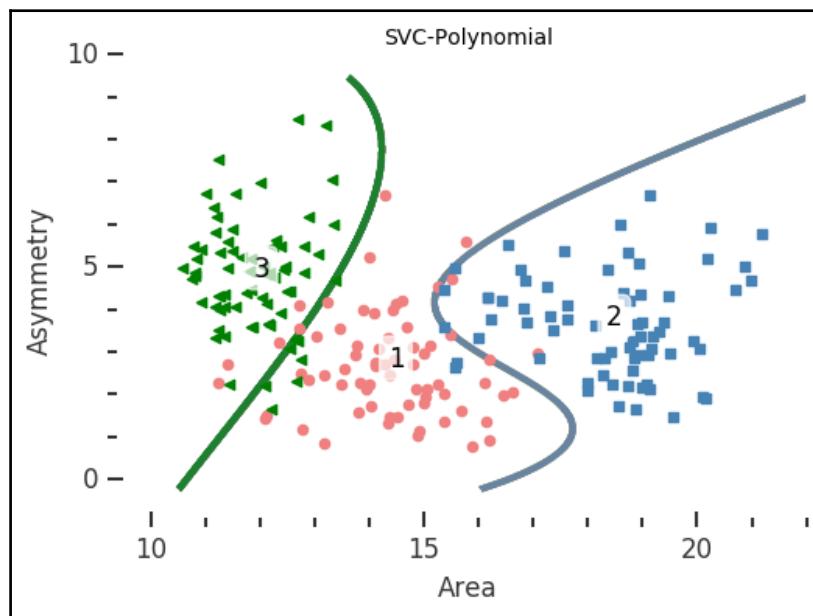


Here, the boundaries are smoother. The borders in the center are roughly the same as with the linear kernel, but they differ going away from the dense regions.

The SVC polynomial

The last SVC kernel that we will cover is the polynomial, and it is exactly what it sounds like, a polynomial. As input, it takes the degree:

```
poly_svc = svm.SVC(kernel='poly', degree=3, C=1.).fit(X, y)
plot_results(poly_svc, 'SVC-Polynomial')
```

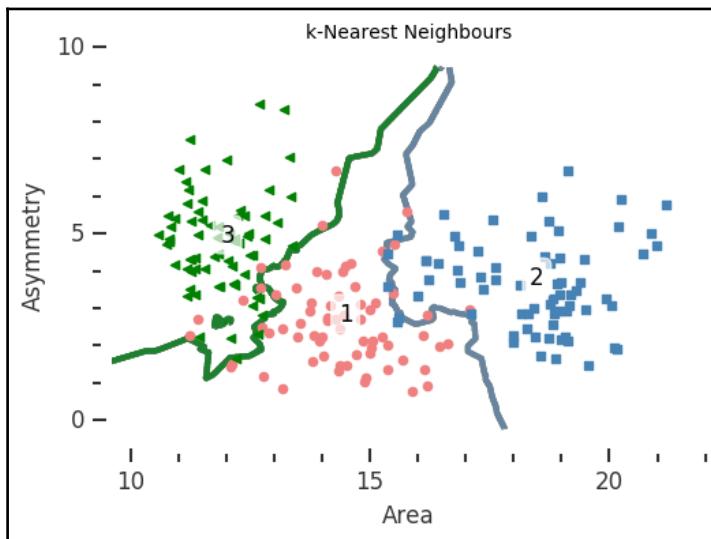


The borders are thus represented by polynomials. I suggest that you try to change the degree to something else and see what happens with the borders.

K-Nearest Neighbour

Now we use the K-Nearest Neighbor. As input, this takes the weight and number of neighbors to compare with. The default weight is one that assumes uniform weights of all n_neighbors nearby points; changing the weights keyword to distance assumes that weights decrease with distance:

```
knn = KNeighborsClassifier(weights = 'uniform', n_neighbors=5).fit(X, y)
plot_results(knn, 'k-Nearest Neighbours')
```

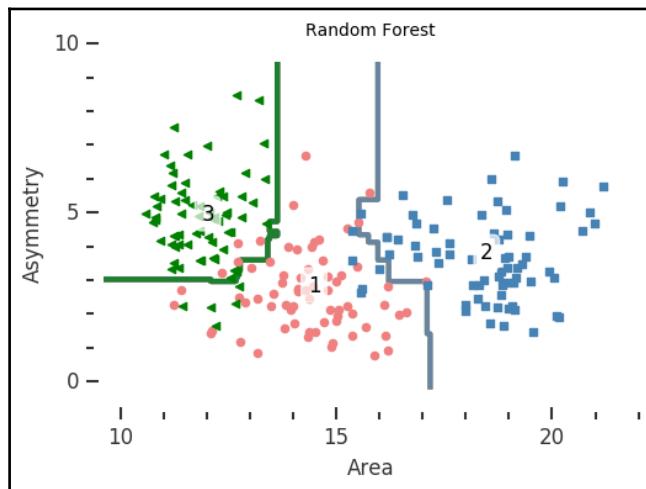


This resembles some of the SVC kernels, but adapted to even smaller changes. Try changing weights to 'distance' and also changing the n_neighbors parameter and see how the results change. What happens when you change n_neighbors to something larger than 30; which of the other classifiers does it almost exactly replicate?

Random Forest

As a last example classifier, we use the Random Forest method. You can think of it as the dendrogram for hierarchical clustering in Chapter 5, *Clustering*; however, each branch is a rule to classify the data here. We give the object three inputs: max_depth, n_estimators, and max_features. The first, max_depth, determines how far each decision tree should go, and n_estimators gives *how many decision trees there are in the forest*. This is not really intuitive, so to show you what it is, first put n_estimators=1, run the code, and look at the output. Then, change to another, higher number and look at the new output:

```
rfc = RandomForestClassifier(  
    max_depth=3,  
    n_estimators=10,  
    max_features='auto').fit(X, y)  
plot_results(rfc, 'Random Forest Classifier')
```



The number of trees in the forest is how many decision trees to build up the classifier with. The resulting plot shows that the random forest classifier is simple yet able to classify fairly complex problems. SVC together with the linear kernel is also simple, but you could imagine how the random forest classifier would be able to classify a more complex problem with fairly low depth and few estimators. I suggest that you take some time and play around with the input parameters for all the classifiers and see how the results change.

Choosing your classifier

The preceding examples show you different results for the SVC (with various kernels), kNN, and Random Forest classifiers. However, when should you use one over the other? In general, try all of the methods on the given problem. The main advantages of SVC are highlighted in these examples—it is very versatile with many different kernels. Decision trees, like the random forest, run the risk of becoming too complex and thus overfitting the data. Also highlighted in the preceding examples, kNN is very good for classifications where the boundary is clearly not linear, as you can see in the results of the preceding image for kNN. There are also other kernels and classifiers. For example, a comparison of 179 classifiers was done for the article, *Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?* (M. Fernandez-Delgado, 2014, JMLR, 15, 3133-3181). All of the common classifiers are covered in the study. However, as stated before, the important thing is to try various classifiers on your data to see what works.

Summary

In this chapter, we looked at various machine learning methods to do regression, clustering, and classification. We compared linear regression using machine learning tools with the same problem in Bayesian inference and standard OLS. Furthermore, we compared the results from the clustering machine learning algorithm, DBSCAN, with what we obtained for hierarchical clustering in Chapter 5, *Clustering*. We concluded by looking at several classification algorithms available in Scikit-learn and how they performed on the same dataset.

With the UCI machine learning repository, finding practice data is not hard. I suggest you visit <http://archive.ics.uci.edu/ml> and look for a dataset to try any of the new things that we have gone through here.

8

Time Series Analysis

Sometimes the data that we will analyze is a variable measured at fixed time intervals; when we have such data, we are talking about a time series. More specifically, at each step of the time series, there is more than one possible outcome and part of the outcome for each step is randomized and might only depend on a few steps back in time. For these reasons, simple linear regression does not work. In time series analysis, we build models to explain the variations in time, which is sometimes referred to as *longitudinal analysis*.

This chapter covers the following topics in time series analysis:

- Time series modeling, its usefulness, and how Pandas handles data
- Various common patterns in time series
- The concept of stationarity and how to test and make your data stationary
- Resampling, smoothing, and calculating rolling statistics
- How to model the known variations and make short forecasts

We start off with some more information about time series and what insights analyzing it can give.

Introduction

Time series analysis is important in several types of situations; it can be used, for example, to describe changes of a variable in time, predict or forecast through modeling the known variations, and then extrapolate these forward in time or assess how certain external stimuli affects a certain time series variable.

There are three main types of modeling and forecasting methods:

- *Extrapolation*, which is the time series analysis we are focusing on in this chapter. This method simply uses historical data from which a model is built and then used to forecast/predict (that is, extrapolate) into the future.
- *Judgemental*, which is used in, for example, decision making and is common where judgment or beliefs (that is, probabilities) need to be incorporated. This can be the case when no historical time series data exists.
- *Econometric*, which is a regression-based method and usually tries to quantify how and to what extent certain variables/events affect the outcome of the time series. As the name suggests, this is sometimes used in economy studies.

There are other methods such as the *Naïve approach* (using the last historical value or values as the forecast); however, we are going to focus on the method most useful for time series analysis in general—the extrapolation method. Most industries use time series analysis at some point in their workflow. Two obvious examples are as follows:

- **Retail**: How much of a certain product should be kept in stock and how much will be sold?
- **Finance**: Managing assets, given the stock data of the previous months, will the stock go up or down tomorrow?



The important thing here is that we are trying to model variations that in part are random, thus some things are impossible to model. Where the time series is fully randomized, the best forecast and model is just a mean and spread.

A time series dataset can be seen as a series of y values at a fixed interval in time, thus no x axis values are part of the data. This can be expressed as follows:

$$y_i = \{y_{i-1}, y_{i-2}, \dots, y_{N-1}, y_N\}$$

Here, each y in the set is just each value at a certain point in time. With these things covered, you are ready to learn about time series analysis in Python with Pandas and statsmodels.

As usual, open Jupyter, start a new notebook, and type in the default imports. I added a few imports as we will use them throughout the chapter. The extra imports, except the default ones (described in Chapter 1, *Tools of the Trade*) are as follows:

```
from pandas.io import data, wb  
import scipy.stats as st
```

```
from statsmodels.tsa import stattools as stt
from statsmodels import tsa
import statsmodels.api as smapi
```

Here, just as mentioned before, you have to replace `pandas.io` with `pandas_datareader` if you have the Pandas version where it is split into a separate package. Furthermore, I will make use of the `despine()` function that we defined earlier, so make sure that you have it in a cell. As you can see, the main package that I will use is `statsmodels`; it has some nice functions to make time series analysis a bit easier. The `statsmodels` developers are working on upgrading the time series analysis to include more advanced functions, so keep an eye out for updates. To start off the analysis, I will read in the first data and go through some unique methods and characteristics that a Pandas time series object has.

Pandas and time series data

In Pandas, there is a certain data type for time series data. This is a normal Pandas `DataFrame` or `Series` where the index is a column of the `datetime` objects. It has to be this kind of object for Pandas to recognize it as dates and for it to understand what to do with the dates. To show you how it works, let us read in a time series dataset.

The first data that we are reading in is the mean measured daily temperature at Fisher River near Dallas, USA from 1st January, 1988 to 31st December, 1991. The data can be downloaded from DataMarket in several formats (<https://datamarket.com/data/set/235d/>), and it can also be acquired from <http://ftp.uni-bayreuth.de/math/statlib/datasets/hipel-mcleod>. Here, I have the data in CSV format. The data comes from the Time Series Data Library (<https://datamarket.com/data/list/?q=provider:tsdl>) and originated in Hipel and McLeod (1994).

The data has two columns: the first with the date and the second with the mean measured temperature on that day. To read in the dates, we need to give a date parsing function to the Pandas CSV data reader, which takes a date in string format and converts it to a `datetime` object, just as we discussed in previous chapters (for example, Chapter 6, *Bayesian Methods*). Opening the data file, you can see that the dates are formatted as year-month-day. Thus, we create a date parsing function for this:

```
dateparse = lambda d: pd.datetime.strptime(d, '%Y-%m-%d')
```

With this date parsing function, we can now read in the data as before:

```
temp = pd.read_csv('data/mean-daily-temperature-fisher-river.csv',
                    parse_dates=['Date'],
                    index_col='Date',
                    date_parser=dateparse)
```

As the columns in the file are named, that is, the first row of the data shows **Date** and **Temp**, we let the reader know that the index column-the column to take as the index-is the column with the **Date** name. We also tell it to parse this column with the date parsing function, which is also given by us. Looking at the first few entries, we can see that it is a full DataFrame object:

```
temp.head()
```

	Temp
Date	
1988-01-01	-23.0
1988-01-02	-20.5
1988-01-03	-22.0
1988-01-04	-30.5
1988-01-05	-31.0

To make our analysis easier and as we are working with a univariate dataset, we can extract only the Pandas series out of it. This is just the column in the DataFrame. With the following, we end up with a Series object:

```
temp = temp.iloc[:,0]
```

The Series object still has the dates as index; in fact, printing out the `index` attribute shows that we have indeed parsed the date as index:

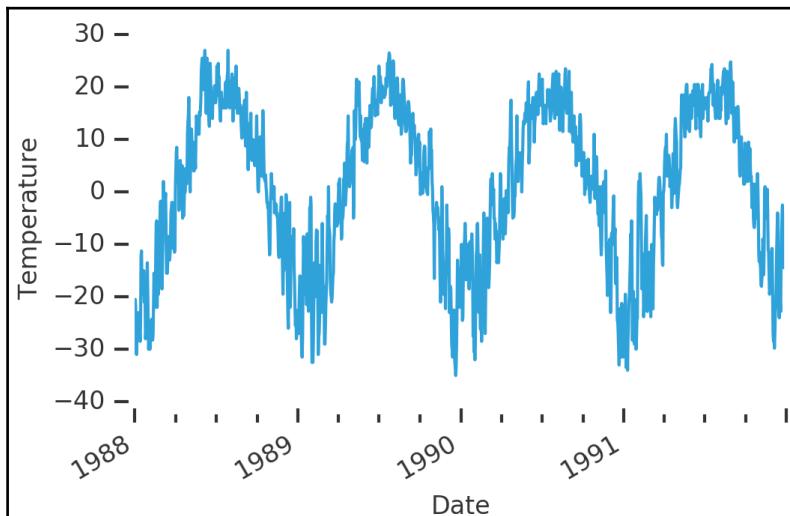
```
temp.index
```

```
DatetimeIndex(['1988-01-01', '1988-01-02', '1988-01-03', '1988-01-04',
                 '1988-01-05', '1988-01-06', '1988-01-07', '1988-01-08',
                 '1988-01-09', '1988-01-10',
                 ...
                 '1991-12-22', '1991-12-23', '1991-12-24', '1991-12-25',
                 '1991-12-26', '1991-12-27', '1991-12-28', '1991-12-29',
                 '1991-12-30', '1991-12-31'],
                dtype='datetime64[ns]', name='Date', length=1461, freq=None)
```

The `dtype='datetime64[ns]'` value shows that we are storing the index as the date with a very high precision. As always, we first visualize the data to see what we are dealing with:

```
temp.plot(lw=1.5)
despine(plt.gca())
plt.gcf().autofmt_xdate()
plt.ylabel('Temperature');
```

As always, these lines simply call the Pandas Series method `plot`, change the line width (`lw`), then get the current axis (`plt.gca()`), which is sent to the `despine()` function, and then set the `y` label:



As you can see, there is a strong pattern in the data. As it is repeated in each year, it is a special type of cyclical pattern, a seasonal pattern. To check some of the statistics, we call the `describe()` method of the Series object that we have:

```
temp.describe()
```

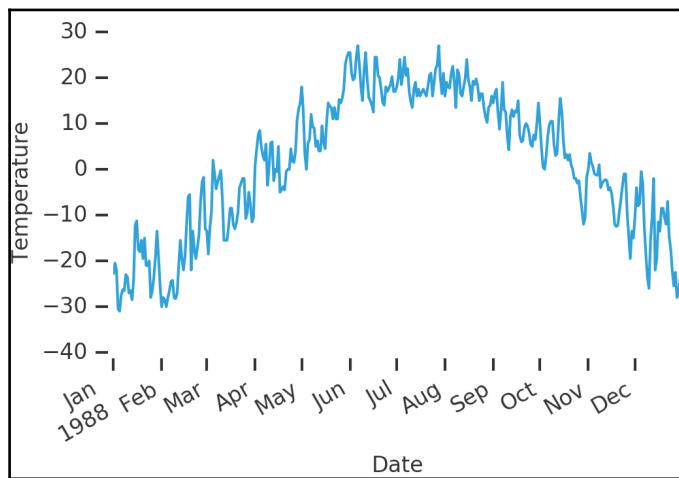
count	1461.000000
mean	0.803320
std	15.154634
min	-35.000000
25%	-11.250000
50%	2.000000
75%	14.500000
max	27.000000
Name:	Temp, dtype: float64

In the printed statistics, we can see that the minimum temperature is **-35**, pretty cold. We also see that there are a lot of measurements, enough for us to work with, and in time series analysis it is very important to have enough data. When we do not have enough data, we have to employ more sophisticated models such as the judgmental models described in the introduction. We now have time series data to work with, and we will start off with looking at how to slice these objects in Pandas.

Indexing and slicing

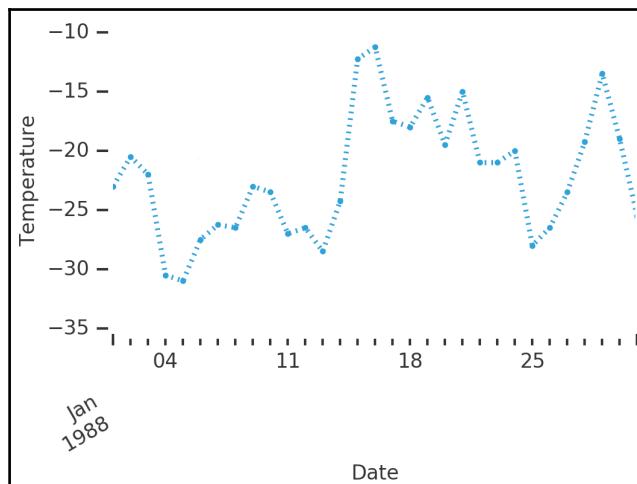
Time series in Pandas can be indexed and sliced in many different ways, but it cannot be with integer indexes. Our index is dates, remember? Thus, to get all the data within the year 1988, we simply index with that year as a string. In the following code, we index it with the year 1988 and then plot the values:

```
temp['1988'].plot(lw=1.5)
despine(plt.gca())
plt.gcf().autofmt_xdate()
plt.minorticks_off()
plt.ylabel('Temperature');
```



The plot shows how the temperature varied over the year 1988, going from almost -30 to roughly +25 and then back to below zero around late October. As you probably suspected, you can also index to a whole month, by just giving the year and month:

```
temp['1988-01'].plot(ls='dotted', marker='.')  
despine(plt.gca())  
plt.gcf().autofmt_xdate()  
plt.ylabel('Temperature');
```



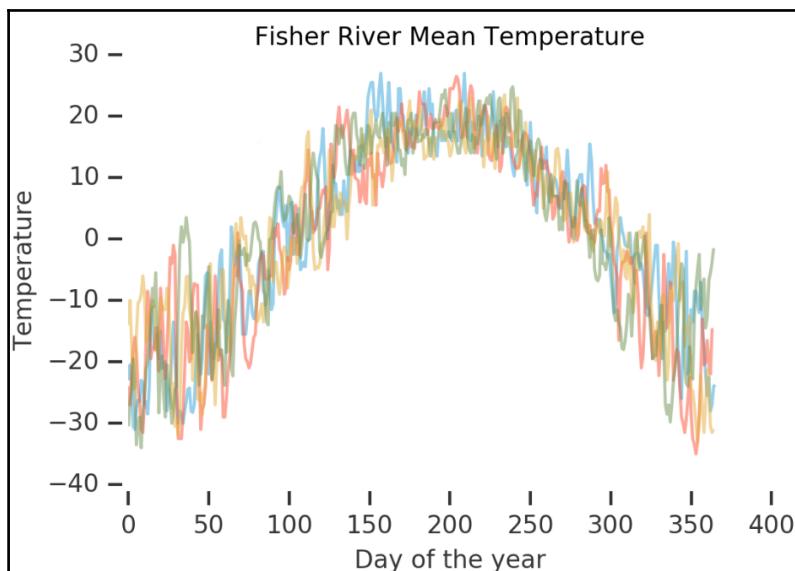
The variation within one month, here January, is quite large, around 20 degrees from minimum to maximum. You can also slice with two indexes, just like a normal array. Try making a plot for a few months by slicing, for example, `temp['1988-06': '1989-06']`.

Just as you can filter out certain values with a normal array, you can filter out certain values in a time series. To only get the values when the temperature was strictly below -25, you do like you always would—do a comparison that returns a Boolean array:

```
temp[temp < -25].head()
```

```
Date
1988-01-04    -30.50
1988-01-05    -31.00
1988-01-06    -27.50
1988-01-07    -26.25
1988-01-08    -26.50
Name: Temp, dtype: float64
```

To finish off this section, make a plot with each year plotted in the same figure as follows:



As could be strongly hinted at from the first plot of this data, it has a seasonal component, which is not at all surprising for such a dataset (outdoor temperature). However, try to make this plot yourself by slicing. Can you also add the months on the *x* axis? Perhaps try to make one for only one month but all years, say April or May. The next step that we will cover is how to manipulate and calculate various estimates on the time series.

Resampling, smoothing, and other estimates

Another useful method to visualize and make some of the initial analysis of the data is resampling, smoothing, and other rolling estimates. When resampling, a frequency keyword needs to be passed to the function. This is a combination of integers and letters, where the letters signify the type of the integer. To give you an idea, some of the frequency specifiers are as follows:

B, business, or D, calendar day

W, weekly

M, calendar month end or MS for start

Q, calendar quarter end or QS for start

A, calendar year end, or AS for start

H, hourly, T, minutely

Most of these can be modified by adding a `B` at the start of the specifier to change it to Business (month, quarter, year, and so on), and there are a few other keywords/descriptors that can be found in the Pandas documentation. Now let's try some of these out in the following examples. As this chapter contains several real-world data examples, which we use to highlight different things, feel free to play around with the data analysis. To resample the data by year, we simply pass an `A` to the `resample()` method:

```
temp.resample('A').head()
```

Date	
1988-12-31	1.138661
1989-12-31	-0.006164
1990-12-31	0.815753
1991-12-31	1.264110

Freq: A-DEC, Name: Temp, dtype: float64

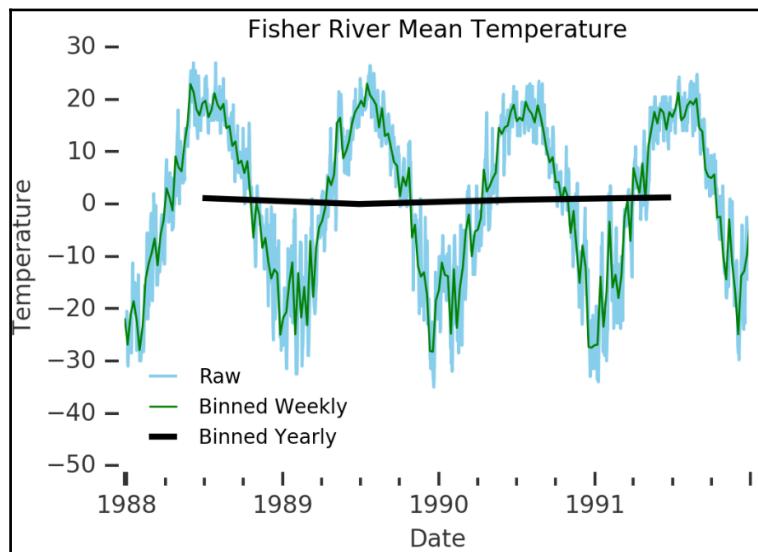
Here, the values are basically the mean of the year, with the label at the end of the year. Now let's make a plot with some of the resampling options to clearly show the variations that happen over these years.



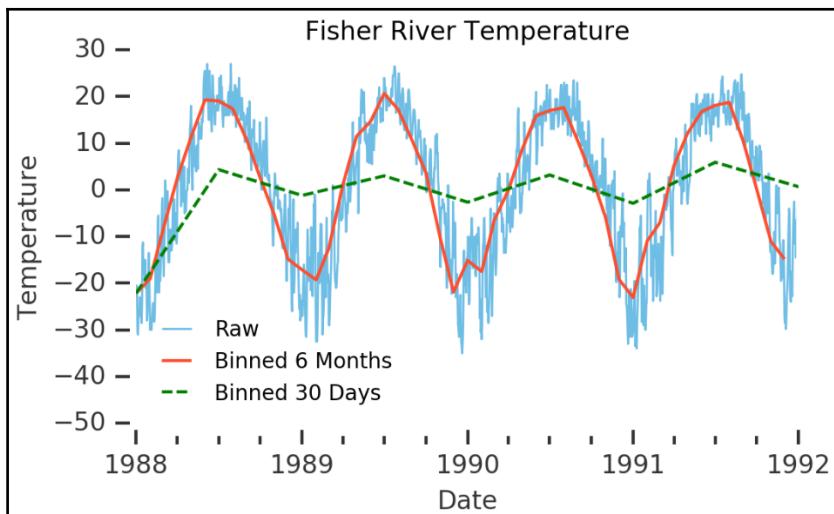
How the `resample()` method works might change a bit in upcoming releases of Pandas. If you are running a version higher than 0.17.1, you should consult the Pandas documentation for more information.

First off, we plot the raw data, then we plot the data resampled to weekly basis, and lastly to yearly basis. However, if we give the frequency descriptor `A` on a yearly basis, it will simply be at the end of the year. It would be nice to show the year-to-year variation where the point is centered not in the beginning of the year but in the middle of the year. To accomplish this, we use the `AS` descriptor, giving us the data resampled over a year with the labels at the start, and then add an offset of roughly half a year with the `loffset='178 D'` keyword:

```
temp.plot(lw=1.5, color='SkyBlue')
temp.resample('W').plot(lw=1, color='Green')
temp.resample('AS', loffset='178 D').plot(color='k')
plt.ylim(-50,30)
plt.ylabel('Temperature')
plt.title('Fisher River Mean Temperature')
plt.legend(['Raw', 'Binned Weekly', 'Binned Yearly'], loc=3)
despine(plt.gca());
```

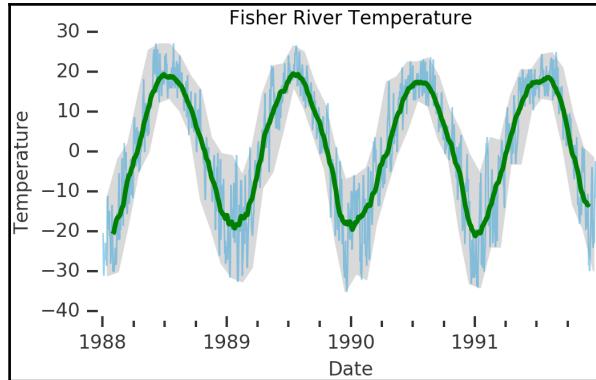


To make the legend more visible, I simply added some space with the `plt.ylim()` function. Now try to make a plot that looks like the following figure, with one month and six months resampling, plotted over the raw data:



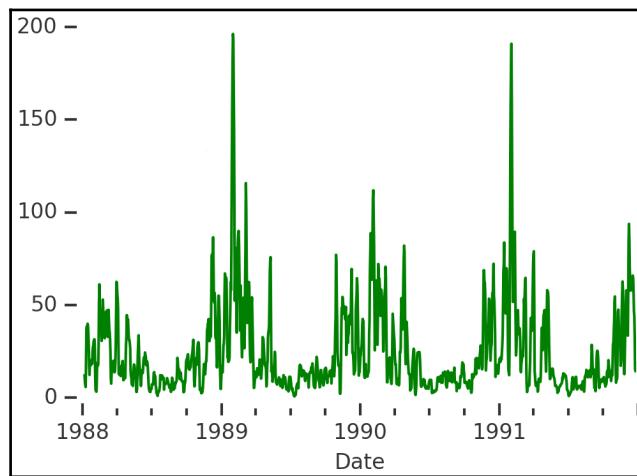
Sometimes we want to calculate a rolling value of something. While the resampling might look like a rolling mean, there is a specific function for it in Pandas. One of the things that we can do with this is combine the rolling mean over the time series and the minimum and maximum values in a region around it to highlight the variation to a nice figure. In the following figure, we plot the rolling mean in a window of 60, meaning that if the data is sampled in days, it will be 60 days. Furthermore, we have told the rolling mean to be centered in the window. To get the minimum and maximum from the raw data, we resample to months, take the minimum and maximum values, and fill the plot between them:

```
temp.plot(lw=1, alpha=0.5)
pd.rolling_mean(temp, center=True, window=60).plot(color='Green')
plt.fill_between(temp.resample('M', label='left',
                               loffset='15 D').index,
                 y1=temp.resample('M', how='max').values,
                 y2=temp.resample('M', how='min').values,
                 color='0.85')
plt.gcf().autofmt_xdate()
plt.ylabel('Temperature')
despine(plt.gca())
plt.title('Fisher River Temperature');
```



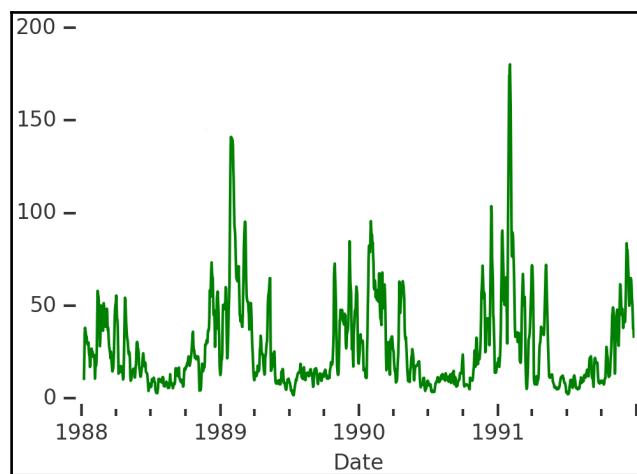
This already looks very good; the rolling mean reproduces the large-scale year-to-year variations of the temperature. While this is a kind of time series analysis and might be enough for first-order analysis and to get a handle of the data, we will look at some more complex methods to model variations. You can calculate other rolling values, such as the covariance:

```
pd.rolling_cov(temp, center=True, window=10).plot(color='Green')
despine(plt.gca());
```



In this case, the covariance is in a window of 10 days and seems very high around the shift of the year. Another rolling value to calculate is the variance:

```
pd.rolling_var(temp, center=True, window=14).plot(color='Green')  
despine(plt.gca());
```



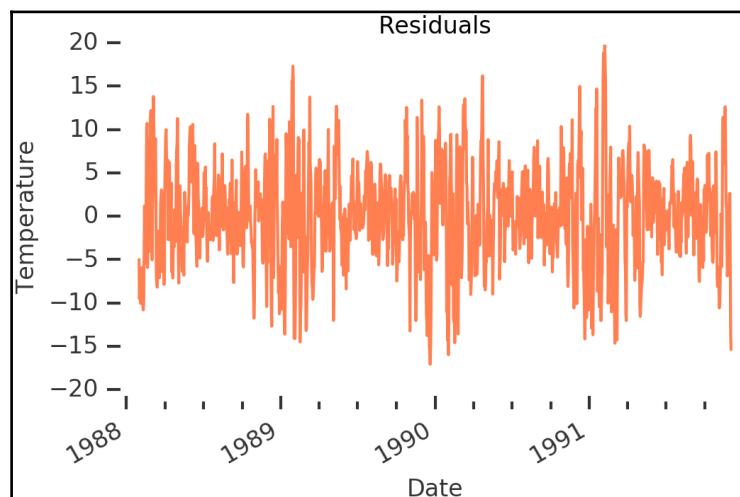
As we shall cover later on, analyzing the variance over time is very important for time series analysis. Try changing the window for both covariance and variance and see how they differ.

We calculated the rolling mean before and saw that it seems to follow the large-scale year-to-year variations of the data. Let's calculate the residuals to subtract this rolling mean from the raw data:

```
temp_residual = temp-pd.rolling_mean(temp, center=True, window=60)
```

Visualizing the residuals, we can see that there is still some periodicity in it. To analyze a time series, we need the data to contain as few of these large-scale patterns as possible:

```
temp_residual.plot(lw=1.5, color='Coral')
despine(plt.gca())
plt.gcf().autofmt_xdate()
plt.title('Residuals')
plt.ylabel('Temperature');
```



Time series analysis is mostly based on the fact that the current value might depend on only a few of the previous values and to a varying extent. So to analyze the data, we need to get rid of these. This naturally leads us to the next topic—stationarity. In the next section, we will discuss this, show you how to test if your data is stationary, and a couple of ways to make it stationary if it is not.

Stationarity

Most time series modeling depends on the data being stationary. The easiest definition of a stationary time series is that most of its statistical characteristics are all roughly constant over time. For statistical characteristics, the mean, variance, and autocorrelation are most commonly mentioned. For this to be true, we cannot have any trends, that is, data cannot increase monotonically over time. There cannot be long cycles of ups and downs either. If any of these things are true, the mean will change over time and the variance too. There are other more complex mathematical tests, such as the following (Augmented) Dickey-Fuller test. We focus on this test here as it is conveniently available in statsmodels.

The fact is that when doing time series analysis, we first need to make sure that the data is stationary. The easiest way to check whether your data is stationary in Python is to do an Augmented Dickey-Fuller test. This is a statistical test that estimates if your dataset is stationary. The statsmodels package has a function that tests this and sends back the diagnostics. The value of the test (we will call it the *ADF* value) needs to be compared to the critical values at 1, 5, and 10%. If the ADF value is below the critical value at 5% and the p-value (yes, the statistical p-value) is small, around less than 0.05, we can reject the null hypothesis that the data is not stationary at a 95% confidence level.

To make it easier to figure out if the results show whether the time series is stationary or not, let's write a small function that runs the function and summarizes the output:

```
def is_stationary(df, maxlag=15, autolag=None, regression='ct'):  
    """Run the Augmented Dickey-Fuller test from Statsmodels  
    and print output.  
    """  
    outpt = stt.adfuller(df,maxlag=maxlag, autolag=autolag,  
                         regression=regression)  
    print('adf\t\t {0:.3f}'.format(outpt[0]))  
    print('p\t\t {0:.3g}'.format(outpt[1]))  
    print('crit. val.\t 1%: {0:.3f},\n          5%: {1:.3f}, 10%: {2:.3f}'.format(outpt[4]["1%"],  
                                         outpt[4]["5%"], outpt[4]["10%"]))  
    print('stationary?\t {0}'.format(['true', 'false'][  
                                     [outpt[0]>outpt[4]['5%']])))  
    return outpt
```

We are now ready to test the stationarity of a dataset, so let's read one in.

This dataset can be downloaded from DataMarket (<https://datamarket.com/data/set/22n4/>). The data comes from the Time Series Data Library (<https://datamarket.com/data/list/?q=provider:tsdl>) and originated in Abraham and Ledolter (1983). It shows the monthly car sales in Quebec from 1960 to 1968. As before, we use a date parser to get a Pandas time series DataFrame directly:

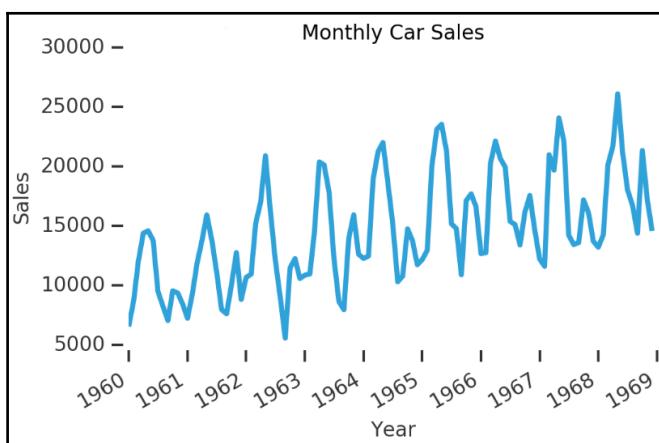
```
carsales = pd.read_csv('data/monthly-car-sales\  
                      -in-quebec-1960.csv',  
                      parse_dates=['Month'],  
                      index_col='Month',  
                      date_parser=lambda d: \  
                      pd.datetime.strptime(d, '%Y-%m'))
```

To go over to a Pandas Series object instead of DataFrame, we do the same thing as before:

```
carsales = carsales.iloc[:,0]
```

Plotting the dataset shows some interesting things. The data has some strong seasonal trends, that is, cyclical patterns within each year. It also has a slow upward trend but more on that later:

```
plt.plot(carsales)  
despine(plt.gca())  
plt.gcf().autofmt_xdate()  
plt.xlim('1960','1969')  
plt.xlabel('Year')  
plt.ylabel('Sales')  
plt.title('Monthly Car Sales');
```



We can now run our small wrapper to test if it is stationary:

```
is_stationary(carsales);
```

adf	-1.673
p	0.763
crit. val.	1%: -4.060, 5%: -3.459, 10%: -3.155
stationary?	false

It is not! Well, this is not a huge surprise as it had all those patterns. This takes us to the next section, where we will look at various patterns and components that time series is made up of.

Patterns and components

In time series, there are mainly four different patterns or components:

- **Trend:** A slow but significant change of the values over time
- **Season:** A change that is cyclical and has a period of less than one year
- **Cycle:** A change that is cyclical and has a period of longer than one year
- **Random:** A component that is random; the best model for purely random data is the mean, given that it has a distribution corresponding to the normal distribution

Thus, before we can analyze our data, it needs to be stationary, and for it to be stationary, we need to take care of the patterns: trend, season, and cycle. The analysis that you will perform will be on part of the time series that does not fit into any of these patterns, with the random component being part of the uncertainty of the model.

Decomposing components

One method of taking care of the various components and making the time series stationary is decomposing. There are different ways of identifying the components; in statsmodels, there is a function to decompose all of them in one go. So let's import it and run our time series through it:

```
from statsmodels.tsa.seasonal import seasonal_decompose  
carsales_decomp = seasonal_decompose(carsales, freq=12)
```

The function takes a frequency as input; this relates to the season so input the seasonal period that you think your data has. In this case, I took 12 as, by looking at the data, it seems like a yearly period. The returned object contains several attributes that are Pandas Series, so let's extract them from the returned object:

```
carsales_trend = carsales_decomp.trend  
carsales_seasonal = carsales_decomp.seasonal  
carsales_residual = carsales_decomp.resid
```

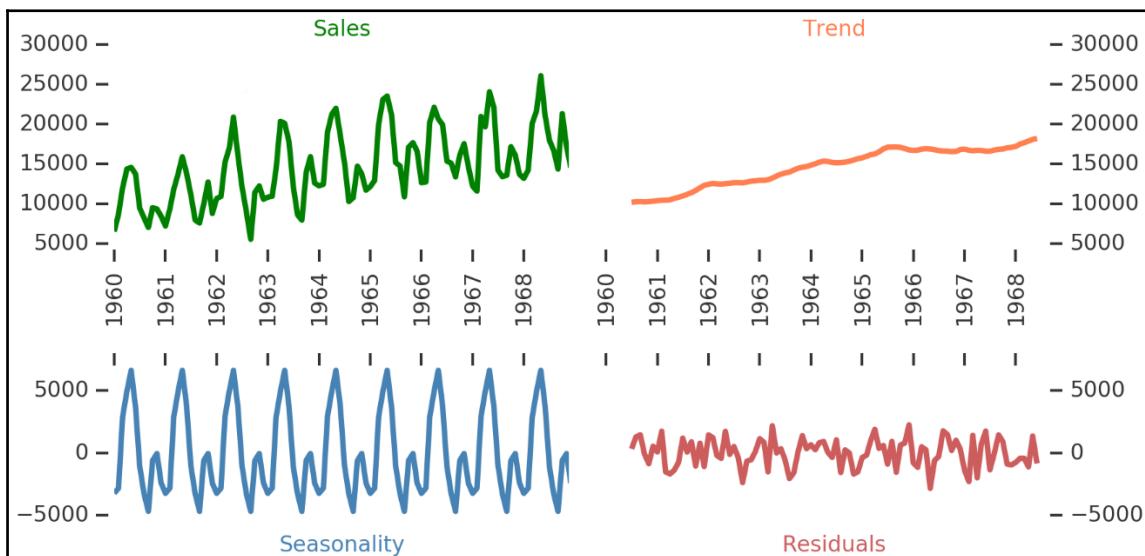
To visualize what these different components are now, we plot them in a figure:

```
def change_plot(ax):  
    despine(ax)  
    ax.locator_params(axis='y', nbins=5)  
    plt.setp(ax.get_xticklabels(), rotation=90, ha='center')  
  
plt.figure(figsize=(9, 4.5))  
  
plt.subplot(221)  
plt.plot(carsales, color='Green')  
change_plot(plt.gca())  
plt.title('Sales', color='Green')  
x1 = plt.xlim()  
y1 = plt.ylim()  
  
plt.subplot(222)  
plt.plot(carsales.index, carsales_trend,  
         color='Coral')  
change_plot(plt.gca())  
plt.title('Trend', color='Coral')  
plt.gca().yaxis.tick_right()  
plt.gca().yaxis.set_label_position("right")  
plt.xlim(x1)  
plt.ylim(y1)  
  
plt.subplot(223)  
plt.plot(carsales.index, carsales_seasonal,
```

```
        color='SteelBlue')
change_plot(plt.gca())
plt.gca().xaxis.tick_top()
plt.gca().xaxis.set_major_formatter(plt.NullFormatter())
plt.xlabel('Seasonality', color='SteelBlue', labelpad=-20)
plt.xlim(xl)
plt.ylim((-8000,8000))

plt.subplot(224)
plt.plot(carsales.index,carsales_residual,
          color='IndianRed')
change_plot(plt.gca())
plt.xlim(xl)
plt.gca().yaxis.tick_right()
plt.gca().yaxis.set_label_position("right")
plt.gca().xaxis.tick_top()
plt.gca().xaxis.set_major_formatter(plt.NullFormatter())
plt.ylim((-8000,8000))
plt.xlabel('Residuals', color='IndianRed', labelpad=-20)

plt.tight_layout()
plt.subplots_adjust(hspace=0.55)
```



Here, we can see all the different components—the raw sales are shown on the upper left, while the general trend of the data is shown on the upper right. The identified seasonality is on the lower left and the residuals after breaking these two out is on the lower right. The seasonal component has multiple periodic peaks. This seasonal component is really interesting as it accounts for a majority of the annual variation in sales. Let's take a closer look at it by plotting the detrended data (that is, the trend subtracted off the raw sales figures) and seasonality changes (during one year):

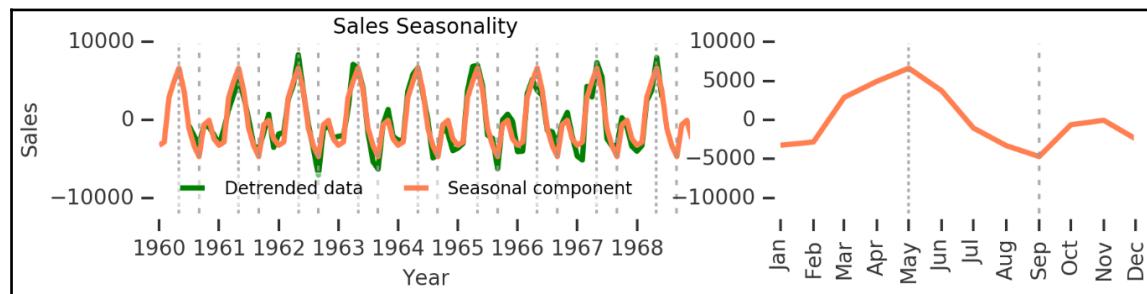
```
fig = plt.figure(figsize=(7,1.5))

ax1 = fig.add_axes([0.1,0.1,0.6,0.9])
ax1.plot(carsales-carsales_trend,
          color='Green', label='Detrended data')
ax1.plot(carsales_seasonal,
          color='Coral', label='Seasonal component')
kwrds=dict(lw=1.5, color='0.6', alpha=0.8)
d1 = pd.datetime(1960,9,1)
dd = pd.Timedelta('365 Days')
[ax1.axvline(d1+dd*i, dashes=(3,5),**kwrds) for i in range(9)]
d2 = pd.datetime(1960,5,1)
[ax1.axvline(d2+dd*i, dashes=(2,2),**kwrds) for i in range(9)]
ax1.set_ylim((-12000,10000))

ax1.locator_params(axis='y', nbins=4)
ax1.set_xlabel('Year')
ax1.set_title('Sales Seasonality')
ax1.set_ylabel('Sales')
ax1.legend(loc=0, ncol=2, frameon=True);

ax2 = fig.add_axes([0.8,0.1,0.4,0.9])
ax2.plot(carsales_seasonal['1960':'1960'],
          color='Coral', label='Seasonal component')
ax2.set_ylim((-12000,10000))
[ax2.axvline(d1+dd*i, dashes=(3,5),**kwrds) for i in range(1)]
d2 = pd.datetime(1960,5,1)
[ax2.axvline(d2+dd*i, dashes=(2,2),**kwrds) for i in range(1)]
despine([ax1, ax2])

import matplotlib.dates as mdates
yrsfmt = mdates.DateFormatter('%b')
ax2.xaxis.set_major_formatter(yrsfmt)
labels = ax2.get_xticklabels()
plt.setp(labels, rotation=90);
```



As you can see here, the seasonal component is really significant. While this is rather obvious for this dataset, it is a good start in time series analysis when you can break the data up into pieces like this, and it gives a wealth of insight into what is happening. Let's save this seasonal component for one year:

```
carsales_seasonal_component = carsales_seasonal['1960'].values
```

The residuals, which are left after subtracting the trend and seasonality, should now be stationary, right? They look like they are stationary. Let's check with our wrapper function. To do this, we first need to get rid of NaN values:

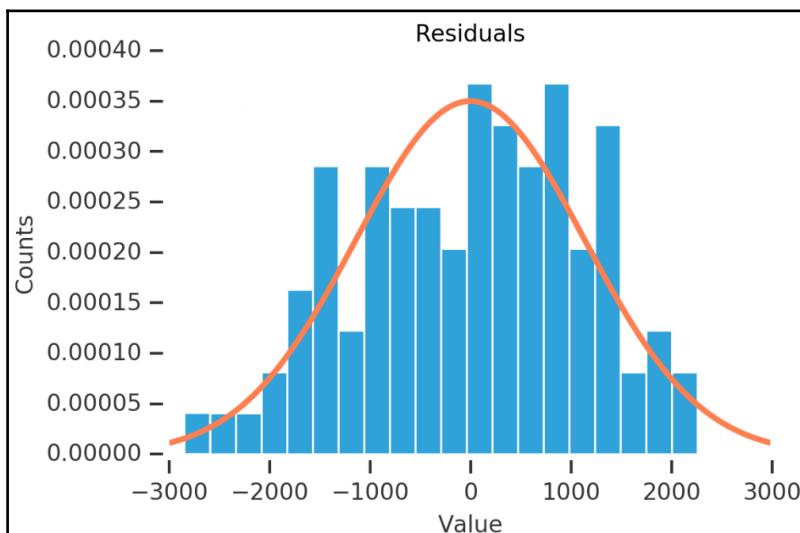
```
carsales_residual.dropna(inplace=True)
is_stationary(carsales_residual());
```

adf	-4.501
p	0.0015
crit. val.	1%: -4.072, 5%: -3.465, 10%: -3.159
stationary?	true

It is now stationary; this would mean that we can continue to analyze the time series and start modeling it. There are some possible bugs in the current version of statsmodels when trying to re-include the seasonal and trend components in the models of the residuals. Due to this, we have tried to do it in a different way.

Before we start making the time series models, I want to look a bit more at the residuals. We can use what we have learned in the first few chapters and check whether the residuals are normally distributed. First, we plot the histogram for the values and overplot a fitted Gaussian probability density distribution:

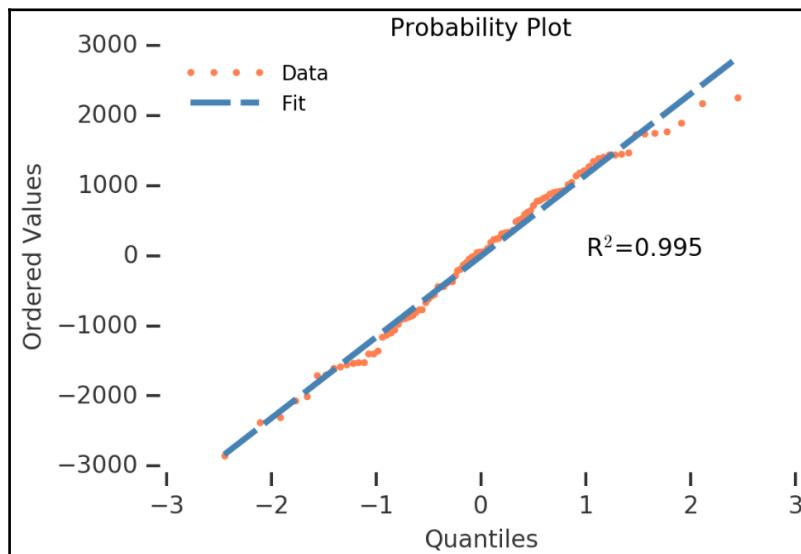
```
loc, shape = st.norm.fit(carsales_residual)
x=range(-3000,3000)
y = st.norm.pdf(x, loc, shape)
n, bins, patches = plt.hist(carsales_residual, bins=20, normed=True)
plt.plot(x,y, color='Coral')
despine(plt.gca())
plt.title('Residuals')
plt.xlabel('Value'); plt.ylabel('Counts');
```



Another check that we used was the probability plot, so let's run it on this as well. However, instead of letting it plot the figure, as we did previously, we will do it ourselves. To do this, we catch the output of `probplot()` and do not give it any axes or plotting functions as input. After we get the variables, we just plot them and a line with the given coefficients:

```
(osm,osr), (slope, intercept, r) = st.probplot(carsales_residual,
                                              dist='norm', fit=True)
line_func = lambda x: slope*x + intercept
plt.plot(osm,osr,
         '.', label='Data', color='Coral')
plt.plot(osm, line_func(osm),
         color='SteelBlue',
```

```
dashes=(20,5), label='Fit')
plt.xlabel('Quantiles'); plt.ylabel('Ordered Values')
despine(plt.gca())
plt.text(1, -14, 'R$^2$={0:.3f}'.format(r))
plt.title('Probability Plot')
plt.legend(loc='best', numpoints=4, handlelength=4);
```

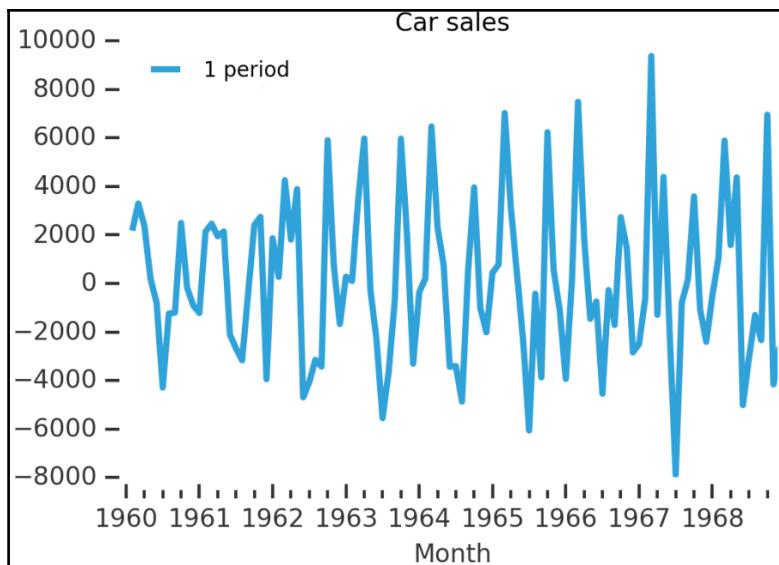


The residuals look like they are normally distributed—the high R^2 value also shows that it is statistically significant. Now that we have checked the residuals of the automatic decomposition, we can go over to the next method of making the data stationary.

Differencing

With differencing, we simply take the difference between two adjacent values. To do this, there is a convenient `diff()` method in Pandas. The following plot shows you the difference with the data shifted by one period:

```
carsales.diff(1).plot(label='1 period', title='Carsales')
plt.legend(loc='best')
despine=plt.gca()
```



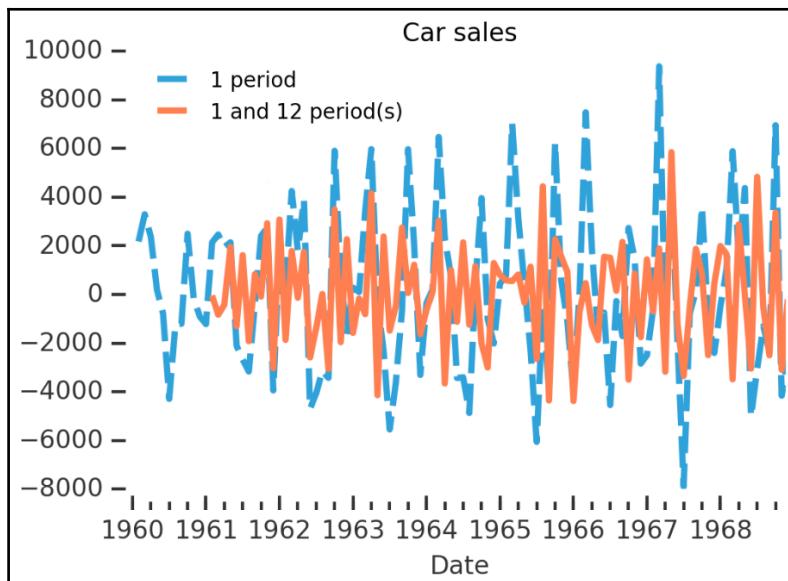
Remember that this is on the raw data—the data with the strong trend and seasonality. While the trend has disappeared, it seems that some of the seasonality is still there; however, let's check whether this is a stationary time series:

```
is_stationary(carsales.diff(1).dropna())
```

adf	-3.124
p	0.101
crit. val.	1%: -4.061, 5%: -3.459, 10%: -3.156
stationary?	false

No, the p-value is higher than 0.05 and the ADF value is higher than at least 5%, so we cannot reject the null hypothesis. Let's run `diff()` again, but with both 1 and 12 periods (that is, 12 months, one year):

```
carsales.diff(1).plot(label='1 period', title='Carsales',
                      dashes=(15, 5))
carsales.diff(1).diff(12).plot(label='1 and 12 period(s)',
                               color='Coral')
plt.legend(loc='best')
despine(plt.gca())
plt.xlabel('Date')
```



It is very hard to judge from this whether it is more or less stationary. We have to run the wrapper on the output to check:

```
is_stationary(carsales.diff(1).diff(12).dropna());
```

adf	-3.875
p	0.0131
crit. val.	1%: -4.077, 5%: -3.467, 10%: -3.160
stationary?	true

This is much better; we seem to have gotten rid of the seasonal and trend components.

I encourage you to use the first example dataset and check some of the things that we covered in this section. How do the various cyclic/seasonal components decompose? What values do you have to use for it to work? In the next section, we will go through some of the general models for time series and how they are used in statsmodels.

Time series models

Modeling time series can become very complex; here, we will go through some of the most employed models one by one and explain some of the ideas behind them. We will start with the autoregressive model, continue with the moving average model, and finish off with the combined autoregressive integrated moving average model. To start off this section, import the statsmodel time series model framework:

```
from statsmodels.tsa.arima_model import ARIMA
```

The ARIMA function takes a Pandas time series and model parameters as input and sends back a model object. To use a combination of the decomposition and differencing method in order to make the time series stationary, I first removed the seasonal component broken out by the statsmodels function and then took the first difference and checked whether it was stationary:

```
is_stationary((carsales-carsales_seasonal).diff(1).dropna());
```

adf	-3.611
p	0.0289
crit. val.	1%: -4.061, 5%: -3.459, 10%: -3.156
stationary?	true

It is stationary—the ADF value is lower than the 5% critical value and p-value is smaller than 0.05. I will save these steps in separate data structures to be used in the modeling:

```
ts = carsales-carsales_seasonal
tsdiff = ts.diff(1)
```

We are now ready to go through the various models.

Autoregressive – AR

For an autoregressive model, we use the values of a number of previous steps to model a value. The important parameter is how many previous steps to use for the model; here, this is parameter p . There are ways to estimate the p parameter beforehand, but sometimes a normal fitting routine of running models with different values and checking against available data is a good way to choose between parameters. The $AR(p)$ (AR of p) model can be expressed in a simple way as follows:

$$y_i = a_0 + \sum_{j=1}^p a_j y_{i-j} + \epsilon$$

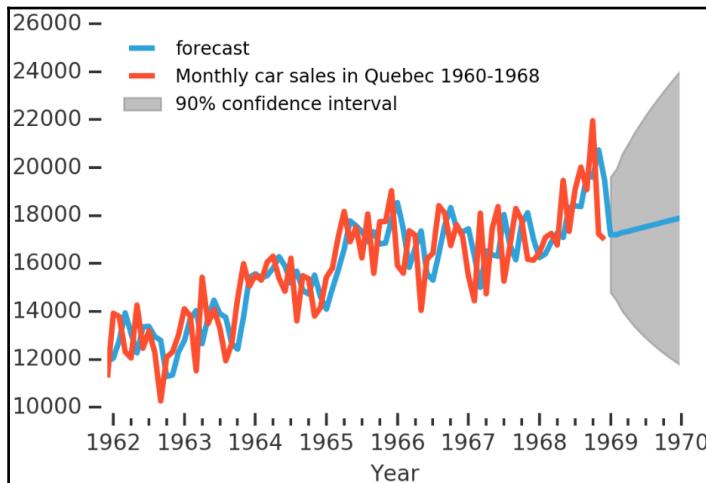
Here, j runs from 1 to p . Note that it shows that the current value y_i is a function of p previous values and a random/uncertainty contribution, ϵ . That is, each value has a part that we can model and a random contribution that we want to minimize. Furthermore, it is important to notice the a parameter-these parameters/coefficients (sometimes also called weights) can be tweaked through fitting to make the model better at reproducing values. They basically control how much each of the previous values matter for the model. While the p parameter is important, it is also important to fit the model to the available historical data to tweak the parameters.

To run this model, we simply give the `order` variable as input to the ARIMA function. It should be a tuple with three values-the first value gives the p value, the second gives the middle value d for how many times to difference the data, and the third and last input is the q for the moving of the average model. To run an AR model with $p=1$ and $d=1$ and then fit it to the data, we simply run the following in a cell:

```
model = ARIMA(ts, order=(1, 1, 0))
arres = model.fit()
```

Now we have fitted the parameters that we showed in the preceding equation. To visualize the fit, there is a convenient function in the fit result object, `arres` in this case:

```
arres.plot_predict(start='1961-12-01', end='1970-01-01', alpha=0.10)
plt.legend(loc='upper left')
despine(plt.gca())
plt.xlabel('Year')
print(arres.aic, arres.bic)
```



You can clearly see how the AR model kind of shifts the existing historical data when it forecasts them; the important thing is the prediction that goes beyond the sample. The 90% confidence interval is also plotted in the grey area; the prediction is mainly the first few values. After this, it converges to a constant trend. In the last line of the preceding code, I also print the **AIC** and **BIC** (`print(arres.aic, arres.bic)`), which is the **Akaike Information Criterion** and **Bayesian Information Criterion**. They are both used as an estimate of how good one model is with respect to another. The lower these values are in comparison to what another model has, the better the model (relatively). In this case, the printout is 1870.3331809826666 and 1878.35166749.

While this model is not perfect, it gives you some estimate of the *future* sales. In the next section, we will create a moving average model.

Moving average – MA

In the moving average model, an average of the previous q values is calculated (called μ) and the random contribution is modeled in the same way as the autoregressive model, assuming that the current value is a function of the mean of q previous values and a small random variation modified with parameters/coefficients, here b . This can be expressed as follows:

$$y_i = \mu + \epsilon_i + \sum_{j=1}^q b_j \epsilon_{i-j}$$

Here, j runs from 1 to q and the average of the previous values can be expressed as follows:

$$\mu = \frac{1}{q} \sum_{j=1}^q y_{i-j}$$

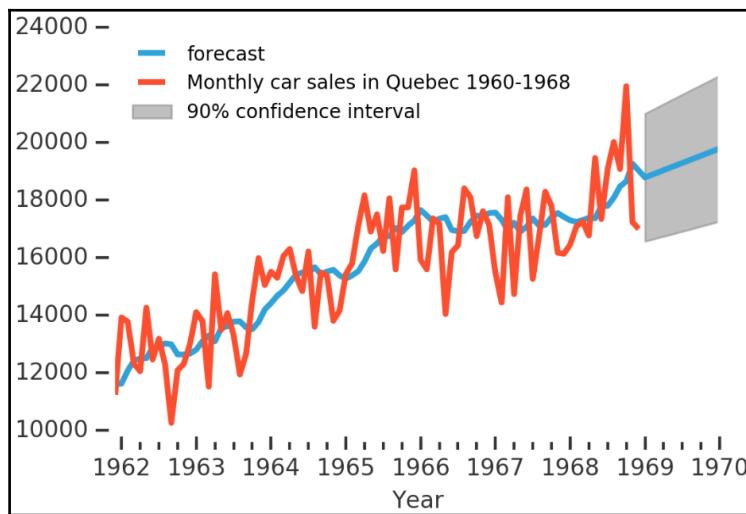
This model assumes in essence that the current value is modeled well by an average of the q previous values, thus it is called the moving average model.

Just like before, we initiate the model with the ARIMA function, but this time with different input so that we can use the MA model:

```
model = ARIMA(ts, order=(0, 1, 1))
mares = model.fit()
```

Plotting this and printing the AIC and BIC shows that it is actually a better model to predict future values:

```
mares.plot_predict(start='1961-12-01', end='1970-01-01', alpha=0.10)
plt.legend(loc='upper left')
despine(plt.gca())
plt.xlabel('Year')
print(mares.aic, mares.bic)
```



The AIC and BIC are 1853.0753124033156 and 1861.09379891, which is slightly lower than the AR model. Judging by the look of it, this model is better at extrapolating the rough trend of the historical data into the future. In the next section, we will make a compound model of both AR and MA models. Before we do this, I will show you two ways of selecting the p and q parameters.

Selecting p and q

The p and q parameters of the AR and MA models should be selected based on how well the resulting model fits the historical data. The AIC or BIC should be compared between the results and the one with the lowest value should be chosen.

Automatic function

The statsmodels package of course has a convenience function for this. We will see how to run this on the data:

```
tsa.stattools.arma_order_select_ic(tsdiff.dropna(), max_ar=2, max_ma=2,
                                    ic='aic')
```

```
{'aic':          0           1           2
 0  1893.258581  1853.075312  1853.070371
 1  1870.333181  1852.567740  1853.880900
 2  1866.617420  1853.644132      NaN, 'aic_min_order': (1, 1)}
```

The output that it prints shows that we should use AR(1) and MA(1) models, and as the input to this is `tsdiff`, d should be 1 as well.

The (Partial) AutoCorrelation Function

While the convenient automatic function does a good job, it runs the whole modeling for various parameter inputs. Another way of estimating p and q is plotting the **autocorrelation function (ACF)** and **partial autocorrelation function (PACF)**. To do this, we first compute them:

```
acf = stt.acf(tsdiff.dropna(), nlags=10)
pacf = stt.pacf(tsdiff.dropna(), nlags=10)
```

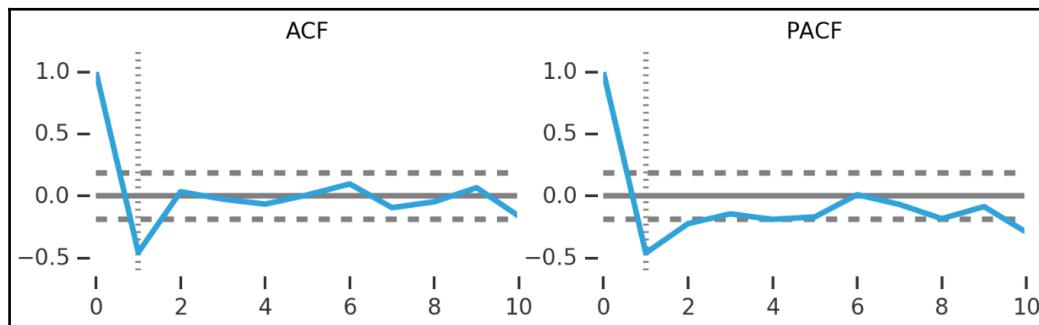
After this, we can plot both and also the critical limits. The limits for the values should be 5% of the (partial) autocorrelation for a stationary time series, which is $1.96/(N-d)$, where N is the number of data points and d is the number of times you have differenced the data. Let's plot the ACF and PACF. In this figure, I also plotted the values suggested by the automatic routine, 1 and 1:

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 2))
ax1.axhline(y=0, color='gray')
ax1.axhline(y=-1.96/ (len(ts)-1)**.5,
            linestyle='--', color='gray')
ax1.axhline(y=1.96/ (len(ts)-1) **.5,
            linestyle='--', color='gray')
ax1.axvline(x=1, ls=':', color='gray')
ax1.plot(acf)
ax1.set_title('ACF')

ax2.axhline(y=0, color='gray')
```

```
ax2.axhline(y=-1.96/ (len(ts)-1) **.5,
             linestyle='--',color='gray')
ax2.axhline(y=1.96/ (len(ts)-1) **.5,
             linestyle='--',color='gray')
ax2.axvline(x=1,ls=':',color='gray')
ax2.plot(pacf)
ax2.set_title('PACF')

despine([ax1,ax2])
```



As you can see, both the curves cross the critical limit at $p=1$ and $q=1$. This is how you can also estimate the p and q parameters, by checking where they cross the critical value and stabilize within the limits. Now we are ready to run the compound model.

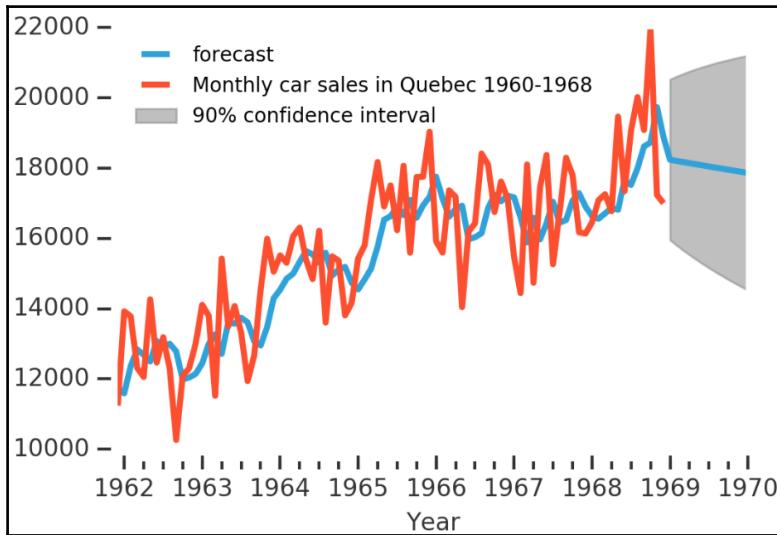
Autoregressive Integrated Moving Average – ARIMA

The last model that we will look at is the **autoregressive integrated moving average (ARIMA)** model. Just as the name suggests, it is a combination of both the previous models. It is run with the same function as before, but for the input tuple, we use all of them. To illustrate some of the functionality, I will run it with a different value of d than the preceding models:

```
model = ARIMA(ts, order=(1, 0, 1))
arimares = model.fit()
```

The following figure shows you the prediction and estimate for the historical values:

```
arimares.plot_predict(start='1961-12-01', end='1970-01-01', alpha=0.10)
plt.legend(loc='upper left')
despine(plt.gca())
plt.xlabel('Year')
print(arimares.aic, arimares.bic)
```



The AIC and BIC are 1880.0061559512924 and 1890.73468086, which is slightly worse than the previous MA model. I suggest that you try to change the input p , d , and q parameters, perhaps to (1,1,1). Play around with different values and difference the data one extra time to see what happens.

Summary

In this chapter, we looked at the many and interesting aspects of time series analysis in Python with Pandas and statsmodels, how they handle the data, and some of the basic manipulation functions that are available. We also looked at the concept of stationarity, how to test your time series for it, and how to transform a non-stationary series into a stationary one. You also found out the various patterns and components that time series can be built up by, and finally, we went through how to create ARIMA models and predict future values based on previous historical data.

This chapter concludes the book. We have covered many different analysis techniques and general statistical knowledge and how to use them in Python to your benefit. With the knowledge in this book, you can start exploring data, any kind of data. In addition to these chapters, there is an appendix. In Appendix, *More on Jupyter Notebook and matplotlib Styles*, I will look at Jupyter Notebook tips and extensions (plugins). I will also provide some links to further resources, including various data repositories for you to find data to download and create your own hypothesis to test.

As mentioned before, it is important to take the time to play around with data and try different algorithms and compare the results. I hope that you have also realized that much of the work in data analysis happens before actually applying the analysis method/algorithm to the data and making a figure with the results, and that making good figures that show all the results without getting bloated is very difficult. In general, with the analysis of real-world data, the process is difficult, and when done, you have to present it all in an easy way for everyone else to understand. With the content taught in this book, you should be able to produce a solid analysis of almost any data and appealing figures in your report that clearly highlight your results.

More on Jupyter Notebook and matplotlib Styles

In this appendix, we will cover several things that will help you when doing data analysis in Jupyter Notebook and compiling reports. This appendix covers the following topics:

- General Jupyter Notebook tips and tricks:
 - Useful keyboard shortcuts to speed up your workflow
 - A short introduction to the Markdown syntax to edit text cells
 - A few other useful tips
- Jupyter Notebook extensions
- **Matplotlib** styles for pretty plotting from the start
- Useful resources such as data repositories, Python packages, and similar

The various tips and tricks are not crucial for data analysis in Python, but it is very useful to make the workflow better and easier to pick up right where you left off in a project. Let's jump right in and start off by looking closer at some good things about Jupyter Notebook.

Jupyter Notebook

Jupyter Notebook is an interactive web application that sends/receives data from a programming language kernel. In this book, we have worked in Python; it is also possible to work in several other programming languages in Jupyter Notebook. The notebook format has support for what it calls checkpoints—when you save, it will create a checkpoint and you can always roll back to that previous checkpoint from **File | Revert to Checkpoint** in the menu.

One of the most important problems that Jupyter Notebook solves is that it provides a full record of your data analysis session; this record along with the data files is all that anyone needs to reproduce your analysis. The record may contain, except the code, (structured) text, images, videos, equations, and even interactive widgets. The notebook can be compiled into other formats that are easier to share, such as PDF and HTML. In addition to these things, it is possible to extend the functionality of Jupyter Notebook with extensions. After looking at some of the more useful keyboard shortcuts, we will go through a few of these extensions.

Useful keyboard shortcuts

First, I would like to go through a few of the most useful keyboard shortcuts. The general approach to keyboard shortcuts in Jupyter Notebook is very simple. It has *two* main modes: *command* and *edit* mode. As you might have suspected, edit mode is when you edit text in a cell and command mode is when you run commands in your notebook. The available keyboard shortcuts are of course reflected in what mode you are in. However, in both modes, *Shift + Enter* will run the current cell and *Ctrl + S* will save the notebook (and create a checkpoint).

Command mode shortcuts

Once in command mode, either by pressing *Ctrl + M* or *Esc*, the following keyboard shortcuts are available:

- *B/A*: This creates a new cell, *B* below or *A* above the current cell.
- *X/C/V*: This cuts, copies, and pastes the cell, just like you are used to in other programs. Pasting the cell here will paste it below the current cell.
- *D, D*: This deletes a cell.
- *Z*: This undoes the deletion.
- *L*: This shows line numbers. This is especially useful when getting error messages with a reference to a line number in your code where it breaks.
- *M*: This converts the current cell to a Markdown cell.
- *Shift + M*: This merges the current cell with the cell below.
- *O*: This toggles to show/hide the output shown directly below the cell.
- *H*: This shows all the keyboard shortcuts.
- *Enter*: This enters edit mode of the selected cell.

Edit mode shortcuts

When you are in edit mode, by pressing *Enter* while selecting the cell you want to edit, you can do the following actions:

- *Tab*: Indent, or tab completion; that is, start typing a command, tab will list available commands/methods/objects/variables to complete with that are present in the name space.
- *Ctrl + Shift + -*: This splits a cell at the current line
- *Ctrl + A*: This selects all content in a cell
- *Ctrl + Z*: This is for undo
- *Ctrl + Shift + Z*: This is for redo
- *Esc*: This enters command mode

As mentioned, these are some of the keyboard shortcuts available. These are the ones that are the most useful in my opinion. If you want to look at *all* of them, enter command mode and press *H*.

Markdown cells

In a Markdown cell that is created by selecting an existing cell and pressing *M*, you can perform the following functions:

- Create headings by preceding the text by a hash and space, "# ".
- Type normal text, just like in any text editor. You can style the text as follows:
 - **Italics** by surrounding the text with stars, that is, *text*
 - **Bold** by surrounding the text with two stars, that is, **text**
- Make bullet lists by preceding each bullet item with a star, as follows:

```
* Item1
* Item 2
  * Sub-item1
```
- Include a URL by typing [your link text] (<http://your-url.com>).
- Include an image with ! [image text] ([url_or_path_to_image.png](#)).
- Make a numbered list by preceding each item in the list with a number.

If you convert a cell to Markdown text, but want to convert it back to a code cell, you simply press *Ctrl + M* or *Esc* to enter command mode and then *Y* to convert the selected cell.

Markdown syntax is very extensive and Jupyter Notebook follows much of the same syntax as that used at GitHub; thus, for more information on what can be done, see <https://help.github.com/articles/basic-writing-and-formatting-syntax/>. Some of the possibilities are also shown in the accompanying notebook of this appendix.

Notebook Python extensions

Jupyter's functionality can be extended with extensions. Some of the extensions rely only on Jupyter, while others rely on external libraries and software. A few of them are inspired by plugins or functions of the CodeMirror online JavaScript editor (<https://codemirror.net>). A collection of Python-specific extensions can be installed from the IPython-contrib repository on GitHub. The URL for the collection is <https://github.com/ipython-contrib/IPython-notebook-extensions>. In this appendix, we will cover some of these extensions.

Installing the extensions

To install the collection of extensions along with the extension manager from the Anaconda repository, follow these steps:

1. Start an Anaconda command prompt and run the following:

```
conda install -c https://conda.binstar.org/juhasch nbextensions
```

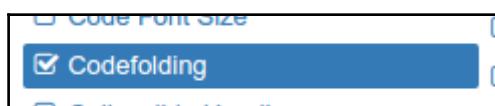
2. To activate some of the extensions that we want to use, start Jupyter Notebook.
3. Open a new browser tab and go to <http://localhost:8888/nbextensions> (where 8888 is the port that Jupyter listens to).
4. The page that you are presented with should look something like the following screenshot. The page is basically a list of the available extensions with checkboxes to activate them. If you click on the name of an extension, the page will load details about that extension:

The screenshot shows the 'Configurable extensions' section of the Jupyter configuration interface. A blue box highlights the checkbox for '(some) LaTeX environments for Jupyter'. Other extensions listed include AutoSaveTime, Autoscroll, Chrome Clipboard, Code Font Size, Codefolding, Collapsible Headings, Comment/Uncomment Hotkey, datestamper, Drag and Drop, Equation Auto Numbering, ExecuteTime, Exercise, Exercise2, Freeze, Gist-it, Help panel, Hide input, Hide input all, highlighter, Initialization cells, Keyboard shortcut editor, Launch QTConsole, Limit Output, Move selected cells, Navigation-Hotkeys, NbExtensions menu item, Notify, Printview, Python Markdown, Rubberband, Ruler, Runtools, Scratchpad, Search-Replace, SKILL Syntax, Skip-Traceback, spellchecker, Table of Contents (2), table_beautifier, Toggle all line numbers, Tree Filter, and zenmode.

- Now, by clicking on the checkbox next to the names, activate the following extensions that we will go through (alphabetical order):

- **Codefolding**
- **Collapsible Headings**
- **Help panel**
- **Initialization cells**
- **NbExtensions menu item**
- **Ruler**
- **Skip-Traceback**
- **Table of Contents (2)**

When you have done these things, each extension will have the checkbox next to it marked, as shown in the following screenshot:



To install the latest version from GitHub instead of that in the Anaconda repository (that is, the previous step 2), you can run the following:
`pip install https://github.com/ipython-contrib/IPython-notebook-extensions/archive/master.zip --user`

In my experience, the click response is a bit buggy, so make sure that they are all marked. After selecting all the specified extensions to be activated, you can also configure some of them. We will look at each of them separately, but the general layout revealed by clicking on the name of each extension is as follows:

- The name of the extension
- A short description
- Which versions of Jupyter Notebook it is compatible with
- An activation/deactivation button
- An image to the right, showing roughly what it does
- Possible parameters/settings for the extension

After this, the interface will grab and output the readme file, which is in Markdown syntax. In this file, the author of the extension puts any additional information that might be useful. In the coming sections, we will go through the extensions one by one.

Codefolding

The codefolding extension is a simple yet very useful extension. It will fold the indented lines of code, for example, functions or classes can be folded. Furthermore, it will also give you the option of folding at comments. The top of the information pane for this extension is shown here:

The screenshot shows the configuration pane for the Codefolding extension. At the top, it says "Codefolding". Below that, a message states: "This extension enables the CodeMirror feature to allow codefolding in code cells". It also mentions compatibility: "compatibility: 4.x". There are two buttons: "Activate" (gray) and "Deactivate" (blue). A "Parameters" section contains a "Hotkey to fold/unfold code" field with "Alt-F" and a "Change" button. On the right, there's a preview area titled "In [2]:" showing Python code:

```
class MyClass(object):
    """
    This is a test class
    """
    def afun(param1):
```

As an example of what you see in the readme file, I'll show you the top of the codefolding extension readme that Jupyter Notebook outputs here:

/nbextensions/usability/codefolding/readme.md?v=20160501205159

This extension adds codefolding functionality from CodeMirror to a codecell.

After clicking on the gutter (left margin of codecell) or typing `Alt+F`, the code gets folded. See the examples below. The folding status is saved in the cell metadata of the notebook, so reloading of a notebook will restore the folding view.

Supported modes

Three different folding modes are supported:

The readme is simply a more extensive description with figures and external links. With the codefolding extension, it is possible to hide long code snippets and functions within a cell. This is shown in the following example. The first image shows an arbitrary function in the way it looks in Jupyter Notebook:

```
def function_one(p1,p2):
    sel = [p1, p2][np.random.randint(2)]
    return sel
```

Clicking on the small arrow in the left margin will collapse the code into one line. It will then look like this:

```
def function_one(p1,p2):
```

As you can see in the first image of this section showing the parameters for this extension, the keyboard shortcut `Alt + F` will toggle the folding. Folding will also work on nested functions and statements; for each indentation level, you can fold the code. You can collapse code cells with comments as the first line as well:

```
# it is possible to use comments
# to hide code as well
function_one(10,30)
```

Once again by clicking on the arrow, you will collapse the rest of the code in the whole cell below it:



This is a very useful extension when you tend to write long functions or code, perhaps a plot with many different components, or if you have help functions written in the notebook.

Collapsible headings

With the collapsible heading extension, it is possible to group whole sections of cells by creating Markdown cells and defining headings. Normally, this would only display the text as a heading. The extension makes the heading and all cells below it collapsible—it will collapse everything below it until a heading of equal or greater level is encountered. The available parameters in the settings page are shown here:

The screenshot shows the 'Collapsible Headings' settings in the Jupyter Notebook settings page. It includes a description, compatibility information, and several configuration options:

- Compatibility:** 4.x
- Activate** (button) and **Deactivate** (button)
- Parameters** section:
 - Add a toolbar button to collapse the closest header cell
 - Add command-mode keyboard shortcuts to collapse/uncollapse the selected heading cell
 - Command-mode shortcut to collapse the selected heading cell:** Left (button) and Change (button)
 - Command-mode shortcut to uncollapse (expand) the selected heading cell:** Right (button) and Change (button)
- heading h2** (represented by a minus sign icon)
- collapsed heading h3** (represented by a plus sign icon)

You can set the keyboard shortcuts to (un)collapse a selected heading, add a toolbar button, and toggle the use of keyboard shortcuts. An example of what the results of using the extension are shown here:

▼ **1 A report**

This document gives short examples on how to work with the Jupyter Notebook and some of the extensions that I hope that you have now activated. It also shows that you can actually write nice summaries of the analysis, perhaps not for the final version of a report, but for early drafts it should be ideal.

▼ **1.1 Some example text and code**

In a Markdown cell you can for example

- Create headings
- Type normal text, just like any text editor and

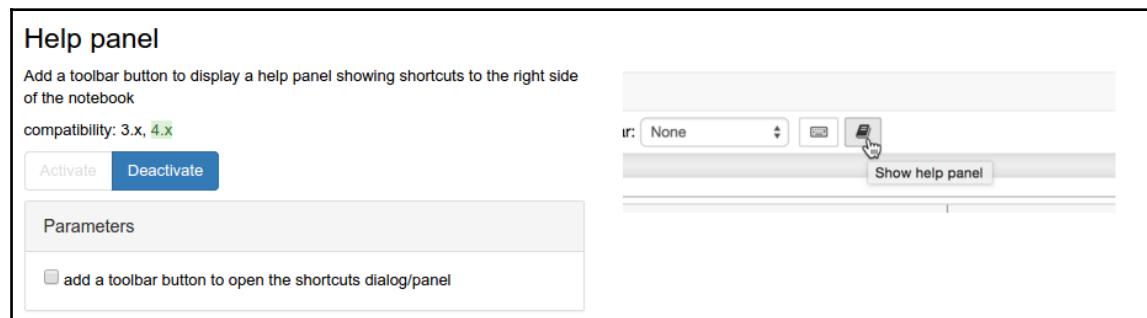
Clicking on the little arrow to the left of the heading will collapse the heading and everything below it under the same section. It will then look like the following image:

▶ **1 A report** [...]

This is very helpful when you are doing multiple analyses of similar or the same data. Try opening up one of the chapters that we worked on in the book with the extension active, and you will see the usefulness of this.

Help panel

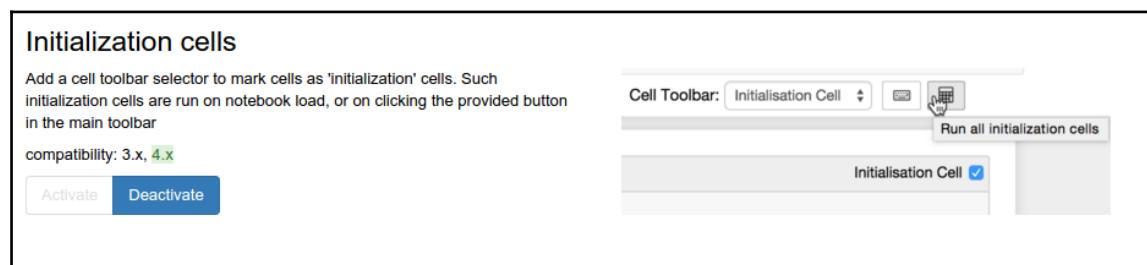
The help panel is useful when you start out writing your own code in Jupyter Notebook, as it has the possibility of displaying all the keyboard shortcuts in a panel alongside your notebook. The top of the details page for the extension looks as follows:



Here, you can check the box for **add a toolbar button to open the shortcuts dialog/panel**. Then you will have a button, as is shown to the right in the preceding image.

Initialization cells

Much of the code in the beginning of an analysis session is something that you want to run every time it is opened. The initialization cells extension alleviates this by adding two things—a cell toolbar that allows you to mark initialization cells and a button to rerun all these marked initialization cells. The following image shows the details page of the extension, and to the right is the button to trigger the rerunning of the initialization cells:



To use this extension, perform the following steps:

1. When activated, go and open a notebook and create the cells you want to have for starters. The accompanying example notebook has some initialization cells in it.
2. To change cells into initialization cells, you navigate to **View | Cell Toolbar | Initialization Cell**. When you have clicked this, each cell will get a toolbar (that is, cell toolbar) with a checkbox in the upper right corner, as shown in the following image:

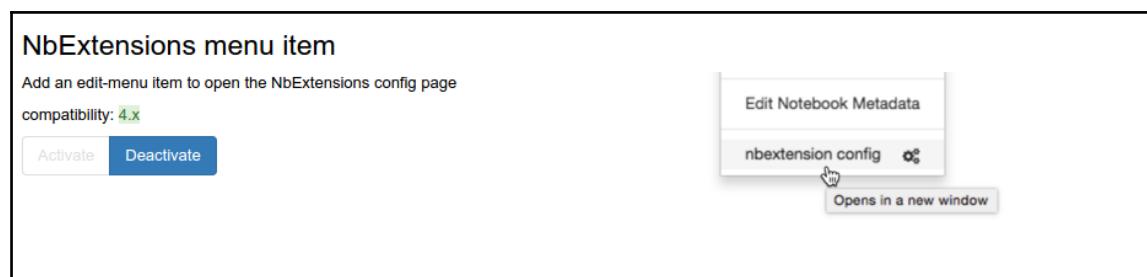


3. Click on the checkbox for the cells that you want to run automatically when you open the notebook, for example, cells with imports, data reading, and data cleaning.
4. Now, close the notebook, open it again, and watch the checked cells run automatically. You can also trigger this by clicking on the button that looks like a calculator; see the first image of this section.

This extension is very useful because sometimes we have to restart our kernel or notebook and when this happens, it is not that much fun to have to rerun all the cells that simply import modules and load data.

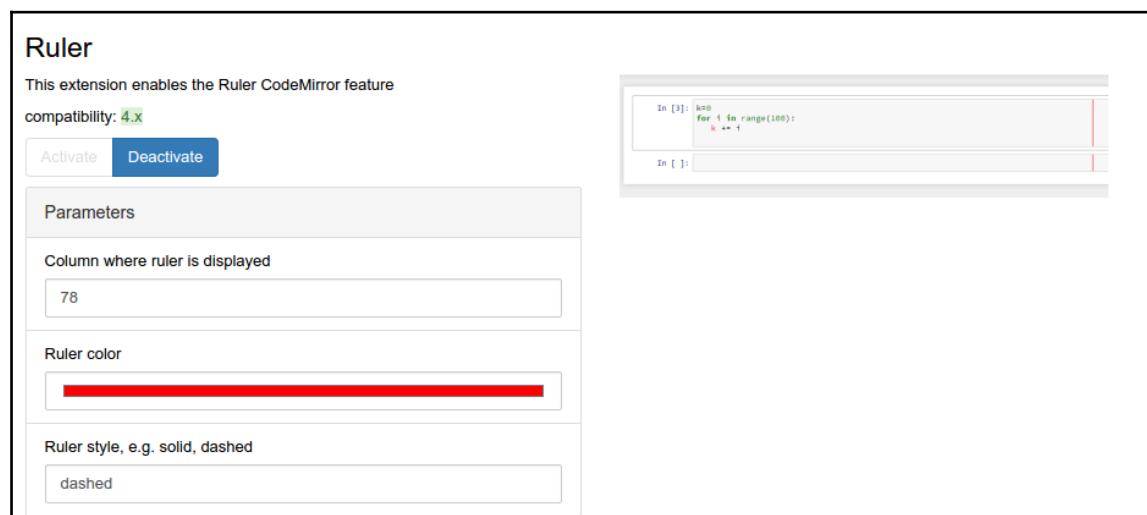
NbExtensions menu item

The NbExtensions menu item extension is very simple; it adds a menu item to open the extensions settings page where you can activate/deactivate extensions. The menu item can be found under the **Edit** item. The following is a screenshot from the extension details page showing the menu item to the left:



Ruler

The ruler is a simple extension and is for aesthetics so that you know when to wrap your code for it to follow standards. The available parameters are the column width and the color of the ruler and its line style, as shown in the following image:



The extension will draw a vertical line in each cell at the column width given in the parameters. The following image shows what it looks like:

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Skip-traceback

Sometimes there is an exception raised in the code that you run in a cell. When the stack trace to the exception is long, Jupyter Notebook will still display the whole trace. It can be a bit tedious to scroll to the bottom of the cell output to get to what caused the exception. There are no parameters to set for this extension. To give you a good example of this, I found a filed bug in the current version of NumPy giving a long trace. You can read about the bug at <https://github.com/numpy/numpy/issues/7547>. To test the skip-traceback extension, follow these instructions:

1. After the standard imports (with the extension activated as we described before), run the following:

```
values = (1+np.array([0, 1.e-15]))*1.e27
plt.plot(values)
```

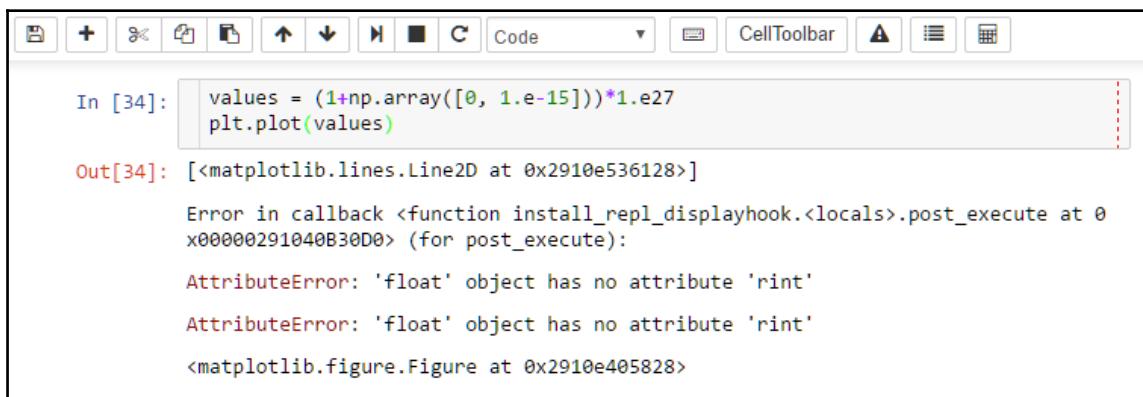
2. You should now see something like the following screenshot:

The screenshot shows a Jupyter Notebook interface. In the top toolbar, there are various icons for file operations, cell selection, and cell execution. Below the toolbar, the code cell content is shown. The code in the cell is:In [33]: values = (1+np.array([0, 1.e-15]))*1.e27
plt.plot(values)When the cell is run, it displays a long traceback in the output area. The traceback starts with a call from `ticker.py` and ends with an `AttributeError`. The error message is:

```
C:\Users\...\Anaconda3\lib\site-packages\matplotlib\ticker.py in _set_form  
at(self, vmin, vmax)  
    608         thresh = 1e-3 * 10 ** loc_range_oom  
    609         while sigfigs >= 0:  
--> 610             if np.abs(locs - np.round(locs, decimals=sigfigs)).max()  
< thresh:  
    611                 sigfigs -= 1  
    612             else:  
  
C:\Users\...\Anaconda3\lib\site-packages\numpy\core\fromnumeric.py in roun  
d_(a, decimals, out)  
    2791     except AttributeError:  
    2792         return _wrapit(a, 'round', decimals, out)  
-> 2793     return round(decimals, out)  
    2794  
    2795  
  
AttributeError: 'float' object has no attribute 'rint'
```

3. The trace is really long; you have to scroll through a long list of pointers and files. Now, click on the button on the toolbar that shows a triangle and an exclamation mark (see the preceding and following images); it toggles the hiding of the traceback.

4. Run the code again and you get the following:



The screenshot shows a Jupyter Notebook cell interface. The toolbar at the top includes icons for file operations, cell selection, and cell execution. Below the toolbar, the input cell (In [34]) contains the Python code: `values = (1+np.array([0, 1.e-15]))*1.e27
plt.plot(values)`. The output cell (Out[34]) displays the results of the plot and an associated error message:
`[<matplotlib.lines.Line2D at 0x2910e536128>]
Error in callback <function install_repl_displayhook.<locals>.post_execute at 0
x00000291040B30D0> (for post_execute):
AttributeError: 'float' object has no attribute 'rint'
AttributeError: 'float' object has no attribute 'rint'
<matplotlib.figure.Figure at 0x2910e405828>`

This is much better and less confusing and shows why skipping traceback is very useful sometimes. There are of course situations when viewing the full trace is useful, for example, when you want to report a bug.

Table of contents

The collapsible headings extension is good when working with long notebooks with multiple sections. The table of contents is useful when navigating around in such notebooks. The plugin only has a few parameters. You can let it number sections, choose to what depth the table of contents go to, and toggle if it should show a floating window or a table at the top of the notebook. Some of these can be set in the floating window as well:

Table of Contents (2)

The ToC2 extension displays a floating (draggable) table of contents of the notebooks headers. Optionally, it also allows to automatically number all notebook's sections, and to add a table of Contents cell at the top of the notebook.

compatibility: [4.x](#)

The screenshot shows the 'Table of Contents (2)' configuration in the Jupyter Notebook settings. It includes a 'Parameters' section with the following options:

- Automatically number notebook's sections
- Maximum level of nested sections to display on the tables of contents (set to 4)
- Add a Table of Contents at the top of the notebook
- Display toc window at startup

At the top left, there are 'Activate' and 'Deactivate' buttons. At the top right, there is a preview window showing a floating 'Contents' sidebar with a list of sections.

In the notebook, you can toggle the floating window with the table of contents by pressing the button. This is shown in the following image:



Once you have pressed the button, the floating window will appear to the right. For the example notebook of this appendix, it will look like the following:



Here, you have four buttons next to **Contents**, except for the clickable headings of the table. Clicking on the headings will take you to that part of the notebook. The first button, [-], will simply collapse the table of contents, and the button next to it will reload it; **n** will toggle the section numbering in the notebook; lastly, the **t** will toggle a table of contents at the top of the notebook in a separate cell. The output of clicking on the last button is shown here:

A screenshot of a Jupyter Notebook interface. The main content area contains a table of contents cell with the following content:

```
▼ Table of Contents
```

- [1 A report](#)
 - [1.1 Some example text and code](#)
 - [1.1.1 A longer function](#)
 - [1.2 Links](#)
 - [1.3 Images](#)
 - [1.4 Summary](#)
- [2 Matplotlib styles](#)
 - [2.1 Example styles](#)
- [3 Find a bug](#)

Below this cell is another cell containing the following code:

```
In [1]: import warnings
```

Other Jupyter Notebook tips

Here, I will give you some extra tips on using Jupyter Notebook. There are many things you can use it for and that is what makes it so good.

External connections

Starting Jupyter Notebook with the extra flag `-ip *`, or an actual IP instead of `*`, will allow external connections, that is, on the same network as your computer (or the Internet if you are connected directly). It will allow others to edit the notebook and actually run code on your computer, so be very careful with this. The full call would look as follows:

```
jupyter notebook -ip *
```

It can be useful in educational settings where you want people to be able to focus on coding and not installing things or if they do not have the right version of a certain package.

Export

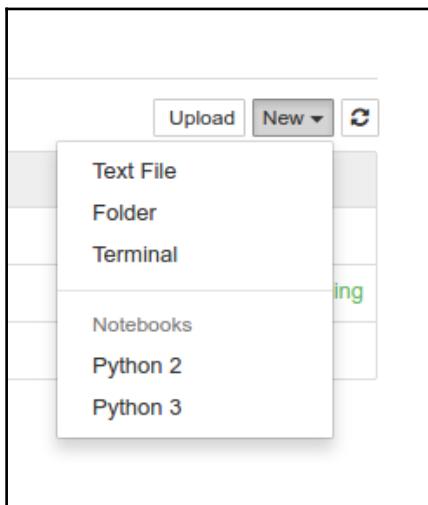
All the notebooks can be exported to PDF, HTML, and other formats. To reach this, navigate to **File | Download as** in the menu. If you export in PDF, then you might want to put the following in a cell at the beginning of your notebook. It will try to make PDF versions of your figures first, which will be vector-based graphics and thus lossless when you resize them and eventually be of better quality when incorporated into the PDF:

```
ip = get_ipython()  
ibe = ip.configurables[-1]  
ibe.figure_formats = { 'pdf', 'png'}  
print(ibe.figure_formats)
```

To export to PDF, you need other external software—a Latex distribution (<https://www.latex-project.org>) and Pandoc (<http://pandoc.org>). Once installed, you should be able to export your notebook to PDF; any Latex compilation errors should show up in the terminal that you started Jupyter Notebook from.

Additional file types

It is also possible to edit any other text file with Jupyter. In the Jupyter dashboard, that is, the main page that is opened when you start it, you can create new files that are not notebooks:



To give you an idea, I have included additional files in the appendix data files—one text file in Markdown format (ending with .md) and a file called `helpfunctions.py` with the `despine()` function that we created in previous chapters. In addition to these two, you also have the `mystyle.mplstyle` file to edit. In the editor, you can choose what format the file is in, and you will get highlighting for it.

Matplotlib styles

Throughout the book, we have worked with our custom style file, `mystyle.mplstyle`. As covered before, in matplotlib, there are numerous style files already included. To print out the styles available in your distribution, simply open a Jupyter Notebook and run the following:

```
import matplotlib.pyplot as plt  
print(plt.style.available())
```

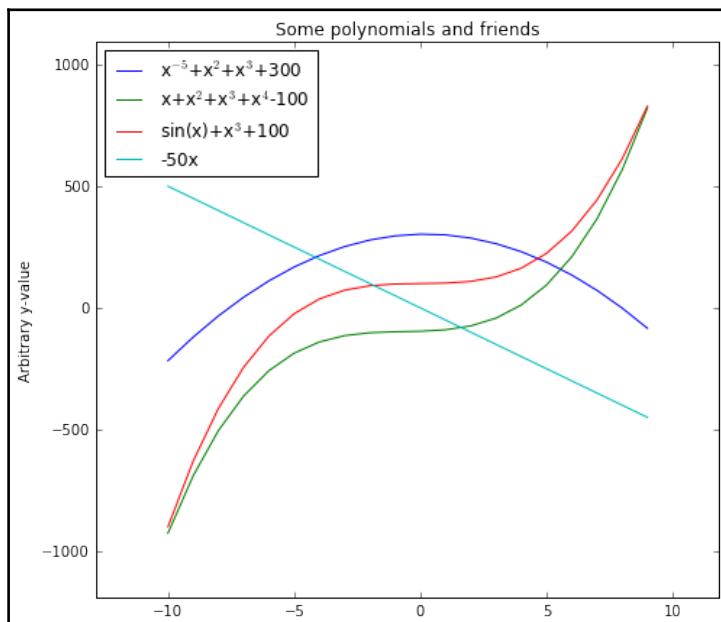
I am running matplotlib 1.5, and so I will get the following output:

```
['seaborn-deep', 'grayscale', 'dark_background', 'seaborn-whitegrid',  
'seaborn-talk', 'seaborn-dark-palette', 'seaborn-colorblind', 'seaborn-notebook',  
'seaborn-dark', 'seaborn-paper', 'seaborn-muted', 'seaborn-white',  
'seaborn-ticks', 'bmh', 'fivethirtyeight', 'seaborn-pastel',  
'ggplot', 'seaborn-poster', 'seaborn-bright', 'seaborn-darkgrid',  
'classic']
```

To get an idea of how a few of these styles look like, let's create a test plot function:

```
def test_plot():
    x = np.arange(-10,10,1)
    p3 = np.poly1d([-5,2,3])
    p4 = np.poly1d([1,2,3,4])
    plt.figure(figsize=(7,6))
    plt.plot(x,p3(x)+300, label='x$^{-5}$+x$^2$+x$^3$+300')
    plt.plot(x,p4(x)-100, label='x+x$^2$+x$^3$+x$^4$-100')
    plt.plot(x,np.sin(x)+x**3+100, label='sin(x)+x$^3$+100')
    plt.plot(x,-50*x, label='-50x')
    plt.legend(loc=2)
    plt.ylabel('Arbitrary y-value')
    plt.title('Some polynomials and friends',
              fontsize='large')
    plt.margins(x=0.15, y=0.15)
    plt.tight_layout()
    return plt.gca()
```

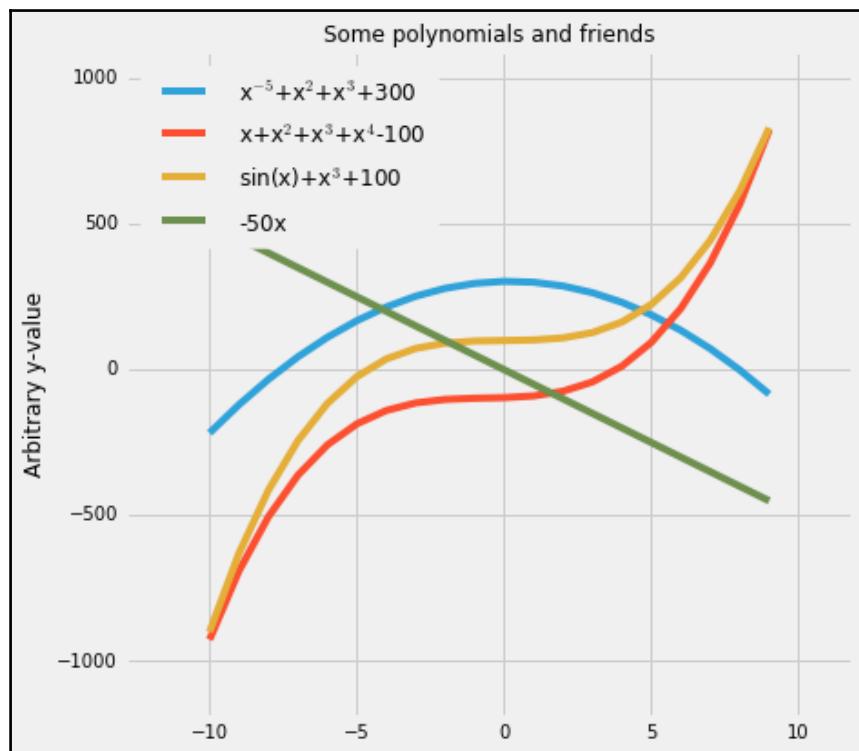
It will plot a few different polynomials and a trigonometric function. With this, we can create plots with different styles applied and compare them directly. If you do not do anything special and just call it, that is, `test_plot()`, you will get something that looks like the following image:



This is the default style in matplotlib 1.5; now we want to test some of the different styles from the preceding list. As the Jupyter Notebook *inline* graphics display uses the style parameters differently (that is, `rcParams`), we cannot reset the parameters that each style sets as we could if we were running a normal Python prompt. Thus, we cannot plot different styles in a row without keeping some parameters from the old style if they are not set in the new. What we can do is the following, where we call the plot function with the 'fivethirtyeight' style set:

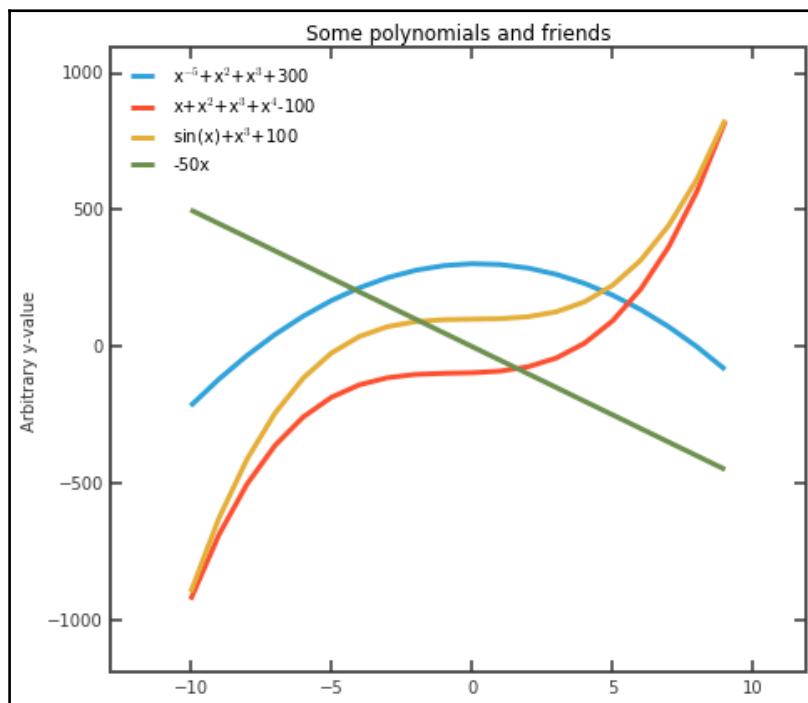
```
with plt.style.context('fivethirtyeight'):  
    test_plot()
```

By putting in the `with` statement, we confine whatever we set in that statement, thus, not changing any of the overall parameters:



This is what the 'fivethirtyeight' style looks like, a gray background with thick colored lines. It is inspired by the statistics site, <http://fivethirtyeight.com>. To spare you a bunch of figures showcasing several different styles, I suggest you run some on your own. One interesting thing is the 'dark-background' style, which can be used if you, for example, usually run presentations with a dark background. I will quickly show you what the `with` statement lets us do as well. Take our `mystyle.mplstyle` file and plot it as follows:

```
import os
stylepath = os.path.join(os.getcwd(), 'mystyle.mplstyle')
with plt.style.context(stylepath):
    test_plot()
```

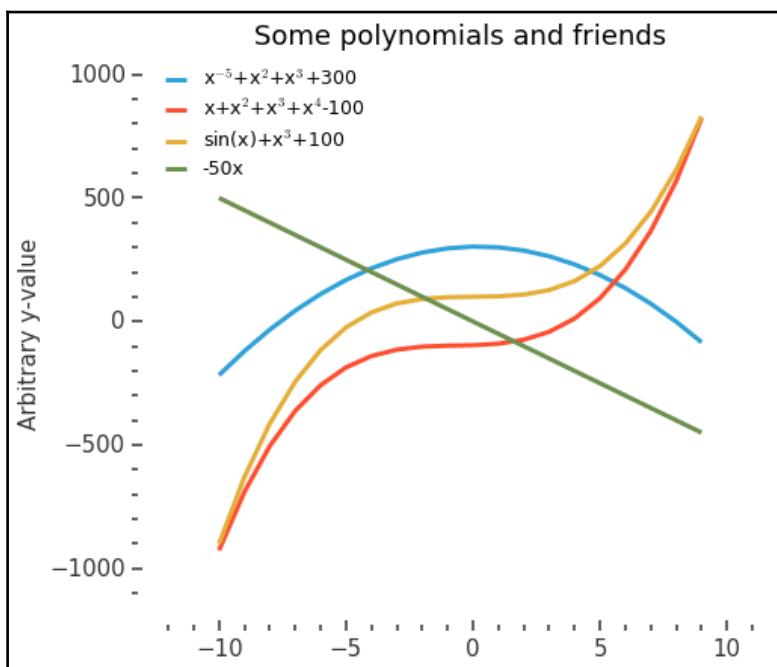


You might not always be completely satisfied with what the figure looks like—the fonts are too small and the big frame around the plot is unnecessary. To make some changes, we can still just call functions to fix things as usual within the `with` statement:

```
from helpfunctions import despine
plt.rcParams['font.size'] = 15
with plt.style.context(stylepath):
```

```
plt.rcParams['legend.fontsize'] = 'Small'  
ax = test_plot()  
despine(ax)  
ax.spines['right'].set_visible(False)  
ax.spines['top'].set_visible(False)  
ax.spines['left'].set_color('w')  
ax.spines['bottom'].set_color('w')  
plt.minorticks_on()
```

The output will be something as follows:



This looks much better and clearer. Could you incorporate some of these extra changes into the `mystyle.mplstyle` file directly? Try to do this—much of it is possible—and in the end, you have a nice style file that you can use.

One last important remark about style files. It is possible to chain several in a row. This means that you can create one style that changes the size of things (axis, lines, and so on) and another, the colors. In this way, you can adapt the one changing sizes if you are using the figure in a presentation or written report.

Useful resources

There are a vast number of resources on the topic of data analysis online, focused especially on Python. I have tried to compile a few here and hope that it will be of use to you. You will find a few sections under which I have listed resources, a short description, and a link where you can find more information.

General resources

General links to Python-related resources:

Continuum Analytics

<https://www.continuum.io>

Makers of Anaconda Python distribution. On their web page, you can find documentation and support.

Python and IPython

<https://python.org> and <http://ipython.org>

There's really no need for an explanation. We thank much in the world for these two projects.

Jupyter Notebook

<https://jupyter.org>

The Jupyter Notebook project web page where you can find more information, documentation, and help.

Python weekly newsletter

<http://www.pythonweekly.com>

A weekly (e-mail) newsletter to make it easier to keep up to date on what is going on in the world of Python.

Stack Overflow

<http://stackoverflow.com>

A question and answer page for basically everything. If you search online for any kind of Python programming problem, chances are high that you will land on one of their web pages. Register and ask or answer a question!

Enthought

<https://www.enthought.com>

Makers of Enthought Canopy that is, just like an Anaconda distribution, a full Python distribution. Enthought also has lots of courses and training for anyone interested.

PyPI

<https://pypi.python.org/pypi>

A repository of most Python packages and the first place that pip looks for packages.

Scipy-toolkits

<https://www.scipy.org/scikits.html>

The portal for the Scipy Toolkits (Scikits), affiliated packages for SciPy. The scikit-learn is a Scikit package.

GitHub

<https://github.com>

A repository for code that uses the famous Git versioning system to keep track of changes to the code. You can register and upload your own code for free as long as you make the code public. The code can be in Python or any other programming language.

Packages

This is a list of useful Python packages. Most of them can be installed via the conda or pip packaging systems.

PyMC

<https://pymc-devs.github.io/pymc/>

Alternatively, <https://github.com/pymc-devs/pymc>

A package for Bayesian inference/modeling analysis in Python; used in Chapter 6, *Bayesian Methods*, in this book.

emcee

<http://daniel.fm/emcee/>

An alternative to PyMC, an MCMC package for Bayesian inference.

scikit-learn

<http://scikit-learn.org>

A tool for machine learning data analysis with Python; used in Chapter 7, *Supervised and Unsupervised Learning*, of this book.

AstroML

<http://www.astroml.org/>

A package for machine learning, focusing on astronomical applications.

OpenAI Gym

<https://gym.openai.com/>

An open and publicly released toolkit to develop and test reinforcement learning algorithms.

Quandl

<https://www.quandl.com/>

A hub to access financial and economic data—they have a Python API that you can install and access large amounts of data with.

Seaborn

<https://stanford.edu/~mwaskom/software/seaborn/>

A package for statistical data visualization with Python. It has a few unique plotting functions that have not yet made it into the matplotlib package.

Data repositories

Here, I list some of the data repositories that are available online.

UCI Machine Learning Repository

<http://archive.ics.uci.edu/ml>

The University of California Irvine, Center for Machine Learning and Intelligent Systems repository of datasets, which is targeted at machine learning problems.

WHO – Global Health Observatory data repository

<http://apps.who.int/gho/data/node.home>

A large database of key health-related data from the whole world.

Eurostat

<http://ec.europa.eu/eurostat>

A database for various key statistics on all the countries in the European Union.

NTSB

<http://www.ntsb.gov>

The National Transportation Safety Board web page, which is a statistics database on automotive, rail, aviation, and marine accidents in USA.

OpenData by Socrata

<https://opendata.socrata.com>

A big database of various datasets (for example, airline accidents statistics for the whole world) that are easy to explore and find data.

General Social Survey (USA)

<http://gss.norc.org>

Yearly surveys in USA, with open and downloadable datasets and an online data exploration tool.

CDC

<http://www.cdc.gov/datastatistics/>

Centers for Disease Control and Prevention (CDC) have a lot of public data available on various diseases and health-related statistics.

Open Data Inception (+2500 sources)

<http://opendatainception.io>

A map showing the location and links to open data resources.

Data.gov.in

<https://data.gov.in>

The Indian government public data portal. It contains a rich and broad set of publicly available data to practice your data analysis skills.

Census.gov

<http://www.census.gov>

The United States census bureau has conducted surveys and collected data on various topics in USA.

Data.europa

<https://data.europa.eu/euodp>

The European Union Open Data Portal provides a single point of access to data from all the EU countries.

Visualization of data

The following is a list of some resources that are useful for visualization (overlapping here is Seaborn, which has been listed previously).

Fivethirtyeight

<http://fivethirtyeight.com/>

A great inspiration when it comes to the visualization of data. The site presents statistical analysis and presentation of data from around the world.

Plotly

<https://plot.ly>

Data analysis and visualization done online. Their tool for Python is now open source and free to use when self-hosted.

mpld3

<http://mpld3.github.io/>

Create interactive Python plots and export to the browser for others to explore.

Summary

In this appendix, we covered several things that are useful when doing data analysis and working in Jupyter Notebook. Hopefully, you will find great use of these resources and knowledge. There is so much data out there from so many different parts of the society just waiting to be analyzed. Given the increase in the amount of data that is produced and stored, we need more people who can analyze and present the data in an understandable way.

Bibliography

This course is a blend of text and quizzes, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Getting Started with Python Data Analysis, Phuong Vo.T.H, Martin Czygan*
- *Python Data Analysis Cookbook, Ivan Idris*
- *Mastering Python Data Analysis, Magnus Vilhelm Persson, Luiz Felipe Martins*

Index

A

AdaBoost

about 452

reference link 454

AdaBoostRegressor class

reference link 454

advanced Pandas use cases

for data analysis 54

hierarchical indexing 54, 56

panel data 56, 58

agglomerative clustering 526

AgglomerativeClustering class

reference link 527

aggregated counts

fitting, to Poisson distribution 238-241

aggregated data

fitting, to gamma distribution 237, 238

Alembic

URL 314

alias command 586

American Standard Code for Information Interchange (ASCII) 308, 567

Anaconda 567

about 168

setting up 168-170

URL 169

Analysis of variance (ANOVA)

about 567

one-way ANOVA 265

two-way ANOVA 265

used, for evaluating relations between variables 265, 266, 267

analytic signal

reference link 342

angle() function

reference link 342

annotate method 77

annotations 75-77

Anscombe's quartet 567

graphing 202-205

reference link 205

Apache Spark 326

approximation 363

arbitrary precision

used, for linear algebra 297, 298

using, for optimization 294, 295

area under the curve

(AUC, ROC AUC or AUROC)

about 472

examining 472-474

argrelmax() function

reference link 340, 520

array creation 16

array functions 21, 22

artificial intelligence (AI) 4

association tables

implementing 310-313

assortativity coefficient, graph

calculating 424, 425

reference link 425

asyncio module

reference link 546

used, for accessing resources

asynchronously 543-546

autoregressive models

reference link 389

used, for determining market efficiency

387, 389

average clustering coefficient
about 423
estimating 423, 424
average_clustering() function
reference link 424

B

bagging
reference link 451
used, for improving results 449, 450
bagging (bootstrap aggregating) 442
BaggingClassifier
reference link 451
bag-of-words model 567
reference link 410
bar plot 71
Bartlett's method 336
basic terms database
implementing 414-418
Bayesian analysis 241
Beaufort scale
reference link 284
Benevolent Dictator for Life (BDFL) 305
Berkeley Vision and Learning Center (BVLC) 8
beta
about 380, 567
reference link 381
betweenness centrality
determining 421, 422
reference link 422
betweenness_centrality() function
reference link 422
bias
determining 241-244
binary variable
correlating, with point biserial correlation 263-265
binomial distribution 241
Bokeh
about 81
differences, with matplotlib 81
plots, creating with 81
Bokeh installation
reference link 217

Bokeh plots
reference link 219
boosting
about 452
reference link 454
used, for better learning 452, 453
Boot2Docker 174
bootstrap aggregating 449
Box-Cox transformation
reference link 280
used, for normalization 278-280
box plots and kernel density plots
combining, with violin plots 220, 221
bright stars
searching for 517, 519, 520
bucketing 284
build options
BUILD_opencv_java 502
BUILD_opencv_python2 502
BUILD_opencv_python3 502
BUILD_opencv_xfeatures2d 502
OPENCV_EXTRA_MODULES_PATH 502
PYTHON3_EXECUTABLE 502
PYTHON3_INCLUDE_DIR 502
PYTHON3_LIBRARY 502
PYTHON3_NUMPY_INCLUDE_DIRS 502
PYTHON3_PACKAGES_PATH 502

C

Cache algorithms
reference link 559
caching 567
Caffe
about 8
reference 8
Calmar ratio
about 372
reference link 374
stocks, ranking with 372, 373
Capital Asset Pricing Model (CAPM)
about 380
reference link 381
cascade classifier
reference link 515

categorized corpus
 creating 402, 403, 404
CategorizedPlaintextCorpusReader
 reference link 405
centrality 420
central tendency, of noisy data
 measuring 275, 276, 277
cepstrum
 about 351
 reference link 354
chebyfit() function
 reference link 296
clip() function
 reference link 272
clique
 about 425, 568
 reference link 426
clique number, graph
 obtaining 425, 426
closeness centrality
 about 420
 reference link 421
clustering 568
clustering coefficient
 about 423
 reference link 424
code style
 standardizing 196, 197, 198, 199
Cohen's kappa
 about 470, 568
 examining 492, 493, 495
 reference link 495
ColorBrewer tool
 reference link 206
color quantization
 about 507
 reference link 509
colors
 quantizing 507, 509
command-line history
 about 587
 reference link 587
command-line tools 586
community package
 reference link 222
complete graph
 about 425, 568
 reference link 426
computational tools 49, 51
concurrent.futures module
 reference link 542
 used, for launching multiple tasks 540-542
conda 168, 567
confidence intervals
 of mean, determining 247-249
 of standard deviation, determining 247, 248
 of variance, determining 247, 248
 reference link 249
confusion matrix
 about 466, 568
 reference link 469
 used, for summarizing results of
 classification 466-468
confusion_matrix() function
 reference link 469
consensus set 446
contingency table
 about 265, 568
 reference link 267
continuous variable
 correlating, with point biserial
 correlation 263-265
continuumio/miniconda3 image
 URL 176
contour plot 72
co-occurrence matrix
 about 523
 reference link 526
coroutines
 about 543
 reference link 546
corpora
 download link 402
correlate() function
 reference link 342
correlation coefficient 495
cosine similarity
 about 427, 568
 reference link 430
cosine_similarity() function
 reference link 430

Count-min sketch
reference link 563
used, for streaming counting 561-563

cross-correlation 341-568

cross-validation (CV)
about 164
nesting 455-458
reference link 458

csvkit tool 129

custom magics
URL 179

D

d3.js
about 215
used for visualizing via mpld3 215-217

data
about 3
clustering hierarchically 460
clustering, with Spark 327-331
fitting, to exponential distribution 234-236
grouping 144, 145
indexing 48
rebinning 284, 285
selecting 48, 49
transforming, with logarithms 282, 283
transforming, with power ladder 280, 282
winsorizing 273, 274

data access
standardizing 196-199

data aggregation 141, 143

data analysis 202
about 4
algorithms 7
artificial intelligence 5
computer science 5
data cleaning 6
data collection 6
data processing 6
data product 7
data requirements 6
domain knowledge 5
exploratory data analysis 6
knowledge domain 5
libraries 7
machine learning 5

mathematics 5
modelling 7
process 4
Python libraries 9
statistics 5
steps 6, 7

database indices
reference link 317

database migration scripts
setting up 313, 314

DataFrame 36-39

data, in binary format
HDF5 114, 115
interacting with 113

data, in MongoDB
interacting with 115-119

data, in Redis
interacting with 120
list 121
ordered set 123
set 122, 123
simple value 120

data, in text format
interacting with 107
reading 107-111
writing 112

data munging
about 128, 129
data, cleaning 130, 132
data, merging 136-139
data, reshaping 139, 140
filtering 133, 135

data points
highlighting, with influence plots 228-230

data processing, using arrays
about 23
data, loading 25
data, saving 24

Data Science Toolbox (DST)
about 170, 568
installing 170-172

data structure, Pandas
about 34
DataFrame 36-39
Series 34, 35

data types 14, 16
date and time objects
 working with 86-93
DbVisualiser software
 reference link 585
decision tree learning 442
degree 424
degree_assortativity_coefficient() function
 reference link 425
degree distribution
 about 424
 reference link 425
density() function
 reference link 420
detail coefficients 363
determinants
 about 505
 reference link 507
DFFITS
 reference link 231
dilation
 about 512
 reference link 514
dimension tables
 star schema, implementing 319-323
discrete cosine transform (DCT)
 about 334, 568
 reference link 356
 used, for analyzing signals 354-356
discrete wavelet transform (DWT)
 applying 363-366
 reference link 366
distance
 reference link 231
distributed processing
 execnet, using 546-549
Docker
 URL, for user guide 176
docker-clean script
 reference link 588
Docker images
 Python applications, sandboxing 174, 175
docker tips 588, 589
document graph, with cosine similarity
 creating 427-430

dummy classifier
 comparing with 482-484
 strategies 482
DummyClassifier class
 reference link 484
dummy regressor
 comparing with 487-489
 strategies 487
DummyRegressor class
 reference link 489
Duncan dataset
 reference link 280

E

ECDF class
 reference link 252
efficient-market hypothesis (EMH)
 about 382, 568
 reference link 384
eigenvalues 568
eigenvectors 568
ensemble learning 432, 438
equal (eq) function 43
equal weights 2 asset portfolio
 optimizing 396-399
essential functionalities
 about 40
 binary operations 42, 43
 functional statistics 43-45
 function application 45
 head and tail 41
 labels, altering 40
 labels, reindexing 40
 sorting 46, 47
execnet
 reference link 549
 used, for distributed processing 546-549
EXIF
 reference link 523
ExifRead
 reference link 523
exponential distribution
 data, fitting into 234-236
 reference link 236
exponential smoothing
 about 343-368

reference link 345
smoothing factor 343
extra trees (extremely randomized trees) 458
extreme values
 exploring 253-256

F

F1-score
 computing 469-471
f1_score() function
 reference link 472
face detection
 about 514, 568
 reference link 517
fact
 star schema, implementing 319-323
false positive rate (FPR) 472
fancy indexing 19
Fano factor
 about 336
 reference link 338
fastcache
 reference link 559
fast Fourier Transform (FFT) 334, 569
Fundamental Algorithmic and Statistical Tools Laboratory (FASTLab) 8
fastNIMeansDenoisingColored() function
 reference link 511
fat tailed distribution
 reference link 374
features
 about 148
 recursively eliminating 432-434
filtering 406, 569
findroot() function
 reference link 296
Fisher transformation 257
folds 569
Fourier analysis 569
fourier() function
 reference link 299
Fourier series 569
frequency spectrum 334
frequency spectrum, of audio
 analyzing 351-353

frequentist approach 241
functions
 plotting, with Pandas 78-80
functools.lru_cache
 reference link 559

G

gamma distribution
 about 234
 aggregated data, fitting 237, 238
 reference link 238
generalized extreme value distribution (GEV)
 about 253
 reference link 256
genetic algorithms 444 569
genetic programming 444
geographical maps
 displaying 224, 226
 reference link 224
ggplot2
 about 226
 reference link 227
ggplot2-like plots
 reference link 226
 using 226-228
Global Interpreter Lock (GIL) 533
GPU (graphical processor units)
 about 569
 harnessing, with Open Computing Language (OpenCL) 564-566
gradient descent
 about 462
 reference link 464
graph_clique_number() function
 reference link 426
graphs
 reference link 420
greater equal (ge) function 43
greater than (gt) function 43
Gross Domestic Product (GDP) 224, 282

H

Haar cascades
 used, for detecting faces 514, 516

Haar feature-based cascade classifiers system 514
Haar wavelet 363
Hadoop Distributed File System (HDFS) 569
 URL 326
 using 325, 326
Haralick texture features
 about 523
 reference link 523
hat-matrix 229
heat dissipation 532
heatmaps
 creating 217-219
Hertzsprung-Russell diagram
 about 270
 reference link 272
Hessian matrix
 about 505
 reference link 507
hierarchical clustering
 about 460
 applying, on images 526, 527
 reference link 461, 527
histogram plot 74
hive plot
 about 221, 569
 used, for visualizing network graphs 221, 223
hiveplot package
 reference link 222
HSL and HSV
 reference link 520
HTML entities
 dealing with 308, 310
HTTP requests
 caching 559, 560
 reference link 560
Hurst exponent
 about 363
 reference link 366
hyperparameter optimization
 about 432
 reference link 458

I

image
 denoising 509, 511
 hierarchical clustering, applying 526, 527
 metadata, extracting from 521, 523
 patches, extracting from 511-513
 segmenting, with spectral clustering 527-529
 texture features, extracting from 523-526
image processing 499
image segmentation
 about 511
 reference link 514
image texture
 reference link 526
indices
 adding, after table creation 316, 317
individual stocks
 correlating, with broader market 377-379
influence plots
 about 569
 used, for highlighting data points 228-231
instantaneous phase
 about 340
 reference link 342
integral image
 about 505
 reference link 507
interpolation 94
interquartile mean (IQM) 275
inverse document frequency 406
IPython
 about 573
 configuring 179, 181
 references 177
IPython notebook
 about 176, 585
 used, for tracking package history 176- 179
 used, for tracking package versions 176-179
IPython notebooks and open data
 reference links 581
IPython notebook widgets
 interacting with 209-212
 reference link 213

Iris-Setosa 151
Iris-Versicolour 151
Iris-Virginica 151

J

jackknife resampling
reference link 249
jackknifing 247 569
Java Runtime Environment (JRE) 325
JavaScript Object Notation (JSON) 569
joblib
installation link 227
used, for reusing models 458, 459
jq tool 129
Just in time compiling
Numba, using 533, 534

K

kernel density
estimating 244-246
estimating, reference link 246
kernel density plots and box plots, combining
with violin plots 220, 221
kernels 564
K-fold cross-validation 569
K-means clustering
reference link 331
kmeans() function
reference link 509
kurtosis
calculating 551-555

L

LDA
reference link 438
learning curves 432, 569
least recently used (LRU) cache
used, for caching 556-558
leaves 442
legends 75-77
lemmatization
about 406-409
reference link 410

less equal (le) function 43
less than (lt) function 43
leverage 229
libraries, for data processing
Mirador 9
Modular toolkit for data
processing (MDP) 9
Natural language processing toolkit
(NLTK) 9
Orange 9
RapidMiner 9
Statsmodels 9
Theano 9
libraries, implemented in C++
Caffe 8
MLpack 8
MultiBoost 8
Vowpal Wabbit 8
libraries, in data analysis
Mahout 8
Mallet 8
overview 7
Spark 8
Weka 7
linear algebra
about 26
arbitrary precision, using for 297, 298
with NumPy 26, 27
linear discriminant analysis (LDA)
about 569
applying, for dimension reduction 437, 438
linkage() function
reference link 461
liquidity
stocks, ranking with 370, 371
LiveStats
reference link 556
lmplot() function
reference link 205
logarithmic plots 570
logarithms
used, for transforming data 282, 283
logging
for robust error checking 182, 184
reference link 185

logistic function 570
logit() function
 applying, for transforming proportions 286, 287
lombscargle() function
 reference link 351
Lomb-Scargle periodogram
 about 349, 570
 reference link 351
 using 349, 350
lu_solve() function
 reference link 299
lxml documentation
 URL 310

M

machine learning (ML) 4
machine learning models
 defining 147, 148
 supervised learning 148
 unsupervised learning 148
Mahout
 about 8
 reference 8
main sequence 270
Mallet
 about 8
 reference 8
mathematics and statistics
 reference links 582
Matplotlib
 about 11, 574
 configuring 190-194
 references 194
Matplotlib API Primer
 about 62-64
 figures 67
 line properties 65, 66
 subplots 67-69
matplotlib color maps
 reference link 208, 209
 selecting 208, 209
matrix of scatterplots
 viewing 213-215
matthews_corrcoef() function
 reference link 497

Matthews correlation coefficient (MCC)
 about 470-570
 reference link 497
maximum clique 425
maximum drawdown 372
maximum likelihood estimation (MLE) method 239
MayaVi 81
mean
 calculating 551-555
mean absolute deviation (MAD) 236
mean_absolute_error() function
 reference link 492
mean absolute error (MeanAE)
 calculating 490, 491
 reference link 492
Mean Absolute Percentage Error (MAPE)
 determining 485, 486
 reference link 487
Mean Percentage Error (MPE)
 determining 485, 486
 reference link 487
mean silhouette coefficient
 used, for evaluating clusters 479-481
mean_squared_error() function
 reference link 479
mean squared error (MSE)
 about 343
 computing 476-478
 reference link 479
medfilt() documentation
 reference link 362
median_absolute_error() function
 reference link 479
median absolute error (MedAE)
 computing 476-478
mel frequency spectrum
 about 354
 reference link 354
mel scale
 about 351
 reference link 354, 356
Memory class
 reference link 459
memory leaks 550 570

memory_profiler module
reference link 551
used, for profiling memory usage 550, 551

memory usage
profiling, with memory_profiler module 550, 551

metadata
extracting, from images 521, 523

methods
for manipulating documents 119

Miniconda 168

Mirador
about 9
reference 9

missing data
working with 51-53

MLpack
about 8
reference 8

models
reusing, with joblib 458, 459

Modern Portfolio Theory (MPT)
about 397
reference link 399

Modular toolkit for data processing (MDP)
about 9
reference 9

Monte Carlo method
about 249
reference link 252

Moore's law 570

moving block bootstrapping
time series data
about 359-361
reference link 362

mpld3
d3.js, used for visualization via 215-217

mpmath 270

MultiBoost 8

multiple models
majority voting 439-441
stacking 438-441

multiple tasks
launching, with concurrent.futures module 540-542

multiple threads
running, with threading module 536-539

N

named entities
recognizing 410-412

named-entity recognition (NER)
about 410, 570
reference link 410, 412

Natural language processing toolkit (NLTK) 9

nested cross-validation 455

network graphs
visualizing, with hive plots 221-223

NetworkX
reference link 222

news articles
tokenizing, in sentences 405
tokenizing, in words 405, 406

n-grams 406

noisy data
central tendency, measuring 275-277

non-ASCII text
dealing with 308, 310

non-negative matrix factorization (NMF)
documentation link 414
reference link 414
used, for extraction of topics 412, 414

non-parametric runs test
used, for examining market 382-384

not equal (ne) function 43

Numba
used, for Just in time compiling 533-535

numerical expressions
speeding up, with Numexpr 535, 536

Numexpr
reference link 536
used, for speeding up numerical expressions 535, 536

NumPy
about 10, 13, 575, 576
linear algebra with 26
random numbers 27-30
URL 182

NumPy arrays
about 14

array creation 16
data type 14, 16
fancy indexing 19
indexing 18
numerical operations on arrays 20
slicing 18

NumPy print options
seeding 194, 195
URL 196

O

object detection 514
object-relational mapping (ORM) 302, 570
octave 503
Open Computing Language (OpenCL)
about 570
used, for harnessing GPU 564-566
Open Source Computer Vision (OpenCV)
about 570
reference link 500
setting up 500-503
Orange
about 9
reference 9
Ostu's thresholding method
about 512
reference link 514
outliers
about 270
clipping 270, 271
filtering 270, 271
reference link 270
overfitting 163

P

Pandas
about 10, 576
data structure 34
configuring 188, 189
package overview 33
parsing functions 111
URL 188

pandas library
about 188
URL 188

Pandas objects
parameters 110

PCA class
reference link 436

pdist() function
reference link 461

peaks
analyzing 338-340

Pearson's correlation
reference link 260
used, for correlating variables 257-260

PEP8
about 14
URL 14

pep8 analyzer
URL 197
using 196

periodogram() function
reference link 336

periodograms
used, for performing spectral analysis 334, 335, 336

presentations
reference links 582, 583

phase synchronization
measuring 340-342

phi coefficient 570

plot types
bar plot 71
contour plot 72
exploring 70
histogram plot 74
scatter plot 70

point biserial correlation
reference link 265
used, for correlating binary variable 263, 264
used, for correlating continuous variable 263, 264

Poisson distribution
about 571
aggregated counts, fitting 238, 240
reference link 241

posterior distribution 241

power ladder
used, for transforming data 280, 281

power spectral density
estimating, with Welch's method 336-338

precision
computing 469, 470, 471
reference link 472

precision_score() function
reference link 472

prediction performance
measuring 162-164

principal component analysis (PCA)
about 160, 570
applying, for dimension reduction 435
reference link 436

principal component regression (PCR)
about 435
reference link 436

principal components 435, 570

prior distribution 241

probability weights
used, for sampling 249-252

probplot() function
reference link 476

Proj.4
reference link 224

proportions
transforming, by applying logit()
function 286-288

PyMongo 11

PyOpenCL
reference link 566

PyOpenCL 2015.2.3
reference link 564

Python applications
sandboxing, with Docker images 174, 175

Python data visualization tools
about 80
Bokeh 81
Mayavi 81, 82

Python libraries, in data analysis
about 9
Matplotlib 11
NumPy 10
Pandas 10
PyMongo 11
scikit-learn library 11

Python threading
reference link 539

R

R library
homepage link 227

RandomForestClassifier class
reference link 445

random forests
about 442
reference link 445, 459
used, for learning 442, 443, 444, 445

random number generators
seeding 194, 195

random walk hypothesis (RWH) 385

random walks
reference link 386
testing for 385, 386

RANSAC algorithm
reference link 448
used, for fitting noisy data 445-447

RapidMiner
about 9
reference 9

recall
computing 469-471
reference link 472

recall_score() function
reference link 472

receiver operating characteristic (ROC)
examining 472, 473
reference link 474

regressor
visualizing 475, 476

reports
standardizing 196-199

reproducible data analysis 168

reproducible sessions 587

requests-cache website
reference link 560

rescaled range 363
reference link 366

residual sum of squares (RSS)
calculating 490, 491
reference link 492

Resilient Distributed Datasets (RDDs) 326
resources
accessing asynchronously, with
 asyncio module 543-545
returns statistics
analyzing 374-376
RFE class
reference link 434
RGB (red, green and blue) 508
risk and return
exploring 380
risk-free rate 380
robust error checking
with logs 182, 184
robust linear model
fitting 288-290
robust regression 288 571
roc_auc_score() function
reference link 474

S

savgol_filter() function
reference link 348
Savitzky-Golay filter
about 346
reference link 348
Scale-invariant Feature Transform (SIFT)
about 503
applying 503, 504
documentation, reference link 505
reference link 505
scatter plot 70, 571
scikit-learn 226
about 577
 references 194, 196
scikit-learn library 11
scikit-learn modules
 data representation, defining 150-152
 defining, for different models 148, 149
SciPy
about 578
for exponential distribution, reference link
 236
for Poisson distribution 241
seaborn 578
seaborn color palettes
about 205-207
reference link 208
selecting 205
search engine indexing
reference link 418
security market line (SML) 380
Selenium
URL 305
using 302
Series 34, 35
shapefile format 224
shared nothing architecture
about 547
reference link 549
shared-nothing architecture 571
Sharpe ratio
about 370
reference link 372
stocks, ranking with 370, 371
short-time Fourier
transform (STFT) 351, 571
signal processing 571
signals
analyzing, with discrete cosine
 transform (DCT) 354-356
silhouette coefficients
about 479
reference link 481
silhouette_score() function
reference link 481
simple and log returns
computing 368, 369
reference link 369
Single Instruction Multiple Data (SIMD) 13
skewness
calculating 551-555
smoothing
evaluating 346-348
social network closeness centrality
calculating 420, 421
social network density
computing 418-420
software aspects 532
software performance
improving 532

Sortino ratio
about 372, 373
reference link 374
stocks, ranking with 372, 373

Spark
about 8
data, clustering 327-331
references 8
setting up 326, 327

Spearman rank correlation
about 571
reference link 263
used, for correlating variables 260-262

spectral analysis
performing, with periodograms 334-336
reference link 336

spectral clustering
about 571
reference link 529
used, for segmenting images 527-529

spectral_clustering() function
reference link 529

Speeded Up Robust Features (SURF)
detecting 505-507
reference link 507

split() function
reference link 520

SQLAlchemy
reference link 319

square root of the MSE (RMSE) 477

stacking
about 439
reference link 441

Stanford Network Analysis Project (SNAP)
about 221
reference link 221

star schema
about 319, 571
implementing, with dimension tables 319-323
implementing, with fact 319-323
URL 324

statistics functions 43-45

Statsmodels
about 9, 579
references 246

stemming 406-409

STFT
reference link 354

stock prices database
populating 391-395
tables, creating for 389, 391

stop words
about 571
reference link 410

streaming algorithms 561

Structured Query Language (SQL) 571

supervised learning
about 152-157
classification 152-157
classification problems 148
regression 152-157
regression problems 148

Support Vector Machine (SVM) 154

SURF
documentation, reference link 507

T

tab separated values (TSV) 224

table column
adding, to existing table 314, 315

tables
creating, for stock prices database 389, 390

tabulate PyPi
URL 199

term frequency 406

term frequency-inverse document frequency (tf-idf) 571

test web server
setting up 317, 319

text method 77

texture features
extracting, from images 523-526

TF-IDF
about 406-409
reference link 410

TfidfVectorizer class
reference link 410

Theano
about 9, 462, 463
documentation link 464
installing 462

threading module
used, for running multiple threads 536-539

Timedeltas 100

time series
about 572
plotting 101, 103, 104
reference, Pandas documentation 88
resampling 94

time series data
block bootstrapping 357-359
block bootstrapping, reference link 359
downsampling 94, 96
unsampling 97, 98

time series primer 85

time slicing 536

time zone handling 99

tmean()
reference link 277

topic models
reference link 414

topic models;about 412

trend smoothing factor 343

trima()
reference link 277

trimean 275

truncated mean 275

two-way ANOVA
reference link 267

U

unigrams 406

unit testing
about 185
performing 185, 187

unittest.mock library
URL 187

unsupervised learning
clustering 158-162
defining 158-162
dimensionality reduction 158-162

V

Vagrant
about 170
reference link 172
URL 171

validation 455

validation curves 432

variables
correlating, with Pearson's correlation 257-259
correlating, with Spearman rank correlation 260, 262
relations evaluating, with ANOVA 265-267

variance
calculating 551-555
reference link 556

Viola-Jones object detection framework
about 514
reference link 517

violin plots
about 220, 572
box plots and kernel density plots, combining with 220, 221
reference link 221

VirtualBox
about 170, 172
URL 171

virtualenv
virtual environment, creating with 172-174

virtual environment
creating, with virtualenv 172-174
creating, with virtualenvwrapper 172-174
URL 174

virtualenvwrapper
URL 174
virtual environment, creating with 172-174

visualization toolkit (VTK) 81

VotingClassifier class
reference link 441

Vowpal Wabbit
about 8
reference 8

W

Wald-Wolfowitz runs test
about 382
reference link 384

watermark extension
using 177

weak learners 452

web

scraping 305-307

web browsing

simulating 302-305

weighted least squares

about 291

used, for taking variance into account 291, 292

Weka

about 7

reference 7

welch() function

reference link 338

Welch's method

reference link 338

used, for estimating power spectral density 336-338

winsorizing technique 273, 274, 572

Within Cluster Sum of Squares (WCSS) 328

Within Set Sum Squared Error (WSSSE) 328

WordNetLemmatizer class

reference link 410

X

xmlstarlet tool 129

XPath

URL 305

Y

YAML

about 170

URL 170

A

adjusted R-square value 88
age-adjustment 73
age-standardization 73
agglomerative clustering 122
Akaike Information Criterion (AIC) 88, 231
Anaconda Scientific Python
 URL 7
autocorrelation function (ACF) 234
automatic function 234
autoregressive (AR) model 230, 231
autoregressive integrated moving average (ARIMA)
 model 235, 236
Aviation Accident Database 141
Aviation Safety Network
 URL 141

B

Bayes formula 139
Bayes Information Criterion (BIC) 88
Bayesian analysis
 of data 151, 152, 153, 155, 156
Bayesian inference 138
Bayesian Information Criterion (BIC) 231
Bayesian method 138, 139
Binomial distribution 64, 67
boxplot 35, 36

C

cartopy 160
Centers for Disease Control and Prevention (CDC)
 265
classifier
 selecting 202
climate change 163
cluster finding 109, 110

clustering 108, 109, 183, 184, 187
components
 about 220
 decomposing 221, 223, 224, 225
confidence intervals
 versus credible 139
continuous data 51
Continuum Analytics 261
coordinates
 plotting 160
cumulative distribution function 42, 43, 44, 45,
 46, 48, 49

D

data repositories 264
data visualization 265
data
 Bayesian analysis 151, 152, 153, 155
 binning 148
Degrees From Equator (DFE) 116
dependent variable 88
Df Model 88
Df Residuals 88
differencing 227, 228
discrete data 51
distance from equator (DFE) 83
distributions
 working with 51, 53, 55, 56, 57, 60, 61
divisive clustering 122

E

Enthought
 URL 7, 262
Eurostat 264
expected value 62
experiment 42

exponential distribution 57

F

feature-rich emcee package
URL 140

G

General Social Survey (GSS)
about 20
data, downloading 20
data, obtaining 20
data, reading 21, 22
URL 264

GitHub
URL 262

gross domestic product (GDP)
about 91
versus absolute latitude 116, 118, 119, 120

H

Heliocentric distance 123

hierarchical cluster algorithm 132, 134, 135, 137

hierarchical clustering analysis
about 122
agglomerative clustering 122
data, reading in 122, 131
data, reducing 122, 127
divisive clustering 122

Hierarchical Data Format (HDF) 99

histogram 23, 26, 27

I

imports 10

indexing
and slicing 209, 211

intercept 72

International Organization for Standardization (ISO) 82

IPython library
URL 8

IPython notebook 7
about 9

J

John Snow
on cholera 110, 111, 112, 113, 114

Jupyter library
URL 8

Jupyter Notebook
about 9, 238
command mode shortcuts 239
edit mode shortcuts 240
keyboard shortcuts 239
markdown cells 240
URL 261

Jupyter
URL 9

K

K-means clustering 116

K-Nearest Neighbor 200

Kernel Density Estimation (KDE) 29

keyboard shortcuts 239

L

Law Dome
URL 164

Least Absolute Shrinkage and Selection Operator (LASSO) 176

libraries 8

linear analysis
Bayesian analysis and OLS, checking with 181, 182, 183

linear regression
about 71, 72, 73, 176
climate data 176, 178, 179, 181
dataset, getting 73, 74, 75, 76, 78, 80
testing with 81, 82, 83, 84, 86, 87, 89, 91

logistic regression 100, 103, 104

M

machine learning
supervised 174, 175
unsupervised 174, 175

Markov Chain Monte Carlo (MCMC) 140

matplotlib library
URL 8, 28

matplotlib styles 256, 257, 259
mean 62
models
 about 41
 creating 166, 167
 forms 41
 origin 63
 sampling 166, 167
month
 binning by 158
MovieTweetings 50K movie ratings dataset
 URL 11
moving average (MA) 232, 233
Mpl toolkits 162
multivariate distribution 68
multivariate regression
 about 91
 economic indicators, adding 91, 94, 97
 taking step back 98

N

National Center for Supercomputing Applications (NCSA) 99
National Health Statistics Reports
 URL 52
National Opinion Research Center (NORC)
 URL 20
National Transportation Safety Board (NTSB)
 about 141
 URL 141
Normal Distribution Plot 31
notebook interface 9
Notebook Python extensions
 about 241
 codefolding extension 243
 collapsible headings 245, 246
 help panel 247
 initialization cells 247
 installing 241
 NbExtensions menu item 249
 ruler 249
 skip-traceback 250
 table of contents 252
Notebooks
 URL 9

NTSB database
 about 141, 142, 143, 144, 146, 147
 URL 141
NumPy
 URL 8

O

OpenData by Socrata 264
 URL 141
Ordinary Least Squares (OLS) 176
Ordinary Least Squares (OLS) method 87, 96

P

p parameter 233
Pandas data type 22
Pandas library
 example 10, 11, 12, 13, 16, 17
 URL 8
Pandas
 and time series data 206, 207, 209
partial autocorrelation function (PACF) 234
patterns 220
Pearson correlation coefficient 79
percent point function 56
point estimates 34
predefined distributions
 URL 51
probability density function (pdf) 61, 62, 63
probability mass function (pmf) 63
pseudorandom 43
pymc library
 URL 8
PyPI
 URL 262
Python packages 140
Python weekly newsletter
 URL 261
Python
 and IPython 261
 URL 7

Q

q parameter 233

R

Random Forest 201
random module
 URL 43
random variables 42
random variates (vs) 44, 58
resampling 212
resources
 about 261
 general 261

S

scatterplots 37, 38, 39
scikit-learn library
 URL 8
Scikit-learn
 about 175
 URL 175
scipy 8
Scipy-toolkits
 URL 262
seeds classification
 about 188, 189
 data visualization 190, 192
 data, classifying 196, 197
 feature, selecting 194, 196
sigmoid function 102
smoothing 212, 213, 214, 215, 217
Stack Overflow
 URL 262
STATA 21
stationarity 218, 219
statistical interface 32
Statistics Handbook

URL 52
suicide rate
 versus GDP 116
Support Vector Machine (SVM) 188
surface sites
 URL 164
SVC linear kernal 198
SVC polynomial 200
SVC Radial Basis Function (RBF) 199

T

time series analysis 204
time series models 229

U

univariate distributions 68
univariate data
 about 23
 boxplot 35
 boxplots 33
 characterization 29, 30
 histograms 23, 24, 25, 26, 27, 28
 numeric summaries 33
 numerical data 33, 34
 statistical interface 32

V

variable
 relationships 37, 38

W

weighted least squares (WLS) 88
World Coordinate System (WCS) 123
World Health Organization 9WHO) 73



Thank you for buying **Python: End-to-end Data Analysis**

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Please check www.PacktPub.com for information on our titles