Undergraduate Topics in Computer Science

Laura Igual · Santi Seguí

Introduction to Data Science

A Python Approach to Concepts, Techniques and Applications

Springer

Undergraduate Topics in Computer Science

Undergraduate Topics in Computer Science (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

More information about this series at http://www.springer.com/series/7592

Laura Igual · Santi Seguí

Introduction to Data Science

A Python Approach to Concepts, Techniques and Applications

With contributions from Jordi Vitrià, Eloi Puertas
Petia Radeva, Oriol Pujol, Sergio Escalera, Francesc Dantí
and Lluís Garrido

Springer

Laura Igual
Departament de Matemàtiques i Informàtica
Universitat de Barcelona
Barcelona
Spain

Santi Seguí
Departament de Matemàtiques i Informàtica
Universitat de Barcelona
Barcelona
Spain

With contributions from Jordi Vitrià, Eloi Puertas, Petia Radeva, Oriol Pujol, Sergio Escalera, Francesc Dantí and Lluís Garrido

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-319-50016-4 ISBN 978-3-319-50017-1 (eBook)
DOI 10.1007/978-3-319-50017-1

Library of Congress Control Number: 2016962046

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Subject Area of the Book

In this era, where a huge amount of information from different fields is gathered and stored, its analysis and the extraction of value have become one of the most attractive tasks for companies and society in general. The design of solutions for the new questions emerged from data has required multidisciplinary teams. Computer scientists, statisticians, mathematicians, biologists, journalists and sociologists, as well as many others are now working together in order to provide knowledge from data. This new interdisciplinary field is called *data science.*

The pipeline of any data science goes through asking the right questions, gathering data, cleaning data, generating hypothesis, making inferences, visualizing data, assessing solutions, etc.

Organization and Feature of the Book

This book is an introduction to concepts, techniques, and applications in data science. This book focuses on the analysis of data, covering concepts from statistics to machine learning, techniques for graph analysis and parallel programming, and applications such as recommender systems or sentiment analysis.

All chapters introduce new concepts that are illustrated by practical cases using real data. Public databases such as Eurostat, different social networks, and MovieLens are used. Specific questions about the data are posed in each chapter. The solutions to these questions are implemented using Python programming language and presented in code boxes properly commented. This allows the reader to learn data science by solving problems which can generalize to other problems.

This book is not intended to cover the whole set of data science methods neither to provide a complete collection of references. Currently, data science is an increasing and emerging field, so readers are encouraged to look for specific methods and references using keywords in the net.

Target Audiences

This book is addressed to upper-tier undergraduate and beginning graduate students from technical disciplines. Moreover, this book is also addressed to professional audiences following continuous education short courses and to researchers from diverse areas following self-study courses.

Basic skills in computer science, mathematics, and statistics are required. Code programming in Python is of benefit. However, even if the reader is new to Python, this should not be a problem, since acquiring the Python basics is manageable in a short period of time.

Previous Uses of the Materials

Parts of the presented materials have been used in the postgraduate course of *Data Science and Big Data* from Universitat de Barcelona. All contributing authors are involved in this course.

Suggested Uses of the Book

This book can be used in any introductory data science course. The problem-based approach adopted to introduce new concepts can be useful for the beginners. The implemented code solutions for different problems are a good set of exercises for the students. Moreover, these codes can serve as a baseline when students face bigger projects.

Supplemental Resources

This book is accompanied by a set of IPython Notebooks containing all the codes necessary to solve the practical cases of the book. The Notebooks can be found on the following GitHub repository: https://github.com/DataScienceUB/introduction-datascience-python-book.

Acknowledgements

We acknowledge all the contributing authors: J. Vitrià, E. Puertas, P. Radeva, O. Pujol, S. Escalera, L. Garrido, and F. Dantí.

Barcelona, Spain

Laura Igual
Santi Seguí

Contents

Authors and Contributors

About the Authors

Dr. Laura Igual is an associate professor from the Department of Mathematics and Computer Science at the Universitat de Barcelona. She received a degree in mathematics from Universitat de Valencia (Spain) in 2000 and a Ph.D. degree from the Universitat Pompeu Fabra (Spain) in 2006. Her particular areas of interest include computer vision, medical imaging, machine learning, and data science.

Dr. Laura Igual is coauthor of Chaps. 3, 6, and 8.

Dr. Santi Seguí is an assistant professor from the Department of Mathematics and Computer Science at the Universitat de Barcelona. He is a computer science engineer by the Universitat Autònoma de Barcelona (Spain) since 2007. He received his Ph.D. degree from the Universitat de Barcelona (Spain) in 2011. His particular areas of interest include computer vision, applied machine learning, and data science.

Dr. Santi Seguí is coauthor of Chaps. 8–10.

Contributors

Francesc Dantí is an adjunct professor and system administrator from the Department of Mathematics and Computer Science at the Universitat de Barcelona. He is a computer science engineer by the Universitat Oberta de Catalunya (Spain). His particular areas of interest are HPC and grid computing, parallel computing, and cybersecurity.

Francesc Dantí is coauthor of Chaps. 2 and 11.

Dr. Sergio Escalera is an associate professor from the Department of Mathematics and Computer Science at the Universitat de Barcelona. He is a computer science engineer by the Universitat Autònoma de Barcelona (Spain) since 2003. He received his Ph.D. degree from the Universitat Autònoma de Barcelona (Spain) in 2008. His research interests include, between others, statistical pattern recognition,

visual object recognition, with special interest in human pose recovery and behavior analysis from multimodal data.

Dr. Sergio Escalera is coauthor of Chaps. 4 and 10.

Dr. Lluís Garrido is an associate professor from the Department of Mathematics and Computer Science at the Universitat de Barcelona. He is a telecommunications engineer by the Universitat Politècnica de Catalunya (UPC) since 1996. He received his Ph.D. degree from the same university in 2002. His particular areas of interest include computer vision, image processing, numerical optimization, parallel computing, and data science.

Dr. Lluís Garrido is coauthor of Chap. 11.

Dr. Eloi Puertas is an assistant professor from the Department of Mathematics and Computer Science at the Universitat de Barcelona. He is a computer science engineer by the Universitat Autònoma de Barcelona (Spain) since 2002. He received his Ph.D. degree from the Universitat de Barcelona (Spain) in 2014. His particular areas of interest include artificial intelligence, software engineering, and data science.

Dr. Eloi Puertas is coauthor of Chaps. 2 and 9.

Dr. Oriol Pujol is a tenured associate professor from the Department of Mathematics and Computer Science at the Universitat de Barcelona. He received his Ph.D. degree from the Universitat Autònoma de Barcelona (Spain) in 2004 for his work in machine learning and computer vision. His particular areas of interest include machine learning, computer vision, and data science.

Dr. Oriol Pujol is coauthor of Chaps. 5 and 7.

Dr. Petia Radeva is a tenured associate professor and senior researcher from the Universitat de Barcelona. She graduated in applied mathematics and computer science in 1989 at the University of Sofia, Bulgaria, and received her Ph.D. degree in Computer Vision for Medical Imaging in 1998 from the Universitat Autònoma de Barcelona, Spain. She is Icrea Academia Researcher from 2015, head of the Consolidated Research Group "Computer Vision at the Universitat of Barcelona," and head of MiLab of Computer Vision Center. Her present research interests are on the development of learning-based approaches for computer vision, deep learning, egocentric vision, lifelogging, and data science.

Dr. Petia Radeva is coauthor of Chaps. 3, 5, and 7.

Dr. Jordi Vitrià is a full professor from the Department of Mathematics and Computer Science at the Universitat de Barcelona. He received his Ph.D. degree from the Universitat Autònoma de Barcelona in 1990. Dr. Jordi Vitrià has published more than 100 papers in SCI-indexed journals and has more than 25 years of experience in working on computer vision and artificial intelligence and its applications to several fields. He is now leader of the "Data Science Group at UB," a technology transfer unit that performs collaborative research projects between the Universitat de Barcelona and private companies.

Dr. Jordi Vitrià is coauthor of Chaps. 1, 4, and 6.

Introduction to Data Science

1

1.1 What is Data Science?

You have, no doubt, already experienced data science in several forms. When you are looking for information on the web by using a search engine or asking your mobile phone for directions, you are interacting with data science products. Data science has been behind resolving some of our most common daily tasks for several years.

Most of the scientific methods that power data science are not new and they have been out there, waiting for applications to be developed, for a long time. Statistics is an old science that stands on the shoulders of eighteenth-century giants such as Pierre Simon Laplace (1749–1827) and Thomas Bayes (1701–1761). Machine learning is younger, but it has already moved beyond its infancy and can be considered a well-established discipline. Computer science changed our lives several decades ago and continues to do so; but it cannot be considered new.

So, why is data science seen as a novel trend within business reviews, in technology blogs, and at academic conferences?

The novelty of data science is not rooted in the latest scientific knowledge, but in a disruptive change in our society that has been caused by the evolution of technology: datification. Datification is the process of rendering into data aspects of the world that have never been quantified before. At the personal level, the list of datified concepts is very long and still growing: business networks, the lists of books we are reading, the films we enjoy, the food we eat, our physical activity, our purchases, our driving behavior, and so on. Even our thoughts are datified when we publish them on our favorite social network; and in a not so distant future, your gaze could be datified by wearable vision registering devices. At the business level, companies are datifying semi-structured data that were previously discarded: web activity logs, computer network activity, machinery signals, etc. Nonstructured data, such as written reports, e-mails, or voice recordings, are now being stored not only for archive purposes but also to be analyzed.

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_1

However, datification is not the only ingredient of the data science revolution. The other ingredient is the democratization of data analysis. Large companies such as Google, Yahoo, IBM, or SAS were the only players in this field when data science had no name. At the beginning of the century, the huge computational resources of those companies allowed them to take advantage of datification by using analytical techniques to develop innovative products and even to take decisions about their own business. Today, the analytical gap between those companies and the rest of the world (companies and people) is shrinking. Access to cloud computing allows any individual to analyze huge amounts of data in short periods of time. Analytical knowledge is free and most of the crucial algorithms that are needed to create a solution can be found, because open-source development is the norm in this field. As a result, the possibility of using rich data to take evidence-based decisions is open to virtually any person or company.

Data science is commonly defined as a methodology by which actionable insights can be inferred from data. This is a subtle but important difference with respect to previous approaches to data analysis, such as business intelligence or exploratory statistics. Performing data science is a task with an ambitious objective: the production of beliefs informed by data and to be used as the basis of decision-making. In the absence of data, beliefs are uninformed and decisions, in the best of cases, are based on best practices or intuition. The representation of complex environments by rich data opens up the possibility of applying all the scientific knowledge we have regarding how to infer knowledge from data.

In general, data science allows us to adopt four different strategies to explore the world using data:

1. *Probing reality*. Data can be gathered by passive or by active methods. In the latter case, data represents the response of the world to our actions. Analysis of those responses can be extremely valuable when it comes to taking decisions about our subsequent actions. One of the best examples of this strategy is the use of A/B testing for web development: What is the best button size and color? The best answer can only be found by probing the world.
2. *Pattern discovery*. Divide and conquer is an old heuristic used to solve complex problems; but it is not always easy to decide how to apply this common sense to problems. Datified problems can be analyzed automatically to discover useful patterns and natural clusters that can greatly simplify their solutions. The use of this technique to profile users is a critical ingredient today in such important fields as programmatic advertising or digital marketing.
3. *Predicting future events*. Since the early days of statistics, one of the most important scientific questions has been how to build robust data models that are capable of predicting future data samples. Predictive analytics allows decisions to be taken in response to future events, not only reactively. Of course, it is not possible to predict the future in any environment and there will always be unpredictable events; but the identification of predictable events represents valuable knowledge. For example, predictive analytics can be used to optimize the tasks

planned for retail store staff during the following week, by analyzing data such as weather, historic sales, traffic conditions, etc.

4. *Understanding people and the world*. This is an objective that at the moment is beyond the scope of most companies and people, but large companies and governments are investing considerable amounts of money in research areas such as understanding natural language, computer vision, psychology and neuroscience. Scientific understanding of these areas is important for data science because in the end, in order to take optimal decisions, it is necessary to know the real processes that drive people's decisions and behavior. The development of deep learning methods for natural language understanding and for visual object recognition is a good example of this kind of research.

1.2 About This Book

Data science is definitely a cool and trendy discipline that routinely appears in the headlines of very important newspapers and on TV stations. Data scientists are presented in those forums as a scarce and expensive resource. As a result of this situation, data science can be perceived as a complex and scary discipline that is only accessible to a reduced set of geniuses working for major companies. The main purpose of this book is to demystify data science by describing a set of tools and techniques that allows a person with basic skills in computer science, mathematics, and statistics to perform the tasks commonly associated with data science.

To this end, this book has been written under the following assumptions:

- Data science is a complex, multifaceted field that can be approached from several points of view: ethics, methodology, business models, how to deal with big data, data engineering, data governance, etc. Each point of view deserves a long and interesting discussion, but the approach adopted in this book focuses on analytical techniques, because such techniques constitute the core toolbox of every data scientist and because they are the key ingredient in predicting future events, discovering useful patterns, and probing the world.
- You have some experience with Python programming. For this reason, we do not offer an introduction to the language. But even if you are new to Python, this should not be a problem. Before reading this book you should start with any online Python course. Mastering Python is not easy, but acquiring the basics is a manageable task for anyone in a short period of time.
- Data science is about evidence-based storytelling and this kind of process requires appropriate tools. The Python data science toolbox is one, not the only, of the most developed environments for doing data science. You can easily install all you need by using Anaconda[1]: a free product that includes a programming language

[1] https://www.continuum.io/downloads.

(Python), an interactive environment to develop and present data science projects (Jupyter notebooks), and most of the toolboxes necessary to perform data analysis.
- Learning by doing is the best approach to learn data science. For this reason all the code examples and data in this book are available to download at https://github.com/DataScienceUB/introduction-datascience-python-book.
- Data science deals with solving real-world problems. So all the chapters in the book include and discuss practical cases using real data.

This book includes three different kinds of chapters. The first kind is about Python extensions. Python was originally designed to have a minimum number of data objects (int, float, string, etc.); but when dealing with data, it is necessary to extend the native set to more complex objects such as (numpy) numerical arrays or (pandas) data frames. The second kind of chapter includes techniques and modules to perform statistical analysis and machine learning. Finally, there are some chapters that describe several applications of data science, such as building recommenders or sentiment analysis. The composition of these chapters was chosen to offer a panoramic view of the data science field, but we encourage the reader to delve deeper into these topics and to explore those topics that have not been covered: big data analytics, deep learning techniques, and more advanced mathematical and statistical methods (e.g., computational algebra and Bayesian statistics).

Acknowledgements This chapter was co-written by Jordi Vitrià.

Toolboxes for Data Scientists

2

2.1 Introduction

In this chapter, first we introduce some of the tools that data scientists use. The toolbox of any data scientist, as for any kind of programmer, is an essential ingredient for success and enhanced performance. Choosing the right tools can save a lot of time and thereby allow us to focus on data analysis.

The most basic tool to decide on is which programming language we will use. Many people use only one programming language in their entire life: the first and only one they learn. For many, learning a new language is an enormous task that, if at all possible, should be undertaken only once. The problem is that some languages are intended for developing high-performance or production code, such as C, C++, or Java, while others are more focused on prototyping code, among these the best known are the so-called scripting languages: Ruby, Perl, and Python. So, depending on the first language you learned, certain tasks will, at the very least, be rather tedious. The main problem of being stuck with a single language is that many basic tools simply will not be available in it, and eventually you will have either to reimplement them or to create a bridge to use some other language just for a specific task.

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_2

In conclusion, you either have to be ready to change to the best language for each task and then glue the results together, or choose a very flexible language with a rich ecosystem (e.g., third-party open-source libraries). In this book we have selected Python as the programming language.

2.2 Why Python?

Python[1] is a mature programming language but it also has excellent properties for newbie programmers, making it ideal for people who have never programmed before. Some of the most remarkable of those properties are easy to read code, suppression of non-mandatory delimiters, dynamic typing, and dynamic memory usage. Python is an interpreted language, so the code is executed immediately in the Python console without needing the compilation step to machine language. Besides the Python console (which comes included with any Python installation) you can find other interactive consoles, such as IPython,[2] which give you a richer environment in which to execute your Python code.

Currently, Python is one of the most flexible programming languages. One of its main characteristics that makes it so flexible is that it can be seen as a multiparadigm language. This is especially useful for people who already know how to program with other languages, as they can rapidly start programming with Python in the same way. For example, Java programmers will feel comfortable using Python as it supports the object-oriented paradigm, or C programmers could mix Python and C code using *cython*. Furthermore, for anyone who is used to programming in functional languages such as Haskell or Lisp, Python also has basic statements for functional programming in its own core library.

In this book, we have decided to use Python language because, as explained before, it is a mature language programming, easy for the newbies, and can be used as a specific platform for data scientists, thanks to its large ecosystem of scientific libraries and its high and vibrant community. Other popular alternatives to Python for data scientists are R and MATLAB/Octave.

2.3 Fundamental Python Libraries for Data Scientists

The Python community is one of the most active programming communities with a huge number of developed toolboxes. The most popular Python toolboxes for any data scientist are NumPy, SciPy, Pandas, and Scikit-Learn.

[1] https://www.python.org/downloads/.
[2] http://ipython.org/install.html.

2.3.1 Numeric and Scientific Computation: NumPy and SciPy

NumPy[3] is the cornerstone toolbox for scientific computing with Python. NumPy provides, among other things, support for multidimensional arrays with basic operations on them and useful linear algebra functions. Many toolboxes use the NumPy array representations as an efficient basic data structure. Meanwhile, *SciPy* provides a collection of numerical algorithms and domain-specific toolboxes, including signal processing, optimization, statistics, and much more. Another core toolbox in SciPy is the plotting library *Matplotlib*. This toolbox has many tools for data visualization.

2.3.2 SCIKIT-Learn: Machine Learning in Python

Scikit-learn[4] is a machine learning library built from NumPy, SciPy, and Matplotlib. Scikit-learn offers simple and efficient tools for common tasks in data analysis such as classification, regression, clustering, dimensionality reduction, model selection, and preprocessing.

2.3.3 PANDAS: Python Data Analysis Library

Pandas[5] provides high-performance data structures and data analysis tools. The key feature of Pandas is a fast and efficient DataFrame object for data manipulation with integrated indexing. The DataFrame structure can be seen as a spreadsheet which offers very flexible ways of working with it. You can easily transform any dataset in the way you want, by reshaping it and adding or removing columns or rows. It also provides high-performance functions for aggregating, merging, and joining datasets. Pandas also has tools for importing and exporting data from different formats: comma-separated value (CSV), text files, Microsoft Excel, SQL databases, and the fast HDF5 format. In many situations, the data you have in such formats will not be complete or totally structured. For such cases, Pandas offers handling of missing data and intelligent data alignment. Furthermore, Pandas provides a convenient Matplotlib interface.

2.4 Data Science Ecosystem Installation

Before we can get started on solving our own data-oriented problems, we will need to set up our programming environment. The first question we need to answer concerns

[3] http://www.scipy.org/scipylib/download.html.
[4] http://www.scipy.org/scipylib/download.html.
[5] http://pandas.pydata.org/getpandas.html.

Python language itself. There are currently two different versions of Python: Python 2.X and Python 3.X. The differences between the versions are important, so there is no compatibility between the codes, i.e., code written in Python 2.X does not work in Python 3.X and vice versa. Python 3.X was introduced in late 2008; by then, a lot of code and many toolboxes were already deployed using Python 2.X (Python 2.0 was initially introduced in 2000). Therefore, much of the scientific community did not change to Python 3.0 immediately and they were stuck with Python 2.7. By now, almost all libraries have been ported to Python 3.0; but Python 2.7 is still maintained, so one or another version can be chosen. However, those who already have a large amount of code in 2.X rarely change to Python 3.X. In our examples throughout this book we will use Python 2.7.

Once we have chosen one of the Python versions, the next thing to decide is whether we want to install the data scientist Python ecosystem by individual toolboxes, or to perform a bundle installation with all the needed toolboxes (and a lot more). For newbies, the second option is recommended. If the first option is chosen, then it is only necessary to install all the mentioned toolboxes in the previous section, in exactly that order.

However, if a bundle installation is chosen, the Anaconda Python distribution[6] is then a good option. The Anaconda distribution provides integration of all the Python toolboxes and applications needed for data scientists into a single directory without mixing it with other Python toolboxes installed on the machine. It contains, of course, the core toolboxes and applications such as NumPy, Pandas, SciPy, Matplotlib, Scikit-learn, IPython, Spyder, etc., but also more specific tools for other related tasks such as data visualization, code optimization, and big data processing.

2.5 Integrated Development Environments (IDE)

For any programmer, and by extension, for any data scientist, the integrated development environment (IDE) is an essential tool. IDEs are designed to maximize programmer productivity. Thus, over the years this software has evolved in order to make the coding task less complicated. Choosing the right IDE for each person is crucial and, unfortunately, there is no "one-size-fits-all" programming environment. The best solution is to try the most popular IDEs among the community and keep whichever fits better in each case.

In general, the basic pieces of any IDE are three: the editor, the compiler, (or interpreter) and the debugger. Some IDEs can be used in multiple programming languages, provided by language-specific plugins, such as Netbeans[7] or Eclipse.[8] Others are only specific for one language or even a specific programming task. In

[6] http://continuum.io/downloads.
[7] https://netbeans.org/downloads/.
[8] https://eclipse.org/downloads/.

the case of Python, there are a large number of specific IDEs, both commercial (PyCharm,[9] WingIDE[10] …) and open-source. The open-source community helps IDEs to spring up, thus anyone can customize their own environment and share it with the rest of the community. For example, Spyder[11] (Scientific Python Development EnviRonment) is an IDE customized with the task of the data scientist in mind.

2.5.1 Web Integrated Development Environment (WIDE): Jupyter

With the advent of web applications, a new generation of IDEs for interactive languages such as Python has been developed. Starting in the academia and e-learning communities, web-based IDEs were developed considering how not only your code but also all your environment and executions can be stored in a server. One of the first applications of this kind of WIDE was developed by William Stein in early 2005 using Python 2.3 as part of his SageMath mathematical software. In SageMath, a server can be set up in a center, such as a university or school, and then students can work on their homework either in the classroom or at home, starting from exactly the same point they left off. Moreover, students can execute all the previous steps over and over again, and then change some particular *code cell* (a segment of the document that may content source code that can be executed) and execute the operation again. Teachers can also have access to student sessions and review the progress or results of their pupils.

Nowadays, such sessions are called notebooks and they are not only used in classrooms but also used to show results in presentations or on business dashboards. The recent spread of such notebooks is mainly due to IPython. Since December 2011, IPython has been issued as a browser version of its interactive console, called IPython notebook, which shows the Python execution results very clearly and concisely by means of cells. Cells can contain content other than code. For example, markdown (a wiki text language) cells can be added to introduce algorithms. It is also possible to insert Matplotlib graphics to illustrate examples or even web pages. Recently, some scientific journals have started to accept notebooks in order to show experimental results, complete with their code and data sources. In this way, experiments can become completely and absolutely replicable.

Since the project has grown so much, IPython notebook has been separated from IPython software and now it has become a part of a larger project: Jupyter[12]. Jupyter (for Julia, Python and R) aims to reuse the same WIDE for all these interpreted languages and not just Python. All old IPython notebooks are automatically imported to the new version when they are opened with the Jupyter platform; but once they

[9] https://www.jetbrains.com/pycharm/.
[10] https://wingware.com/.
[11] https://github.com/spyder-ide/spyder.
[12] http://jupyter.readthedocs.org/en/latest/install.html.

are converted to the new version, they cannot be used again in old IPython notebook versions.

In this book, all the examples shown use Jupyter notebook style.

2.6 Get Started with Python for Data Scientists

Throughout this book, we will come across many practical examples. In this chapter, we will see a very basic example to help get started with a data science ecosystem from scratch. To execute our examples, we will use Jupyter notebook, although any other console or IDE can be used.

The Jupyter Notebook Environment

Once all the ecosystem is fully installed, we can start by launching the Jupyter notebook platform. This can be done directly by typing the following command on your terminal or command line: `$ jupyter notebook`

If we chose the bundle installation, we can start the Jupyter notebook platform by clicking on the Jupyter Notebook icon installed by Anaconda in the start menu or on the desktop.

The browser will immediately be launched displaying the Jupyter notebook home-page, whose URL is http://localhost:8888/tree. Note that a special port is used; by default it is 8888. As can be seen in Fig. 2.1, this initial page displays a tree view of a directory. If we use the command line, the root directory is the same directory where we launched the Jupyter notebook. Otherwise, if we use the Anaconda launcher, the root directory is the current user directory. Now, to start a new notebook, we only need to press the New Notebooks Python 2 button at the top on the right of the home page.

As can be seen in Fig. 2.2, a blank notebook is created called `Untitled`. First of all, we are going to change the name of the notebook to something more appropriate. To do this, just click on the notebook name and rename it: `DataScience-GetStartedExample`.

Let us begin by importing those toolboxes that we will need for our program. In the first cell we put the code to import the *Pandas* library as `pd`. This is for convenience; every time we need to use some functionality from the Pandas library, we will write `pd` instead of `pandas`. We will also import the two core libraries mentioned above: the numpy library as `np` and the matplotlib library as `plt`.

In []:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

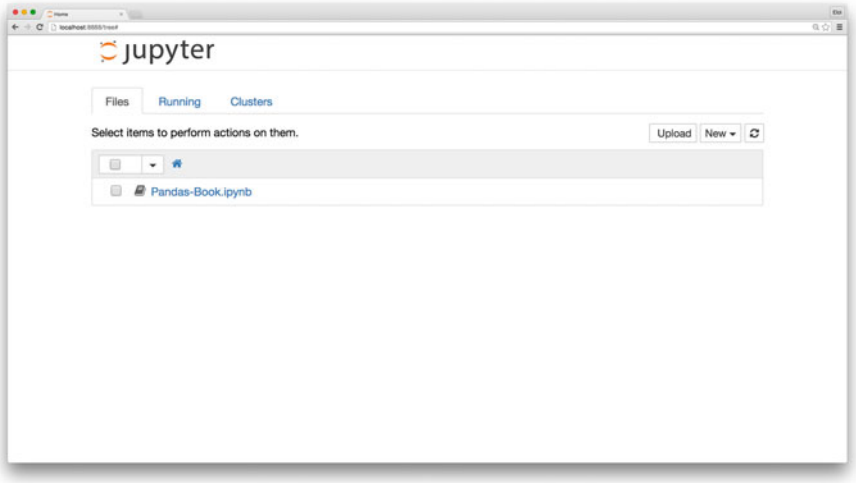


Fig. 2.1 IPython notebook home page, displaying a home tree directory

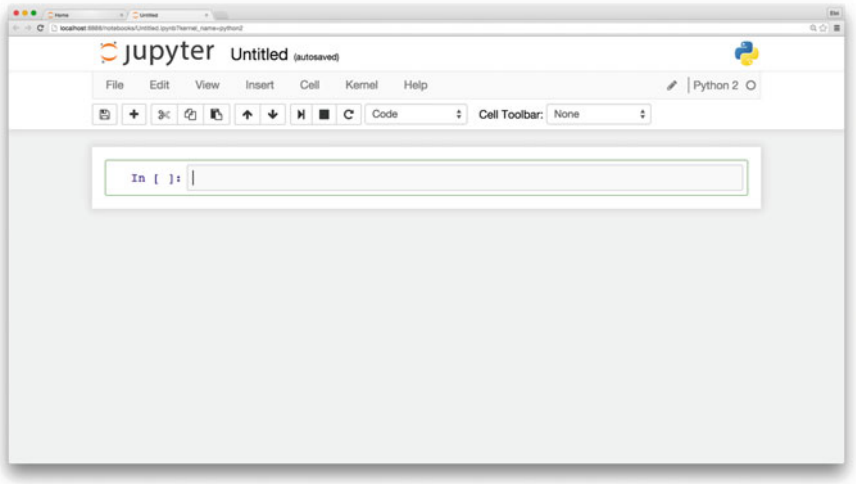


Fig. 2.2 An empty new notebook

To execute just one cell, we press the ▶I button or click on Cell > Run or press the keys Ctrl + Enter. While execution is underway, the header of the cell shows the * mark:

In [*]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

While a cell is being executed, no other cell can be executed. If you try to execute another cell, its execution will not start until the first cell has finished its execution.

Once the execution is finished, the header of the cell will be replaced by the next number of execution. Since this will be the first cell executed, the number shown will be 1. If the process of importing the libraries is correct, no output cell is produced.

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

For simplicity, other chapters in this book will avoid writing these imports.

The DataFrame Data Structure

The key data structure in Pandas is the `DataFrame` object. A DataFrame is basically a tabular data structure, with rows and columns. Rows have a specific index to access them, which can be any name or value. In Pandas, the columns are called `Series`, a special type of data, which in essence consists of a list of several values, where each value has an index. Therefore, the DataFrame data structure can be seen as a spreadsheet, but it is much more flexible. To understand how it works, let us see how to create a DataFrame from a common Python dictionary of lists. First, we will create a new cell by clicking Insert > Insert Cell Below or pressing the keys Ctrl + B. Then, we write in the following code:

In [2]:
```
data = {'year': [
         2010, 2011, 2012,
         2010, 2011, 2012,
         2010, 2011, 2012
     ],
     'team': [
         'FCBarcelona', 'FCBarcelona',
         'FCBarcelona', 'RMadrid',
         'RMadrid', 'RMadrid',
         'ValenciaCF', 'ValenciaCF',
         'ValenciaCF'
     ],
     'wins':   [30, 28, 32, 29, 32, 26, 21, 17, 19],
     'draws':  [6, 7, 4, 5, 4, 7, 8, 10, 8],
     'losses': [2, 3, 2, 4, 2, 5, 9, 11, 11]
     }
football = pd.DataFrame(data, columns = [
     'year', 'team', 'wins', 'draws', 'losses'
     ]
)
```

In this example, we use the pandas DataFrame object constructor with a dictionary of lists as argument. The value of each entry in the dictionary is the name of the column, and the lists are their values.

The DataFrame columns can be arranged at construction time by entering a keyword `columns` with a list of the names of the columns ordered as we want. If the

column keyword is not present in the constructor, the columns will be arranged in alphabetical order. Now, if we execute this cell, the result will be a table like this:

Out[2]:

	year	team	wins	draws	losses
0	2010	FCBarcelona	30	6	2
1	2011	FCBarcelona	28	7	3
2	2012	FCBarcelona	32	4	2
3	2010	RMadrid	29	5	4
4	2011	RMadrid	32	4	2
5	2012	RMadrid	26	7	5
6	2010	ValenciaCF	21	8	9
7	2011	ValenciaCF	17	10	11
8	2012	ValenciaCF	19	8	11

where each entry in the dictionary is a column. The index of each row is created automatically taking the position of its elements inside the entry lists, starting from 0. Although it is very easy to create DataFrames from scratch, most of the time what we will need to do is import chunks of data into a DataFrame structure, and we will see how to do this in later examples.

Apart from DataFrame data structure creation, Panda offers a lot of functions to manipulate them. Among other things, it offers us functions for aggregation, manipulation, and transformation of the data. In the following sections, we will introduce some of these functions.

Open Government Data Analysis Example Using Pandas

To illustrate how we can use Pandas in a simple real problem, we will start doing some basic analysis of government data. For the sake of transparency, data produced by government entities must be open, meaning that they can be freely used, reused, and distributed by anyone. An example of this is the Eurostat, which is the home of European Commission data. Eurostat's main role is to process and publish comparable statistical information at the European level. The data in Eurostat are provided by each member state and it is free to reuse them, for both noncommercial and commercial purposes (with some minor exceptions).

Since the amount of data in the Eurostat database is huge, in our first study we are only going to focus on data relative to indicators of educational funding by the member states. Thus, the first thing to do is to retrieve such data from Eurostat. Since open data have to be delivered in a plain text format, CSV (or any other delimiter-separated value) formats are commonly used to store tabular data. In a delimiter-separated value file, each line is a data record and each record consists of one or more fields, separated by the delimiter character (usually a comma). Therefore, the data we will use can be found already processed at book's Github repository as `educ_figdp_1_Data.csv` file. Of course, it can also be downloaded as unprocessed tabular data from the Eurostat database site[13] following the path:

Tables by themes » Population and social conditions » Education and training » Education » Indicators on education finance » Public expenditure on education.

2.6.1 Reading

Let us start reading the data we downloaded. First of all, we have to create a new notebook called `Open Government Data Analysis` and open it. Then, after ensuring that the `educ_figdp_1_Data.csv` file is stored in the same directory as our notebook directory, we will write the following code to read and show the content:

In [1]:
```
edu = pd.read_csv('files/ch02/educ_figdp_1_Data.csv',
                  na_values = ':',
                  usecols = ["TIME","GEO","Value"])
edu
```

Out[1]:

	TIME	GEO	Value
0	2000	European Union ...	NaN
1	2001	European Union ...	NaN
2	2002	European Union ...	5.00
3	2003	European Union ...	5.03
...	...	...	...
382	2010	Finland	6.85
383	2011	Finland	6.76

384 rows × 5 columns

The way to read CSV (or any other separated value, providing the separator character) files in Pandas is by calling the `read_csv` method. Besides the name of the file, we add the `na_values` key argument to this method along with the character that represents "non available data" in the file. Normally, CSV files have a header with the names of the columns. If this is the case, we can use the `usecols` parameter to select which columns in the file will be used.

In this case, the DataFrame resulting from reading our data is stored in `edu`. The output of the execution shows that the `edu` DataFrame size is 384 rows × 3 columns. Since the DataFrame is too large to be fully displayed, three dots appear in the middle of each row.

Beside this, Pandas also has functions for reading files with formats such as Excel, HDF5, tabulated files, or even the content from the clipboard (`read_excel()`, `read_hdf()`, `read_table()`, `read_clipboard()`). Whichever function we use, the result of reading a file is stored as a DataFrame structure.

To see how the data looks, we can use the `head()` method, which shows just the first five rows. If we use a number as an argument to this method, this will be the number of rows that will be listed:

[13] http://ec.europa.eu/eurostat/data/database.

In [2]:
```
edu.head()
```

Out[2]:

	TIME	GEO	Value
0	2000	European Union ...	NaN
1	2001	European Union ...	NaN
2	2002	European Union ...	5.00
3	2003	European Union ...	5.03
4	2004	European Union ...	4.95

Similarly, it exists the `tail()` method, which returns the last five rows by default.

In [3]:
```
edu.tail()
```

Out[3]:

379	2007	Finland	5.90
380	2008	Finland	6.10
381	2009	Finland	6.81
382	2010	Finland	6.85
383	2011	Finland	6.76

If we want to know the names of the columns or the names of the indexes, we can use the DataFrame attributes `columns` and `index` respectively. The names of the columns or indexes can be changed by assigning a new list of the same length to these attributes. The values of any DataFrame can be retrieved as a Python array by calling its `values` attribute.

If we just want quick statistical information on all the numeric columns in a DataFrame, we can use the function `describe()`. The result shows the count, the mean, the standard deviation, the minimum and maximum, and the percentiles, by default, the 25th, 50th, and 75th, for all the values in each column or series.

In [4]:
```
edu.describe()
```

Out[4]:

	TIME	Value
count	384.000000	361.000000
mean	2005.500000	5.203989
std	3.456556	1.021694
min	2000.000000	2.880000
25%	2002.750000	4.620000
50%	2005.500000	5.060000
75%	2008.250000	5.660000
max	2011.000000	8.810000

Name: Value, dtype: float64

2.6.2 Selecting Data

If we want to select a subset of data from a DataFrame, it is necessary to indicate this subset using square brackets (`[ ]`) after the DataFrame. The subset can be specified in several ways. If we want to select only one column from a DataFrame, we only need to put its name between the square brackets. The result will be a Series data structure, not a DataFrame, because only one column is retrieved.

In [5]:
```
edu['Value']
```

Out[5]:
```
0    NaN
1    NaN
2   5.00
3   5.03
4   4.95
...  ...
380 6.10
381 6.81
382 6.85
383 6.76
Name: Value, dtype: float64
```

If we want to select a subset of rows from a DataFrame, we can do so by indicating a range of rows separated by a colon (`:`) inside the square brackets. This is commonly known as a *slice* of rows:

In [6]:
```
edu[10:14]
```

Out[6]:

	TIME	GEO	Value
10	2010	European Union (28 countries)	5.41
11	2011	European Union (28 countries)	5.25
12	2000	European Union (27 countries)	4.91
13	2001	European Union (27 countries)	4.99

This instruction returns the slice of rows from the 10th to the 13th position. Note that the slice does not use the index labels as references, but the position. In this case, the labels of the rows simply coincide with the position of the rows.

If we want to select a subset of columns and rows using the labels as our references instead of the positions, we can use `ix` indexing:

In [7]:
```
edu.ix[90:94, ['TIME','GEO']]
```

Out[7]:

	TIME	GEO
90	2006	Belgium
91	2007	Belgium
92	2008	Belgium
93	2009	Belgium
94	2010	Belgium

This returns all the rows between the indexes specified in the slice before the comma, and the columns specified as a list after the comma. In this case, `ix` references the index labels, which means that `ix` does not return the 90th to 94th rows, but it returns all the rows between the row labeled 90 and the row labeled 94; thus if the index 100 is placed between the rows labeled as 90 and 94, this row would also be returned.

2.6.3 Filtering Data

Another way to select a subset of data is by applying Boolean indexing. This indexing is commonly known as a *filter*. For instance, if we want to filter those values less than or equal to 6.5, we can do it like this:

In [8]:

```
edu[edu['Value'] > 6.5].tail()
```

Out[8]:

	TIME	GEO	Value
218	2002	Cyprus	6.60
281	2005	Malta	6.58
94	2010	Belgium	6.58
93	2009	Belgium	6.57
95	2011	Belgium	6.55

Boolean indexing uses the result of a Boolean operation over the data, returning a mask with True or False for each row. The rows marked True in the mask will be selected. In the previous example, the Boolean operation `edu['Value'] > 6.5` produces a Boolean mask. When an element in the "Value" column is greater than 6.5, the corresponding value in the mask is set to True, otherwise it is set to False. Then, when this mask is applied as an index in `edu[edu['Value'] > 6.5]`, the result is a filtered DataFrame containing only rows with values higher than 6.5. Of course, any of the usual Boolean operators can be used for filtering: $<$ (less than), $<=$ (less than or equal to), $>$ (greater than), $>=$ (greater than or equal to), $==$ (equal to), and $!=$ (not equal to).

2.6.4 Filtering Missing Values

Pandas uses the special value `NaN` (not a number) to represent missing values. In Python, `NaN` is a special floating-point value returned by certain operations when

Table 2.1 List of most common aggregation functions

Function	Description
count()	Number of non-null observations
sum()	Sum of values
mean()	Mean of values
median()	Arithmetic median of values
min()	Minimum
max()	Maximum
prod()	Product of values
std()	Unbiased standard deviation
var()	Unbiased variance

one of their results ends in an undefined value. A subtle feature of `NaN` values is that two `NaN` are never equal. Because of this, the only safe way to tell whether a value is missing in a DataFrame is by using the `isnull()` function. Indeed, this function can be used to filter rows with missing values:

In [9]:
```
edu[edu["Value"].isnull()].head()
```

Out[9]:

	TIME	GEO	Value
0	2000	European Union (28 countries)	NaN
1	2001	European Union (28 countries)	NaN
36	2000	Euro area (18 countries)	NaN
37	2001	Euro area (18 countries)	NaN
48	2000	Euro area (17 countries)	NaN

2.6.5 Manipulating Data

Once we know how to select the desired data, the next thing we need to know is how to manipulate data. One of the most straightforward things we can do is to operate with columns or rows using aggregation functions. Table 2.1 shows a list of the most common aggregation functions. The result of all these functions applied to a row or column is always a number. Meanwhile, if a function is applied to a DataFrame or a selection of rows and columns, then you can specify if the function should be applied to the rows for each column (setting the `axis=0` keyword on the invocation of the function), or it should be applied on the columns for each row (setting the `axis=1` keyword on the invocation of the function).

In [10]:
```
edu.max(axis = 0)
```

Out[10]:
```
TIME          2011
GEO          Spain
Value         8.81
dtype: object
```

Note that these are functions specific to Pandas, not the generic Python functions. There are differences in their implementation. In Python, `NaN` values propagate through all operations without raising an exception. In contrast, Pandas operations exclude `NaN` values representing missing data. For example, the pandas `max` function excludes `NaN` values, thus they are interpreted as missing values, while the standard Python `max` function will take the mathematical interpretation of `NaN` and return it as the maximum:

In [11]:
```
print "Pandas max function:", edu['Value'].max()
print "Python max function:", max(edu['Value'])
```

Out[11]:
```
Pandas max function: 8.81
Python max function: nan
```

Beside these aggregation functions, we can apply operations over all the values in rows, columns or a selection of both. The rule of thumb is that an operation between columns means that it is applied to each row in that column and an operation between rows means that it is applied to each column in that row. For example we can apply any binary arithmetical operation (+,-,*,/) to an entire row:

In [12]:
```
s = edu["Value"]/100
s.head()
```

Out[12]:
```
0          NaN
1          NaN
2       0.0500
3       0.0503
4       0.0495
Name: Value, dtype: float64
```

However, we can apply any function to a DataFrame or Series just setting its name as argument of the `apply` method. For example, in the following code, we apply the `sqrt` function from the NumPy library to perform the square root of each value in the `Value` column.

In [13]:
```
s = edu["Value"].apply(np.sqrt)
s.head()
```

Out[13]:
```
0            NaN
1            NaN
2       2.236068
3       2.242766
4       2.224860
Name: Value, dtype: float64
```

If we need to design a specific function to apply it, we can write an in-line function, commonly known as a λ-function. A λ-function is a function without a name. It is only necessary to specify the parameters it receives, between the `lambda` keyword and the colon (`:`). In the next example, only one parameter is needed, which will be the value of each element in the `Value` column. The value the function returns will be the square of that value.

In [14]:
```
s = edu["Value"].apply(lambda d: d**2)
s.head()
```

Out[14]:
```
0          NaN
1          NaN
2      25.0000
3      25.3009
4      24.5025
Name: Value, dtype: float64
```

Another basic manipulation operation is to set new values in our DataFrame. This can be done directly using the assign operator (=) over a DataFrame. For example, to add a new column to a DataFrame, we can assign a Series to a selection of a column that does not exist. This will produce a new column in the DataFrame after all the others. You must be aware that if a column with the same name already exists, the previous values will be overwritten. In the following example, we assign the Series that results from dividing the `Value` column by the maximum value in the same column to a new column named `ValueNorm`.

In [15]:
```
edu['ValueNorm'] = edu['Value']/edu['Value'].max()
edu.tail()
```

Out[15]:

	TIME	GEO	Value	ValueNorm
379	2007	Finland	5.90	0.669694
380	2008	Finland	6.10	0.692395
381	2009	Finland	6.81	0.772985
382	2010	Finland	6.85	0.777526
383	2011	Finland	6.76	0.767310

Now, if we want to remove this column from the DataFrame, we can use the `drop` function; this removes the indicated rows if `axis=0`, or the indicated columns if `axis=1`. In Pandas, all the functions that change the contents of a DataFrame, such as the drop function, will normally return a copy of the modified data, instead of overwriting the DataFrame. Therefore, the original DataFrame is kept. If you do not want to keep the old values, you can set the keyword `inplace` to `True`. By default, this keyword is set to `False`, meaning that a copy of the data is returned.

In [16]:
```
edu.drop('ValueNorm', axis = 1, inplace = True)
edu.head()
```

Out[16]:

	TIME	GEO	Value
0	2000	European Union (28 countries)	NaN
1	2001	European Union (28 countries)	NaN
2	2002	European Union (28 countries)	5
3	2003	European Union (28 countries)	5.03
4	2004	European Union (28 countries)	4.95

Instead, if what we want to do is to insert a new row at the bottom of the DataFrame, we can use the Pandas `append` function. This function receives as argument the new row, which is represented as a dictionary where the keys are the name of the columns and the values are the associated value. You must be aware to setting the `ignore_index` flag in the `append` method to `True`, otherwise the index 0 is given to this new row, which will produce an error if it already exists:

In [17]:

```
edu = edu.append({"TIME": 2000,"Value": 5.00,"GEO": 'a'},
                 ignore_index = True)
edu.tail()
```

Out[17]:

	TIME	GEO	Value
380	2008	Finland	6.1
381	2009	Finland	6.81
382	2010	Finland	6.85
383	2011	Finland	6.76
384	2000	a	5

Finally, if we want to remove this row, we need to use the `drop` function again. Now we have to set the axis to 0, and specify the index of the row we want to remove. Since we want to remove the last row, we can use the max function over the indexes to determine which row is.

In [18]:

```
edu.drop(max(edu.index), axis = 0, inplace = True)
edu.tail()
```

Out[18]:

	TIME	GEO	Value
379	2007	Finland	5.9
380	2008	Finland	6.1
381	2009	Finland	6.81
382	2010	Finland	6.85
383	2011	Finland	6.76

The `drop()` function is also used to remove missing values by applying it over the result of the `isnull()` function. This has a similar effect to filtering the `NaN` values, as we explained above, but here the difference is that a copy of the DataFrame without the `NaN` values is returned, instead of a view.

In [19]:

```
eduDrop = edu.drop(edu["Value"].isnull(), axis = 0)
eduDrop.head()
```

Out[19]:

	TIME	GEO	Value
2	2002	European Union (28 countries)	5.00
3	2003	European Union (28 countries)	5.03
4	2004	European Union (28 countries)	4.95
5	2005	European Union (28 countries)	4.92
6	2006	European Union (28 countries)	4.91

To remove NaN values, instead of the generic drop function, we can use the specific `dropna()` function. If we want to erase any row that contains an NaN value, we have to set the `how` keyword to `any`. To restrict it to a subset of columns, we can specify it using the `subset` keyword. As we can see below, the result will be the same as using the `drop` function:

In [20]:

```
eduDrop = edu.dropna(how = 'any', subset = ["Value"])
eduDrop.head()
```

Out[20]:

	TIME	GEO	Value
2	2002	European Union (28 countries)	5.00
3	2003	European Union (28 countries)	5.03
4	2004	European Union (28 countries)	4.95
5	2005	European Union (28 countries)	4.92
6	2006	European Union (28 countries)	4.91

If, instead of removing the rows containing NaN, we want to fill them with another value, then we can use the `fillna()` method, specifying which value has to be used. If we want to fill only some specific columns, we have to set as argument to the `fillna()` function a dictionary with the name of the columns as the key and which character to be used for filling as the value.

In [21]:

```
eduFilled = edu.fillna(value = {"Value": 0})
eduFilled.head()
```

Out[21]:

	TIME	GEO	Value
0	2000	European Union (28 countries)	0.00
1	2001	European Union (28 countries)	0.00
2	2002	European Union (28 countries)	5.00
3	2003	European Union (28 countries)	4.95
4	2004	European Union (28 countries)	4.95

2.6.6 Sorting

Another important functionality we will need when inspecting our data is to sort by columns. We can sort a DataFrame using any column, using the `sort` function. If we want to see the first five rows of data sorted in descending order (i.e., from the largest to the smallest values) and using the `Value` column, then we just need to do this:

In [22]:
```
edu.sort_values(by = 'Value', ascending = False,
                inplace = True)
edu.head()
```

Out[22]:

	TIME	GEO	Value
130	2010	Denmark	8.81
131	2011	Denmark	8.75
129	2009	Denmark	8.74
121	2001	Denmark	8.44
122	2002	Denmark	8.44

Note that the `inplace` keyword means that the DataFrame will be overwritten, and hence no new DataFrame is returned. If instead of `ascending = False` we use `ascending = True`, the values are sorted in ascending order (i.e., from the smallest to the largest values).

If we want to return to the original order, we can sort by an index using the `sort_index` function and specifying `axis=0`:

In [23]:
```
edu.sort_index(axis = 0, ascending = True, inplace = True)
edu.head()
```

Out[23]:

	TIME	GEO	Value
0	2000	European Union ...	NaN
1	2001	European Union ...	NaN
2	2002	European Union ...	5.00
3	2003	European Union ...	5.03
4	2004	European Union ...	4.95

2.6.7 Grouping Data

Another very useful way to inspect data is to group it according to some criteria. For instance, in our example it would be nice to group all the data by country, regardless of the year. Pandas has the `groupby` function that allows us to do exactly this. The value returned by this function is a special grouped DataFrame. To have a proper DataFrame as a result, it is necessary to apply an aggregation function. Thus, this function will be applied to all the values in the same group.

For example, in our case, if we want a DataFrame showing the mean of the values for each country over all the years, we can obtain it by grouping according to country and using the mean function as the aggregation method for each group. The result would be a DataFrame with countries as indexes and the mean values as the column:

In [24]:
```
group = edu[["GEO", "Value"]].groupby('GEO').mean()
group.head()
```

Out[24]:

	Value
GEO	
Austria	5.618333
Belgium	6.189091
Bulgaria	4.093333
Cyprus	7.023333
Czech Republic	4.16833

2.6.8 Rearranging Data

Up until now, our indexes have been just a numeration of rows without much meaning. We can transform the arrangement of our data, redistributing the indexes and columns for better manipulation of our data, which normally leads to better performance. We can rearrange our data using the `pivot_table` function. Here, we can specify which columns will be the new indexes, the new values, and the new columns.

For example, imagine that we want to transform our DataFrame to a spreadsheet-like structure with the country names as the index, while the columns will be the years starting from 2006 and the values will be the previous `Value` column. To do this, first we need to filter out the data and then pivot it in this way:

In [25]:

```
filtered_data = edu[edu["TIME"] > 2005]
pivedu = pd.pivot_table(filtered_data, values = 'Value',
                        index = ['GEO'],
                        columns = ['TIME'])
pivedu.head()
```

Out[25]:

TIME	2006	2007	2008	2009	2010	2011
GEO						
Austria	5.40	5.33	5.47	5.98	5.91	5.80
Belgium	5.98	6.00	6.43	6.57	6.58	6.55
Bulgaria	4.04	3.88	4.44	4.58	4.10	3.82
Cyprus	7.02	6.95	7.45	7.98	7.92	7.87
Czech Republic	4.42	4.05	3.92	4.36	4.25	4.51

Now we can use the new index to select specific rows by label, using the `ix` operator:

In [26]:

```
pivedu.ix[['Spain','Portugal'], [2006,2011]]
```

Out[26]:

TIME	2006	2011
GEO		
Spain	4.26	4.82
Portugal	5.07	5.27

Pivot also offers the option of providing an argument `aggr_function` that allows us to perform an aggregation function between the values if there is more

than one value for the given row and column after the transformation. As usual, you can design any custom function you want, just giving its name or using a λ-function.

2.6.9 Ranking Data

Another useful visualization feature is to rank data. For example, we would like to know how each country is ranked by year. To see this, we will use the pandas `rank` function. But first, we need to clean up our previous pivoted table a bit so that it only has real countries with real data. To do this, first we drop the Euro area entries and shorten the Germany name entry, using the `rename` function and then we drop all the rows containing any NaN, using the `dropna` function.

Now we can perform the ranking using the `rank` function. Note here that the parameter `ascending=False` makes the ranking go from the highest values to the lowest values. The Pandas rank function supports different tie-breaking methods, specified with the `method` parameter. In our case, we use the `first` method, in which ranks are assigned in the order they appear in the array, avoiding gaps between ranking.

In [27]:
```
pivedu = pivedu.drop([
                      'Euro area (13 countries)',
                      'Euro area (15 countries)',
                      'Euro area (17 countries)',
                      'Euro area (18 countries)',
                      'European Union (25 countries)',
                      'European Union (27 countries)',
                      'European Union (28 countries)'
                     ],
                     axis = 0)
pivedu = pivedu.rename(index = {'Germany (until 1990 former territory
    of the FRG)': 'Germany'})
pivedu = pivedu.dropna()
pivedu.rank(ascending = False, method = 'first').head()
```

Out[27]:

TIME	2006	2007	2008	2009	2010	2011
GEO						
Austria	10	7	11	7	8	8
Belgium	5	4	3	4	5	5
Bulgaria	21	21	20	20	22	21
Cyprus	2	2	2	2	2	3
Czech Republic	19	20	21	21	20	18

If we want to make a global ranking taking into account all the years, we can sum up all the columns and rank the result. Then we can sort the resulting values to retrieve the top five countries for the last 6 years, in this way:

In [28]:
```
totalSum = pivedu.sum(axis = 1)
totalSum.rank(ascending = False, method = 'dense')
        .sort_values().head()
```

Out[28]:
```
GEO
Denmark                1
Cyprus                 2
Finland                3
Malta                  4
Belgium                5
dtype: float64
```

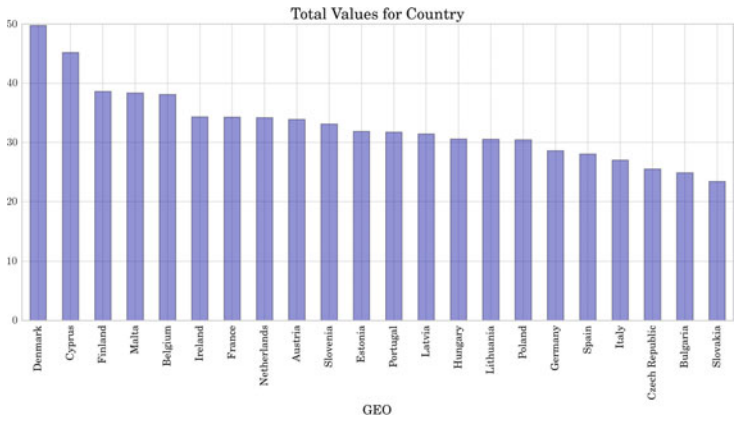
Notice that the `method` keyword argument in the in the `rank` function specifies how items that compare equals receive ranking. In the case of `dense`, items that compare equals receive the same ranking number, and the next not equal item receives the immediately following ranking number.

2.6.10 Plotting

Pandas DataFrames and Series can be plotted using the `plot` function, which uses the library for graphics Matplotlib. For example, if we want to plot the accumulated values for each country over the last 6 years, we can take the Series obtained in the previous example and plot it directly by calling the `plot` function as shown in the next cell:

In [29]:
```
totalSum = pivedu.sum(axis = 1)
                 .sort_values(ascending = False)
totalSum.plot(kind = 'bar', style = 'b', alpha = 0.4,
              title = "Total Values for Country")
```

Out[29]:



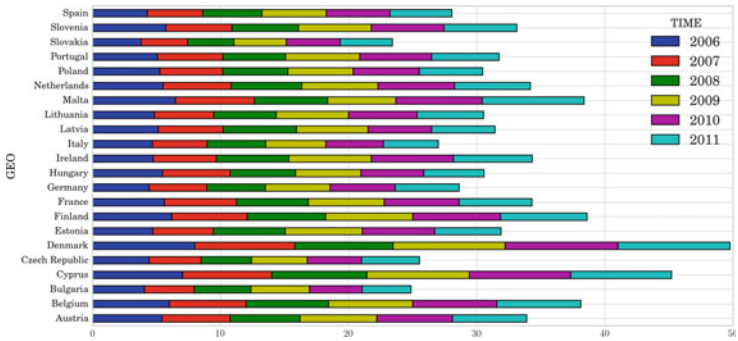
Note that if we want the bars ordered from the highest to the lowest value, we need to sort the values in the Series first. The parameter `kind` used in the `plot` function defines which kind of graphic will be used. In our case, a bar graph. The parameter `style` refers to the style properties of the graphic, in our case, the color

of bars is set to `b` (blue). The alpha channel can be modified adding a keyword parameter `alpha` with a percentage, producing a more translucent plot. Finally, using the `title` keyword the name of the graphic can be set.

It is also possible to plot a DataFrame directly. In this case, each column is treated as a separated Series. For example, instead of printing the accumulated value over the years, we can plot the value for each year.

In [30]:
```
my_colors = ['b', 'r', 'g', 'y', 'm', 'c']
ax = pivedu.plot(kind = 'barh',
                 stacked = True,
                 color = my_colors)
ax.legend(loc = 'center left', bbox_to_anchor = (1, .5))
```

Out[30]:



In this case, we have used a horizontal bar graph (`kind='barh'`) stacking all the years in the same country bar. This can be done by setting the parameter `stacked` to `True`. The number of default colors in a plot is only 5, thus if you have more than 5 Series to show, you need to specify more colors or otherwise the same set of colors will be used again. We can set a new set of colors using the keyword `color` with a list of colors. Basic colors have a single-character code assigned to each, for example, "b" is for blue, "r" for red, "g" for green, "y" for yellow, "m" for magenta, and "c" for cyan. When several Series are shown in a plot, a legend is created for identifying each one. The name for each Series is the name of the column in the DataFrame. By default, the legend goes inside the plot area. If we want to change this, we can use the `legend` function of the axis object (this is the object returned when the plot function is called). By using the `loc` keyword, we can set the relative position of the legend with respect to the plot. It can be a combination of right or left and upper, lower, or center. With `bbox_to_anchor` we can set an absolute position with respect to the plot, allowing us to put the legend outside the graph.

2.7 Conclusions

This chapter has been a brief introduction to the most essential elements of a programming environment for data scientists. The tutorial followed in this chapter is just a starting point for more advanced projects and techniques. As we will see in the following chapters, Python and its ecosystem is a very empowering choice for developing data science projects.

Acknowledgements This chapter was co-written by Eloi Puertas and Francesc Dantí.

Descriptive Statistics

3

3.1 Introduction

Descriptive statistics helps to simplify large amounts of data in a sensible way. In contrast to inferential statistics, which will be introduced in a later chapter, in descriptive statistics we do not draw conclusions beyond the data we are analyzing; neither do we reach any conclusions regarding hypotheses we may make. We do not try to infer characteristics of the "population" (see below) of the data, but claim to present quantitative descriptions of it in a manageable form. It is simply a way to describe the data.

Statistics, and in particular descriptive statistics, is based on two main concepts:

- a *population* is a collection of objects, items ("units") about which information is sought;
- a *sample* is a part of the population that is observed.

Descriptive statistics applies the concepts, measures, and terms that are used to describe the basic features of the samples in a study. These procedures are essential to provide summaries about the samples as an approximation of the population. Together with simple graphics, they form the basis of every quantitative analysis of data. In order to describe the sample data and to be able to infer any conclusion, we should go through several steps:

1. *Data preparation*: Given a specific example, we need to prepare the data for generating statistically valid descriptions.
2. *Descriptive statistics*: This generates different statistics to describe and summarize the data concisely and evaluate different ways to visualize them.

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_3

3.2 Data Preparation

One of the first tasks when analyzing data is to collect and prepare the data in a format appropriate for analysis of the samples. The most common steps for data preparation involve the following operations.

1. *Obtaining the data:* Data can be read directly from a file or they might be obtained by scraping the web.
2. *Parsing the data:* The right parsing procedure depends on what format the data are in: plain text, fixed columns, CSV, XML, HTML, etc.
3. *Cleaning the data:* Survey responses and other data files are almost always incomplete. Sometimes, there are multiple codes for things such as, not asked, did not know, and declined to answer. And there are almost always errors. A simple strategy is to remove or ignore incomplete records.
4. *Building data structures:* Once you read the data, it is necessary to store them in a data structure that lends itself to the analysis we are interested in. If the data fit into the memory, building a data structure is usually the way to go. If not, usually a database is built, which is an out-of-memory data structure. Most databases provide a mapping from keys to values, so they serve as dictionaries.

3.2.1 The Adult Example

Let us consider a public database called the "Adult" dataset, hosted on the UCI's Machine Learning Repository.[1] It contains approximately 32,000 observations concerning different financial parameters related to the US population: age, sex, marital (marital status of the individual), country, income (Boolean variable: whether the person makes more than $50,000 per annum), education (the highest level of education achieved by the individual), occupation, capital gain, etc.

We will show that we can explore the data by asking questions like: "Are men more likely to become high-income professionals than women, i.e., to receive an income of over $50,000 per annum?"

[1] https://archive.ics.uci.edu/ml/datasets/Adult.

First, let us read the data:

In [1]:
```
file = open('files/ch03/adult.data', 'r')
def chr_int(a):
    if a.isdigit(): return int(a)
    else: return 0

data = []
for line in file:
     data1 = line.split(', ')
     if len(data1) == 15:
        data.append([chr_int(data1[0]), data1[1],
                     chr_int(data1[2]), data1[3],
                     chr_int(data1[4]), data1[5],
                     data1[6], data1[7], data1[8],
                     data1[9], chr_int(data1[10]),
                     chr_int(data1[11]),
                     chr_int(data1[12]),
                     data1[13], data1[14]
                    ])
```

Checking the data, we obtain:

In [2]:
```
print data[1:2]
```

Out[2]:
```
[[50, 'Self-emp-not-inc', 83311, 'Bachelors', 13,
'Married-civ-spouse', 'Exec-managerial', 'Husband', 'White',
'Male', 0, 0, 13, 'United-States', '<=50K']]
```

One of the easiest ways to manage data in Python is by using the `DataFrame` structure, defined in the *Pandas* library, which is a two-dimensional, size-mutable, potentially heterogeneous tabular data structure with labeled axes:

In [3]:
```
df = pd.DataFrame(data)
df.columns = [
    'age', 'type_employer', 'fnlwgt',
    'education', 'education_num', 'marital',
    'occupation',' relationship', 'race',
    'sex', 'capital_gain', 'capital_loss',
    'hr_per_week', 'country', 'income'
    ]
```

The command `shape` gives exactly the number of data samples (in rows, in this case) and features (in columns):

In [4]:
```
df.shape
```

Out[4]:
```
(32561, 15)
```

Thus, we can see that our dataset contains 32,561 data records with 15 features each. Let us count the number of items per country:

In [5]:
```
counts = df.groupby('country').size()
print counts.head()
```

Out[5]:
```
country
? 583
Cambodia 19
Vietnam 67
Yugoslavia 16
```

The first row shows the number of samples with unknown country, followed by the number of samples corresponding to the first countries in the dataset.

Let us split people according to their gender into two groups: men and women.

In [6]:
```
ml = df[(df.sex == 'Male')]
```

If we focus on high-income professionals separated by sex, we can do:

In [7]:
```
ml1 = df[(df.sex == 'Male') & (df.income=='>50K\n')
    ]
fm = df[(df.sex == 'Female')]
fm1 = df[(df.sex == 'Female') & (df.income=='>50K\n
    ')]
```

3.3 Exploratory Data Analysis

The data that come from performing a particular measurement on all the subjects in a sample represent our observations for a single characteristic like `country`, `age`, `education`, etc. These measurements and categories represent a *sample distribution* of the variable, which in turn approximately represents the *population distribution* of the variable. One of the main goals of exploratory data analysis is to visualize and summarize the sample distribution, thereby allowing us to make tentative assumptions about the population distribution.

3.3.1 Summarizing the Data

The data in general can be categorical or quantitative. For categorical data, a simple tabulation of the frequency of each category is the best non-graphical exploration for data analysis. For example, we can ask ourselves what is the proportion of high-income professionals in our database:

In [8]:
```
df1 = df[(df.income=='>50K\n')]
print 'The rate of people with high income is: ',
      int(len(df1)/float(len(df))*100), '%.'
print 'The rate of men with high income is: ',
      int(len(ml1)/float(len(ml))*100), '%.'
print 'The rate of women with high income is: ',
      int(len(fm1)/float(len(fm))*100), '%.'
```

Out[8]:
```
The rate of people with high income is: 24 %.
The rate of men with high income is: 30 %.
The rate of women with high income is: 10 %.
```

Given a quantitative variable, exploratory data analysis is a way to make preliminary assessments about the population distribution of the variable using the data of the observed samples. The characteristics of the population distribution of a quantitative variable are its *mean*, *deviation*, *histograms*, *outliers*, etc. Our observed data represent just a finite set of samples of an often infinite number of possible samples. The characteristics of our randomly observed samples are interesting only to the degree that they represent the population of the data they came from.

3.3.1.1 Mean

One of the first measurements we use to have a look at the data is to obtain *sample statistics* from the data, such as the sample mean [1]. Given a sample of n values, $\{x_i\}$, $i = 1, \ldots, n$, the *mean*, μ, is the sum of the values divided by the number of values,[2] in other words:

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i . \tag{3.1}$$

The terms mean and *average* are often used interchangeably. In fact, the main distinction between them is that the mean of a sample is the summary statistic computed by Eq. (3.1), while an average is not strictly defined and could be one of many summary statistics that can be chosen to describe the central tendency of a sample.

In our case, we can consider what the average age of men and women samples in our dataset would be in terms of their mean:

[2] We will use the following notation: X is a random variable, $\mathbf{x}$ is a column vector, $\mathbf{x}^{\mathbf{T}}$ (the transpose of $\mathbf{x}$) is a row vector, $\mathbf{X}$ is a matrix, and x_i is the i-th element of a dataset.

In [9]:
```
print 'The average age of men is: ',
      ml['age'].mean()
print 'The average age of women is: ',
      fm['age'].mean()

print 'The average age of high-income men is: ',
      ml1['age'].mean()
print 'The average age of high-income women is: ',
      fm1['age'].mean()
```

Out[9]:
```
The average age of men is: 39.4335474989
The average age of women is: 36.8582304336
The average age of high-income men is: 44.6257880516
The average age of high-income women is: 42.1255301103
```

This difference in the sample means can be considered initial evidence that there are differences between men and women with high income!

Comment: Later, we will work with both concepts: the population mean and the sample mean. We should not confuse them! The first is the mean of samples taken from the population; the second, the mean of the whole population.

3.3.1.2 Sample Variance

The mean is not usually a sufficient descriptor of the data. We can go further by knowing two numbers: mean and *variance*. The variance σ^2 describes the spread of the data and it is defined as follows:

$$\sigma^2 = \frac{1}{n}\sum_i (x_i - \mu)^2. \tag{3.2}$$

The term $(x_i - \mu)$ is called the *deviation* from the mean, so the variance is the mean squared deviation. The square root of the variance, σ, is called the *standard deviation*. We consider the standard deviation, because the variance is hard to interpret (e.g., if the units are grams, the variance is in grams squared).

Let us compute the mean and the variance of hours per week men and women in our dataset work:

In [10]:
```
ml_mu = ml['age'].mean()
fm_mu = fm['age'].mean()
ml_var = ml['age'].var()
fm_var = fm['age'].var()
ml_std = ml['age'].std()
fm_std = fm['age'].std()
print 'Statistics of age for men: mu:',
      ml_mu, 'var:', ml_var, 'std:', ml_std
print 'Statistics of age for women: mu:',
      fm_mu, 'var:', fm_var, 'std:', fm_std
```

```
Out[10]: Statistics of age for men: mu: 39.4335474989 var: 178.773751745
         std: 13.3706301925
         Statistics of age for women: mu: 36.8582304336 var:
         196.383706395 std: 14.0136970994
```

We can see that the mean number of hours worked per week by women is significantly lesser than that worked by men, but with much higher variance and standard deviation.

3.3.1.3 Sample Median

The mean of the samples is a good descriptor, but it has an important drawback: what will happen if in the sample set there is an error with a value very different from the rest? For example, considering hours worked per week, it would normally be in a range between 20 and 80; but what would happen if by mistake there was a value of 1000? An item of data that is significantly different from the rest of the data is called an *outlier*. In this case, the mean, μ, will be drastically changed towards the outlier. One solution to this drawback is offered by the statistical *median*, μ_{12}, which is an order statistic giving the middle value of a sample. In this case, all the values are ordered by their magnitude and the median is defined as the value that is in the middle of the ordered list. Hence, it is a value that is much more robust in the face of outliers.

Let us see, the median age of working men and women in our dataset and the median age of high-income men and women:

In [11]:
```
ml_median = ml['age'].median()
fm_median = fm['age'].median()
print "Median age per men and women: ",
      ml_median, fm_median

ml_median_age = ml1['age'].median()
fm_median_age = fm1['age'].median()
print "Median age per men and women with high-
    income: ",
      ml_median_age, fm_median_age
```

```
Out[11]: Median age per men and women: 38.0 35.0
         Median age per men and women with high-income: 44.0 41.0
```

As expected, the median age of high-income people is higher than the whole set of working people, although the difference between men and women in both sets is the same.

3.3.1.4 Quantiles and Percentiles

Sometimes we are interested in observing how sample data are distributed in general. In this case, we can order the samples $\{x_i\}$, then find the x_p so that it divides the data into two parts, where:

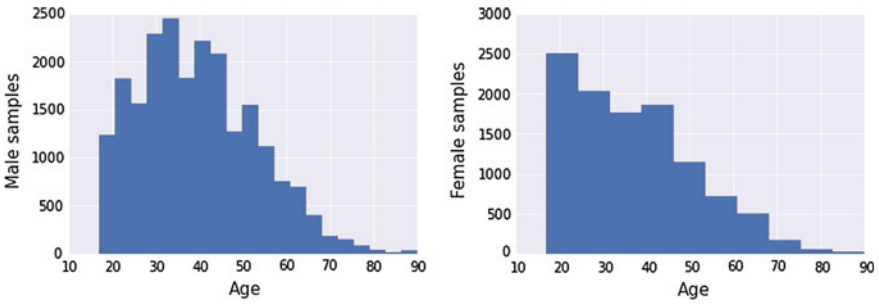


Fig. 3.1 Histogram of the age of working men (*left*) and women (*right*)

- a fraction p of the data values is less than or equal to x_p and
- the remaining fraction $(1 - p)$ is greater than x_p.

That value, x_p, is the p-*th* quantile, or the $100 \times p$-*th* percentile. For example, a 5-number summary is defined by the values x_{min}, Q_1, Q_2, Q_3, x_{max}, where Q_1 is the $25 \times p$-*th* percentile, Q_2 is the $50 \times p$-*th* percentile and Q_3 is the $75 \times p$-*th* percentile.

3.3.2 Data Distributions

Summarizing data by just looking at their mean, median, and variance can be dangerous: very different data can be described by the same statistics. The best thing to do is to validate the data by inspecting them. We can have a look at the data distribution, which describes how often each value appears (i.e., what is its frequency).

The most common representation of a distribution is a *histogram*, which is a graph that shows the frequency of each value. Let us show the age of working men and women separately.

In [12]:
```
ml_age = ml['age']
ml_age.hist(normed = 0, histtype = 'stepfilled',
    bins = 20)
```

In [13]:
```
fm_age = fm['age']
fm_age.hist(normed = 0, histtype = 'stepfilled',
    bins = 10)
```

The output can be seen in Fig. 3.1. If we want to compare the histograms, we can plot them overlapping in the same graphic as follows:

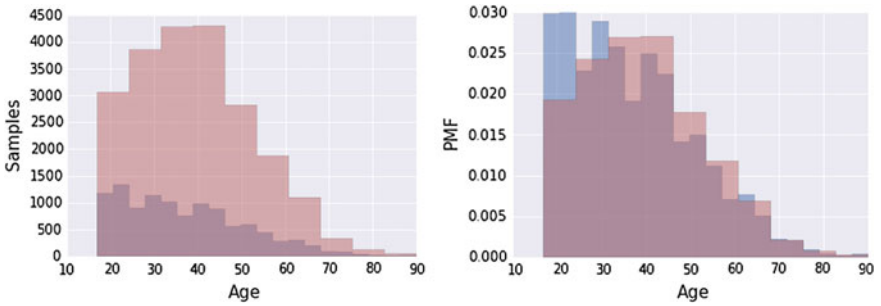


Fig. 3.2 Histogram of the age of working men (in *ochre*) and women (in *violet*) (*left*). Histogram of the age of working men (in *ochre*), women (in *blue*), and their intersection (in *violet*) after samples normalization (*right*)

In [14]:
```
import seaborn as sns
fm_age.hist(normed = 0, histtype = 'stepfilled',
            alpha = .5, bins = 20)
ml_age.hist(normed = 0, histtype = 'stepfilled',
            alpha = .5,
            color = sns.desaturate("indianred",
                .75),
            bins = 10)
```

The output can be seen in Fig. 3.2 (left). Note that we are visualizing the absolute values of the number of people in our dataset according to their age (the abscissa of the histogram). As a side effect, we can see that there are many more men in these conditions than women.

We can normalize the frequencies of the histogram by dividing/normalizing by n, the number of samples. The normalized histogram is called the *Probability Mass Function* (PMF).

In [15]:
```
fm_age.hist(normed = 1, histtype = 'stepfilled',
            alpha = .5, bins = 20)
ml_age.hist(normed = 1, histtype = 'stepfilled',
            alpha = .5, bins = 10,
            color = sns.desaturate("indianred",
                .75))
```

This outputs Fig. 3.2 (right), where we can observe a comparable range of individuals (men and women).

The *Cumulative Distribution Function* (CDF), or just distribution function, describes the probability that a real-valued random variable X with a given probability distribution will be found to have a value less than or equal to x. Let us show the CDF of age distribution for both men and women.

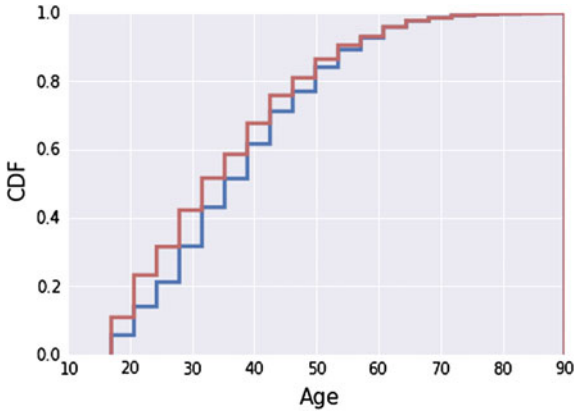


Fig. 3.3 The CDF of the age of working male (in *blue*) and female (in *red*) samples

In [16]:

```
ml_age.hist(normed = 1, histtype = 'step',
            cumulative = True,  linewidth = 3.5,
            bins = 20)
fm_age.hist(normed = 1, histtype='step',
            cumulative = True,  linewidth = 3.5,
            bins = 20,
            color = sns.desaturate("indianred",
                .75))
```

The output can be seen in Fig. 3.3, which illustrates the CDF of the age distributions for both men and women.

3.3.3 Outlier Treatment

As mentioned before, outliers are data samples with a value that is far from the central tendency. Different rules can be defined to detect outliers, as follows:

- Computing samples that are far from the median.
- Computing samples whose values exceed the mean by 2 or 3 standard deviations.

For example, in our case, we are interested in the age statistics of men versus women with high incomes and we can see that in our dataset, the minimum age is 17 years and the maximum is 90 years. We can consider that some of these samples are due to errors or are not representable. Applying the domain knowledge, we focus on the median age (37, in our case) up to 72 and down to 22 years old, and we consider the rest as outliers.

In [17]:
```
df2 = df.drop(df.index[
    (df.income == '>50K\n') &
    (df['age'] > df['age'].median() + 35) &
    (df['age'] > df['age'].median() -15)
    ])
ml1_age = ml1['age']
fm1_age = fm1['age']

ml2_age = ml1_age.drop(ml1_age.index[
    (ml1_age > df['age'].median() + 35) &
    (ml1_age > df['age'].median() - 15)
    ])
fm2_age = fm1_age.drop(fm1_age.index[
    (fm1_age > df['age'].median() + 35) &
    (fm1_age > df['age'].median() - 15)
    ])
```

We can check how the mean and the median changed once the data were cleaned:

In [18]:
```
mu2ml = ml2_age.mean()
std2ml = ml2_age.std()
md2ml = ml2_age.median()
mu2fm = fm2_age.mean()
std2fm = fm2_age.std()
md2fm = fm2_age.median()

print "Men statistics:"
print "Mean:", mu2ml, "Std:", std2ml
print "Median:", md2ml
print "Min:", ml2_age.min(), "Max:", ml2_age.max()

print "Women statistics:"
print "Mean:", mu2fm, "Std:", std2fm
print "Median:", md2fm
print "Min:", fm2_age.min(), "Max:", fm2_age.max()
```

Out[18]:
```
Men statistics: Mean: 44.3179821239 Std: 10.0197498572 Median:
44.0 Min: 19 Max: 72
Women statistics: Mean: 41.877028181 Std: 10.0364418073 Median:
41.0 Min: 19 Max: 72
```

Let us visualize how many outliers are removed from the whole data by:

In [19]:
```
plt.figure(figsize = (13.4, 5))
df.age[(df.income == '>50K\n')]
      .plot(alpha = .25, color = 'blue')
df2.age[(df2.income == '>50K\n')]
      .plot(alpha = .45, color = 'red')
```

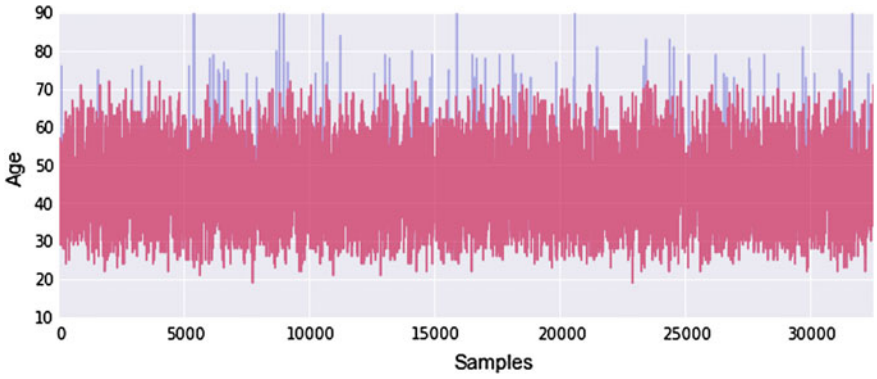


Fig. 3.4 The *red* shows the cleaned data without the considered outliers (in *blue*)

Figure 3.4 shows the outliers in blue and the rest of the data in red. Visually, we can confirm that we removed mainly outliers from the dataset.

Next we can see that by removing the outliers, the difference between the populations (men and women) actually decreased. In our case, there were more outliers in men than women. If the difference in the mean values before removing the outliers is 2.5, after removing them it slightly decreased to 2.44:

In [20]:
```
print 'The mean difference with outliers is: %4.2f.
    '
       % (ml_age.mean() - fm_age.mean())
print 'The mean difference without outliers is:
    %4.2f.'
       % (ml2_age.mean() - fm2_age.mean())
```

Out[20]:
```
The mean difference with outliers is: 2.58.
The mean difference without outliers is: 2.44.
```

Let us observe the difference of men and women incomes in the cleaned subset with some more details.

In [21]:
```
countx, divisionx = np.histogram(ml2_age, normed =
    True)
county, divisiony = np.histogram(fm2_age, normed =
    True)

val = [(divisionx[i] + divisionx[i+1])/2
       for i in range(len(divisionx) - 1)]
plt.plot(val, countx - county, 'o-')
```

The results are shown in Fig. 3.5. One can see that the differences between male and female values are slightly negative before age 42 and positive after it. Hence, women tend to be promoted (receive more than 50 K) earlier than men.

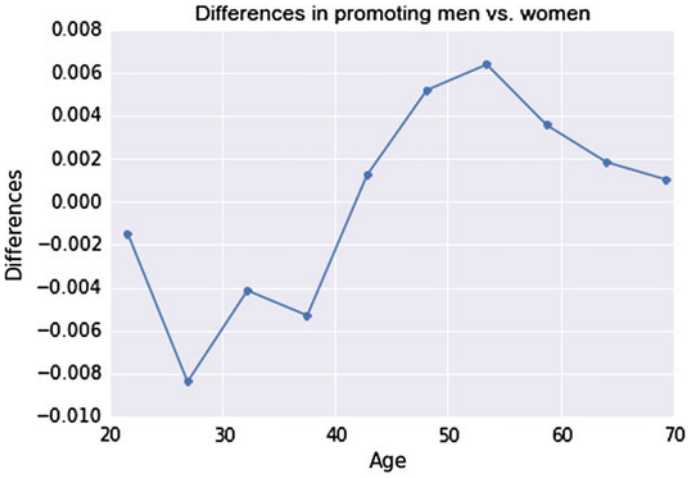


Fig. 3.5 Differences in high-income earner men versus women as a function of age

3.3.4 Measuring Asymmetry: Skewness and Pearson's Median Skewness Coefficient

For univariate data, the formula for *skewness* is a statistic that measures the asymmetry of the set of n data samples, x_i:

$$g_1 = \frac{1}{n} \frac{\sum_i (x_i - \mu^3)}{\sigma^3}, \tag{3.3}$$

where μ is the mean, σ is the standard deviation, and n is the number of data points.

Negative deviation indicates that the distribution "skews left" (it extends further to the left than to the right). One can easily see that the skewness for a normal distribution is zero, and any symmetric data must have a skewness of zero. Note that skewness can be affected by outliers! A simpler alternative is to look at the relationship between the mean μ and the median μ_{12}.

In [22]:
```
def skewness(x):
    res = 0
    m = x.mean()
    s = x.std()
    for i in x:
        res += (i-m) * (i-m) * (i-m)
    res /= (len(x) * s * s * s)
    return res

print "Skewness of the male population = ",
      skewness(ml2_age)
print "Skewness of the female population is = ",
      skewness(fm2_age)
```

Out[22]:
```
Skewness of the male population = 0.266444383843
Skewness of the female population = 0.386333524913
```

That is, the female population is more skewed than the male, probably since men could be most prone to retire later than women.

The **Pearson's median skewness coefficient** is a more robust alternative to the skewness coefficient and is defined as follows:

$$g_p = 3(\mu - \mu_{12})\sigma.$$

There are many other definitions for skewness that will not be discussed here. In our case, if we check the Pearson's skewness coefficient for both men and women, we can see that the difference between them actually increases:

In [23]:
```
def pearson(x):
    return 3*(x.mean() - x.median())*x.std()

print "Pearson's coefficient of the male population
    = ",
      pearson(ml2_age)
print "Pearson's coefficient of the female
   population = ",
      pearson(fm2_age)
```

Out[23]:
```
Pearson's coefficient of the male population = 9.55830402221
Pearson's coefficient of the female population = 26.4067269073
```

3.3.4.1 Discussions

After exploring the data, we obtained some apparent effects that seem to support our initial assumptions. For example, the mean age for men in our dataset is 39.4 years; while for women, is 36.8 years. When analyzing the high-income salaries, the mean age for men increased to 44.6 years; while for women, increased to 42.1 years. When the data were cleaned from outliers, we obtained mean age for high-income men: 44.3, and for women: 41.8. Moreover, histograms and other statistics show the skewness of the data and the fact that women used to be promoted a little bit earlier than men, in general.

3.3.5 Continuous Distribution

The distributions we have considered up to now are based on empirical observations and thus are called *empirical distributions*. As an alternative, we may be interested in considering distributions that are defined by a continuous function and are called *continuous distributions* [2]. Remember that we defined the PMF, $f_X(x)$, of a discrete random variable X as $f_X(x) = P(X = x)$ for all x. In the case of a continuous random variable X, we speak of the *Probability Density Function* (PDF), which

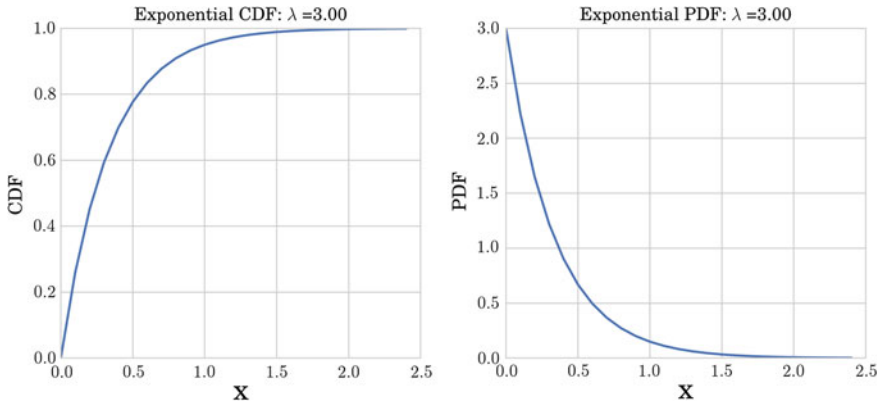


Fig. 3.6 Exponential CDF (*left*) and PDF (*right*) with $\lambda = 3.00$

is defined as $F_X(x)$ where this satisfies: $F_X(x) = \int_{\infty}^{x} f_X(t)\delta t$ for all x. There are many continuous distributions; here, we will consider the most common ones: the exponential and the normal distributions.

3.3.5.1 The Exponential Distribution

Exponential distributions are well known since they describe the inter-arrival time between events. When the events are equally likely to occur at any time, the distribution of the inter-arrival time tends to an exponential distribution. The CDF and the PDF of the exponential distribution are defined by the following equations:

$$CDF(x) = 1 - e^{-\lambda x}, \quad PDF(x) = \lambda e^{-\lambda x}.$$

The parameter λ defines the shape of the distribution. An example is given in Fig. 3.6. It is easy to show that the mean of the distribution is $\frac{1}{\lambda}$, the variance is $\frac{1}{\lambda^2}$ and the median is $\frac{\ln(2)}{\lambda}$.

Note that for a small number of samples, it is difficult to see that the exact empirical distribution fits a continuous distribution. The best way to observe this match is to generate samples from the continuous distribution and see if these samples match the data. As an exercise, you can consider the birthdays of a large enough group of people, sorting them and computing the inter-arrival time in days. If you plot the CDF of the inter-arrival times, you will observe the exponential distribution.

There are a lot of real-world events that can be described with this distribution, including the time until a radioactive particle decays; the time it takes before your next telephone call; and the time until default (on payment to company debt holders) in reduced-form credit risk modeling. The random variable X of the lifetime of some batteries is associated with a probability density function of the form: $PDF(x) = \frac{1}{4}e^{-\frac{x}{4}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$.

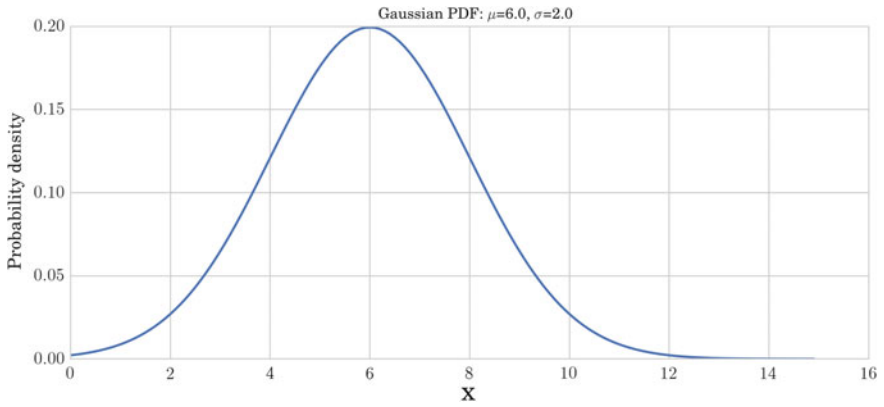


Fig. 3.7 Normal PDF with $\mu = 6$ and $\sigma = 2$

3.3.5.2 The Normal Distribution

The *normal distribution*, also called the *Gaussian distribution*, is the most common since it represents many real phenomena: economic, natural, social, and others. Some well-known examples of real phenomena with a normal distribution are as follows:

- The size of living tissue (length, height, weight).
- The length of inert appendages (hair, nails, teeth) of biological specimens.
- Different physiological measurements (e.g., blood pressure), etc.

The normal CDF has no closed-form expression and its most common representation is the PDF:

$$\mathrm{PD}F(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

The parameter σ defines the shape of the distribution. An example of the PDF of a normal distribution with $\mu = 6$ and $\sigma = 2$ is given in Fig. 3.7.

3.3.6 Kernel Density

In many real problems, we may not be interested in the parameters of a particular distribution of data, but just a continuous representation of the data. In this case, we should estimate the distribution non-parametrically (i.e., making no assumptions about the form of the underlying distribution) using kernel density estimation. Let us imagine that we have a set of data measurements without knowing their distribution and we need to estimate the continuous representation of their distribution. In this case, we can consider a Gaussian kernel to generate the density around the data. Let us consider a set of random data generated by a bimodal normal distribution. If we consider a Gaussian kernel around the data, the sum of those kernels can give us

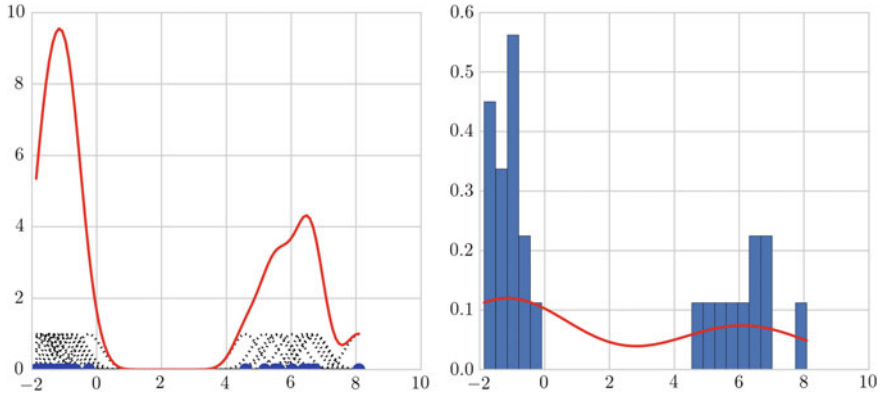


Fig. 3.8 Summed kernel functions around a random set of points (*left*) and the kernel density estimate with the optimal bandwidth (*right*) for our dataset. Random data shown in *blue*, kernel shown in *black* and summed function shown in *red*

a continuous function that when normalized would approximate the density of the distribution:

In [24]:
```
x1 = np.random.normal(-1, 0.5, 15)
x2 = np.random.normal(6, 1, 10)
y = np.r_[x1, x2] # r_ translates slice objects to
    concatenation along the first axis.
x = np.linspace(min(y), max(y), 100)

s = 0.4 # Smoothing parameter

# Calculate the kernels
kernels = np.transpose([norm.pdf(x, yi, s) for yi
    in y])
plt.plot(x, kernels, 'k:')
plt.plot(x, kernels.sum(1), 'r')
plt.plot(y, np.zeros(len(y)), 'bo', ms = 10)
```

Figure 3.8 (left) shows the result of the construction of the continuous function from the kernel summarization.

In fact, the library SciPy[3] implements a Gaussian kernel density estimation that automatically chooses the appropriate bandwidth parameter for the kernel. Thus, the final construction of the density estimate will be obtained by:

[3] http://www.scipy.org.

In [25]:
```
from scipy.stats import kde
density = kde.gaussian_kde(y)
xgrid = np.linspace(x.min(), x.max(), 200)
plt.hist(y, bins = 28, normed = True)
plt.plot(xgrid, density(xgrid), 'r-')
```

Figure 3.8 (right) shows the result of the kernel density estimate for our example.

3.4 Estimation

An important aspect when working with statistical data is being able to use estimates to approximate the values of unknown parameters of the dataset. In this section, we will review different kinds of estimators (estimated mean, variance, standard score, etc.).

3.4.1 Sample and Estimated Mean, Variance and Standard Scores

In continuation, we will deal with point estimators that are single numerical estimates of parameters of a population.

3.4.1.1 Mean

Let us assume that we know that our data are coming from a normal distribution and the random samples drawn are as follows:

$$\{0.33, -1.76, 2.34, 0.56, 0.89\}.$$

The question is can we guess the mean μ of the distribution? One approximation is given by the sample mean, $\bar{x}$. This process is called *estimation* and the statistic (e.g., the sample mean) is called an *estimator*. In our case, the sample mean is 0.472, and it seems a logical choice to represent the mean of the distribution. It is not so evident if we add a sample with a value of -465. In this case, the sample mean will be -77.11, which does not look like the mean of the distribution. The reason is due to the fact that the last value seems to be an outlier compared to the rest of the sample. In order to avoid this effect, we can try first to remove outliers and then to estimate the mean; or we can use the sample median as an estimator of the mean of the distribution. If there are no outliers, the sample mean $\bar{x}$ minimizes the following *mean squared error*:

$$MSE = \frac{1}{n}\sum(\bar{x} - \mu)^2,$$

where n is the number of times we estimate the mean.

Let us compute the MSE of a set of random data:

In [26]:
```
NTs = 200
mu = 0.0
var = 1.0
err = 0.0
NPs = 1000
for i in range(NTs):
    x = np.random.normal(mu, var, NPs)
    err += (x.mean()-mu)**2
print 'MSE: ', err/NTests
```

Out[26]:
```
MSE: 0.00019879541147
```

3.4.1.2 Variance

If we ask ourselves what is the variance, σ^2, of the distribution of X, analogously we can use the sample variance as an estimator. Let us denote by $\bar{\sigma}^2$ the sample variance estimator:

$$\bar{\sigma}^2 = \frac{1}{n}\sum(x_i - \bar{x})^2.$$

For large samples, this estimator works well, but for a small number of samples it is biased. In those cases, a better estimator is given by:

$$\bar{\sigma}^2 = \frac{1}{n-1}\sum(x_i - \bar{x})^2.$$

3.4.1.3 Standard Score

In many real problems, when we want to compare data, or estimate their correlations or some other kind of relations, we must avoid data that come in different units. For example, weight can come in kilograms or grams. Even data that come in the same units can still belong to different distributions. We need to normalize them to standard scores. Given a dataset as a series of values, $\{x_i\}$, we convert the data to standard scores by subtracting the mean and dividing them by the standard deviation:

$$z_i = \frac{(x_i - \mu)}{\sigma}.$$

Note that this measure is dimensionless and its distribution has a mean of 0 and variance of 1. It inherits the "shape" of the dataset: if X is normally distributed, so is Z; if X is skewed, so is Z.

3.4.2 Covariance, and Pearson's and Spearman's Rank Correlation

Variables of data can express relations. For example, countries that tend to invest in research also tend to invest more in education and health. This kind of relationship is captured by the covariance.

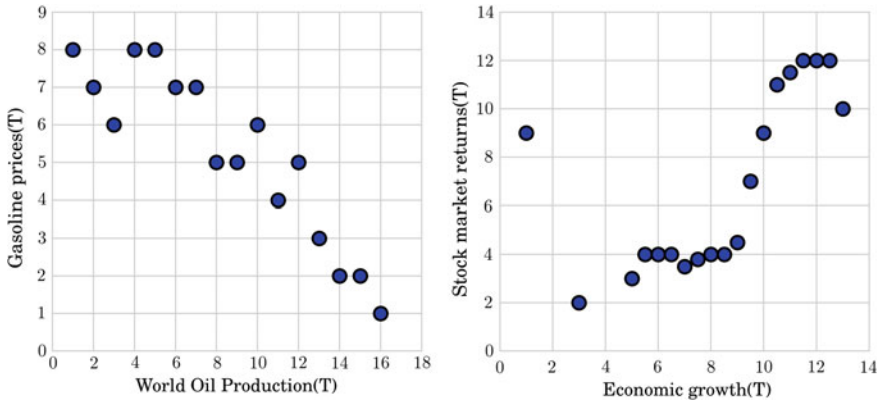


Fig. 3.9 Positive correlation between economic growth and stock market returns worldwide (*left*). Negative correlation between the world oil production and gasoline prices worldwide (*right*)

3.4.2.1 Covariance

When two variables share the same tendency, we speak about *covariance*. Let us consider two series, $\{x_i\}$ and $\{y_i\}$. Let us center the data with respect to their mean: $dx_i = x_i - \mu_X$ and $dy_i = y_i - \mu_Y$. It is easy to show that when $\{x_i\}$ and $\{y_i\}$ vary together, their deviations tend to have the same sign. The covariance is defined as the mean of the following products:

$$Cov(X, Y) = \frac{1}{n} \sum_{i=1}^{n} dx_i dy_i,$$

where n is the length of both sets. Still, the covariance itself is hard to interpret.

3.4.2.2 Correlation and the Pearson's Correlation

If we normalize the data with respect to their deviation, that leads to the standard scores; and then multiplying them, we get:

$$\rho_i = \frac{x_i - \mu_X}{\sigma_X} \frac{y_i - \mu_Y}{\sigma_Y}.$$

The mean of this product is $\rho = \frac{1}{n} \sum_{i=1}^{n} \rho_i$. Equivalently, we can rewrite ρ in terms of the covariance, and thus obtain the *Pearson's correlation*:

$$\rho = \frac{Cov(X, Y)}{\sigma_X \sigma_Y}.$$

Note that the Pearson's correlation is always between -1 and $+1$, where the magnitude depends on the degree of correlation. If the Pearson's correlation is 1 (or -1), it means that the variables are perfectly correlated (positively or negatively) (see Fig. 3.9). This means that one variable can predict the other very well. However,

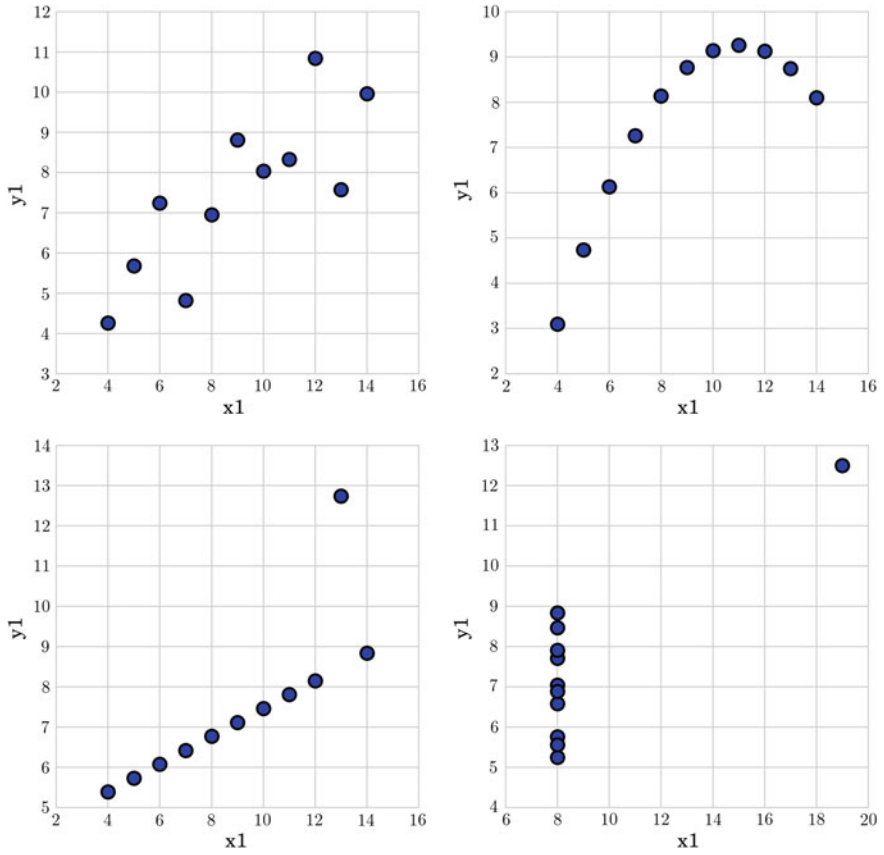


Fig. 3.10 Anscombe configurations

having $\rho = 0$, does not necessarily mean that the variables are not correlated! Pearson's correlation captures correlations of first order, but not nonlinear correlations. Moreover, it does not work well in the presence of outliers.

3.4.2.3 Spearman's Rank Correlation

The *Spearman's rank correlation* comes as a solution to the robustness problem of Pearson's correlation when the data contain outliers. The main idea is to use the ranks of the sorted sample data, instead of the values themselves. For example, in the list [4, 3, 7, 5], the rank of 4 is 2, since it will appear second in the ordered list ([3, 4, 5, 7]). Spearman's correlation computes the correlation between the ranks of the data. For example, considering the data: X = [10, 20, 30, 40, 1000], and Y = [−70, −1000, −50, −10, −20], where we have an outlier in each one set. If we compute the ranks, they are [1.0, 2.0, 3.0, 4.0, 5.0] and [2.0, 1.0, 3.0, 5.0, 4.0]. As value of the Pearson's coefficient, we get 0.28, which does not show much correlation

between the sets. However, the Spearman's rank coefficient, capturing the correlation between the ranks, gives as a final value of 0.80, confirming the correlation between the sets. As an exercise, you can compute the Pearson's and the Spearman's rank correlations for the different Anscombe configurations given in Fig. 3.10. Observe if linear and nonlinear correlations can be captured by the Pearson's and the Spearman's rank correlations.

3.5 Conclusions

In this chapter, we have familiarized ourselves with the basic concepts and procedures of descriptive statistics to explore a dataset. As we have seen, it helps us to understand the experiment or a dataset in detail and allows us to put the data in perspective. We introduced the central measures of tendency such as the sample mean and median; and measures of variability such as the variance and standard deviation. We have also discussed how these measures can be affected by outliers. In order to go deeper into visualizing the dataset, we have introduced histograms, quantiles, and percentiles.

In many situations, when the values are continuous variables, it is convenient to use continuous distributions; the most common of which are the normal and the exponential distributions. The advantage of most continuous distributions is that we can have an explicit expression for their PDF and CDF, as well as the mean and variance in terms of a closed formula. Also, we learned how, by using the kernel density, we can obtain a continuous representation of the sample distribution. Finally, we discussed how to estimate the correlation and the covariance of datasets, where two of the most popular measures are the Pearson's and the Spearman's rank correlations, which are affected in different ways by the outliers of the dataset.

Acknowledgements This chapter was co-written by Petia Radeva and Laura Igual.

References

1. A. B. Downey, "Probability and Statistics for Programmers", O'Reilly Media, 2011, ISBN-10: 1449307116.
2. Probability Distributions: Discrete vs. Continuous, http://stattrek.com/probability-distributions/discrete-continuous.aspx.

Statistical Inference

4

4.1 Introduction

There is not only one way to address the problem of statistical inference. In fact, there are two main approaches to statistical inference: the frequentist and Bayesian approaches. Their differences are subtle but fundamental:

- In the case of the *frequentist approach*, the main assumption is that there is a population, which can be represented by several parameters, from which we can obtain numerous random samples. Population parameters are fixed but they are not accessible to the observer. The only way to derive information about these parameters is to take a sample of the population, to compute the parameters of the sample, and to use statistical inference techniques to make probable propositions regarding population parameters.
- The *Bayesian approach* is based on a consideration that data are fixed, not the result of a repeatable sampling process, but parameters describing data can be described probabilistically. To this end, Bayesian inference methods focus on producing parameter distributions that represent all the knowledge we can extract from the sample and from prior information about the problem.

A deep understanding of the differences between these approaches is far beyond the scope of this chapter, but there are many interesting references that will enable you to learn about it [1]. What is really important is to realize that the approaches are based on different assumptions which determine the validity of their inferences. The assumptions are related in the first case to a sampling process; and to a statistical model in the second case. Correct inference requires these assumptions to be correct. The fulfillment of this requirement is not part of the method, but it is the responsibility of the data scientist.

In this chapter, to keep things simple, we will only deal with the first approach, but we suggest the reader also explores the second approach as it is well worth it!

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_4

4.2 Statistical Inference: The Frequentist Approach

As we have said, the ultimate objective of statistical inference, if we adopt the frequentist approach, is to produce probable propositions concerning population parameters from analysis of a sample. The most important classes of propositions are as follows:

- Propositions about *point estimates*. A point estimate is a particular value that best approximates some parameter of interest. For example, the mean or the variance of the sample.
- Propositions about *confidence intervals* or *set estimates*. A confidence interval is a range of values that best represents some parameter of interest.
- Propositions about the acceptance or rejection of a *hypothesis*.

In all these cases, the production of propositions is based on a simple assumption: we can estimate the probability that the result represented by the proposition has been caused by chance. The estimation of this probability by sound methods is one of the main topics of statistics.

The development of traditional statistics was limited by the scarcity of computational resources. In fact, the only computational resources were mechanical devices and human computers, teams of people devoted to undertaking long and tedious calculations. Given these conditions, the main results of classical statistics are theoretical approximations, based on idealized models and assumptions, to measure the effect of chance on the statistic of interest. Thus, concepts such as the *Central Limit Theorem*, the *empirical sample distribution* or the *t-test* are central to understanding this approach.

The development of modern computers has opened an alternative strategy for measuring chance that is based on simulation; producing computationally intensive methods including resampling methods (such as bootstrapping), Markov chain Monte Carlo methods, etc. The most interesting characteristic of these methods is that they allow us to treat more realistic models.

4.3 Measuring the Variability in Estimates

Estimates produced by descriptive statistics are not equal to the *truth* but they are better as more data become available. So, it makes sense to use them as central elements of our propositions and to measure its variability with respect to the sample size.

4.3.1 Point Estimates

Let us consider a dataset of accidents in Barcelona in 2013. This dataset can be downloaded from the OpenDataBCN website,[1] Barcelona City Hall's open data service. Each register in the dataset represents an accident via a series of features: weekday, hour, address, number of dead and injured people, etc. This dataset will represent our population: the set of all reported traffic accidents in Barcelona during 2013.

4.3.1.1 Sampling Distribution of Point Estimates

Let us suppose that we are interested in describing the daily number of traffic accidents in the streets of Barcelona in 2013. If we have access to the *population*, the computation of this parameter is a simple operation: the total number of accidents divided by 365.

In [1]:
```
data = pd.read_csv("files/ch04/ACCIDENTS_GU_BCN_2013.csv")
data['Date'] = data[u'Dia de mes'].apply(lambda x: str(x))
               + '-' +
               data[u'Mes de any'].apply(lambda x: str(x))
data['Date'] = pd.to_datetime(data['Date'])
accidents = data.groupby(['Date']).size()
print accidents.mean()
```

Out[1]:
```
Mean: 25.9095
```

But now, for illustrative purposes, let us suppose that we only have access to a limited part of the data (the *sample*): the number of accidents during *some* days of 2013. Can we still give an approximation of the population mean?

The most intuitive way to go about providing such a mean is simply to take the *sample mean*. The sample mean is a point estimate of the population mean. If we can only choose one value to estimate the population mean, then this is our best guess.

The problem we face is that estimates generally vary from one sample to another, and this sampling variation suggests our estimate may be close, but it will not be exactly equal to our parameter of interest. How can we measure this variability?

In our example, because we have access to the population, we can empirically build the *sampling distribution of the sample mean*[2] for a given number of observations. Then, we can use the sampling distribution to compute a measure of the variability.

In Fig. 4.1, we can see the empirical sample distribution of the mean for $s = 10.000$ samples with $n = 200$ observations from our dataset. This empirical distribution has been built in the following way:

[1] http://opendata.bcn.cat/.

[2] Suppose that we draw all possible samples of a given size from a given population. Suppose further that we compute the mean for each sample. The probability distribution of this statistic is called the *mean sampling distribution*.

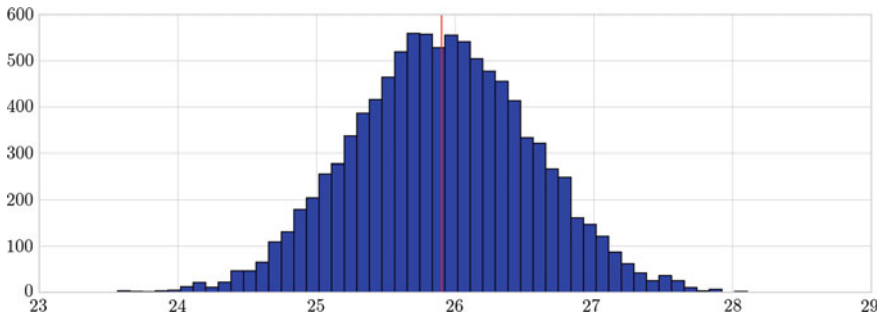


Fig. 4.1 Empirical distribution of the sample mean. In *red*, the mean value of this distribution

1. Draw s (a large number) independent samples $\{\mathbf{x}^1, \ldots, \mathbf{x}^s\}$ from the population where each element $\mathbf{x}^j$ is composed of $\{x_i^j\}_{i=1,\ldots,n}$.
2. Evaluate the sample mean $\hat{\mu}^j = \frac{1}{n}\sum_{i=1}^{n} x_i^j$ of each sample.
3. Estimate the sampling distribution of $\hat{\mu}$ by the empirical distribution of the sample replications.

In [2]:

```
# population
df = accidents.to_frame()
N_test = 10000
elements = 200
# mean array of samples
means = [0] * N_test
# sample generation
for i in range(N_test):
    rows = np.random.choice(df.index.values, elements)
    sampled_df = df.ix[rows]
    means[i] = sampled_df.mean()
```

In general, given a point estimate from a sample of size n, we define its *sampling distribution* as the distribution of the point estimate based on samples of size n from its population. This definition is valid for point estimates of other population parameters, such as the population median or population standard deviation, but we will focus on the analysis of the sample mean.

The sampling distribution of an estimate plays an important role in understanding the real meaning of propositions concerning point estimates. It is very useful to think of a particular point estimate as being drawn from such a distribution.

4.3.1.2 The Traditional Approach

In real problems, we do not have access to the real population and so estimation of the sampling distribution of the estimate from the empirical distribution of the sample replications is not an option. But this problem can be solved by making use of some theoretical results from traditional statistics.

It can be mathematically shown that given n independent observations $\{x_i\}_{i=1,..,n}$ of a population with a standard deviation σ_x, the standard deviation of the sample mean $\sigma_{\bar{x}}$, or *standard error*, can be approximated by this formula:

$$SE = \frac{\sigma_x}{\sqrt{n}}$$

The demonstration of this result is based on the Central Limit Theorem: an old theorem with a history that starts in 1810 when Laplace released his first paper on it.

This formula uses the standard deviation of the population σ_x, which is not known, but it can be shown that if it is substituted by its empirical estimate $\hat{\sigma}_x$, the estimation is sufficiently good if $n > 30$ and the population distribution is not skewed. This allows us to estimate the standard error of the sample mean even if we do not have access to the population.

So, how can we give a measure of the variability of the sample mean? The answer is simple: by giving the **empirical standard error of the mean distribution**.

In [3]:
```
rows = np.random.choice(df.index.values, 200)
sampled_df = df.ix[rows]
est_sigma_mean = sampled_df.std()/math.sqrt(200)

print 'Direct estimation of SE from one sample of
      200 elements:', est_sigma_mean[0]
print 'Estimation of the SE by simulating 10000 samples of
      200 elements:', np.array(means).std()
```

Out[3]:
```
Direct estimation of SE from one sample of 200 elements: 0.6536
Estimation of the SE by simulating 10000 samples of 200
elements: 0.6362
```

Unlike the case of the sample mean, there is no formula for the standard error of other interesting sample estimates, such as the median.

4.3.1.3 The Computationally Intensive Approach

Let us consider from now that our full dataset is a sample from a hypothetical population (this is the most common situation when analyzing real data!).

A modern alternative to the traditional approach to statistical inference is the bootstrapping method [2]. In the bootstrap, we draw n observations *with replacement* from the original data to create a bootstrap sample or resample. Then, we can calculate the mean for this resample. By repeating this process a large number of times, we can build a good approximation of the mean sampling distribution (see Fig. 4.2).

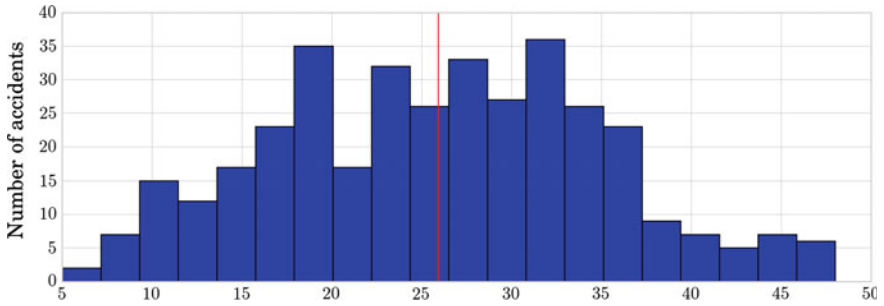


Fig. 4.2 Mean sampling distribution by bootstrapping. In *red*, the mean value of this distribution

In [4]:
```
def meanBootstrap(X, numberb):
    x = [0]*numberb
    for i in range(numberb):
        sample = [X[j]
                  for j
                  in np.random.randint(len(X), size=len(X))
                 ]
        x[i] = np.mean(sample)
    return x
m = meanBootstrap(accidents, 10000)
print "Mean estimate:", np.mean(m)
```

Out[4]:
```
Mean estimate: 25.9094
```

The basic idea of the bootstrapping method is that the observed sample contains sufficient information about the underlying distribution. So, the information we can extract from resampling the sample is a good approximation of what can be expected from resampling the population.

The bootstrapping method can be applied to other simple estimates such as the median or the variance and also to more complex operations such as estimates of censored data.[3]

4.3.2 Confidence Intervals

A point estimate Θ, such as the sample mean, provides a *single plausible value for a parameter*. However, as we have seen, a point estimate is rarely perfect; usually there is some error in the estimate. That is why we have suggested using the standard error as a measure of its variability.

Instead of that, a next logical step would be to provide *a plausible range of values* for the parameter. A plausible range of values for the sample parameter is called a *confidence interval*.

[3]Censoring is a condition in which the value of observation is only partially known.

We will base the definition of *confidence interval* on two ideas:

1. Our point estimate is the most plausible value of the parameter, so it makes sense to build the confidence interval around the point estimate.
2. The *plausibility* of a range of values can be defined from the sampling distribution of the estimate.

For the case of the mean, the Central Limit Theorem states that its sampling distribution is normal:

Theorem 4.1 *Given a population with a finite mean μ and a finite non-zero variance σ^2, the sampling distribution of the mean approaches a normal distribution with a mean of μ and a variance of σ^2/n as n, the sample size, increases.*

In this case, and in order to define an interval, we can make use of a well-known result from probability that applies to normal distributions: roughly 95% of the time our estimate will be within 1.96 standard errors of the true mean of the distribution. If the interval spreads out 1.96 standard errors from a normally distributed point estimate, intuitively we can say that we are *roughly 95% confident that we have captured the true parameter*.

$$CI = [\Theta - 1.96 \times SE, \Theta + 1.96 \times SE]$$

In [5]:
```
m = accidents.mean()
se = accidents.std()/math.sqrt(len(accidents))
ci = [m - se*1.96, m + se*1.96]
print "Confidence interval:", ci
```

Out[5]:
```
Confidence interval: [24.975, 26.8440]
```

Suppose we want to consider confidence intervals where the confidence level is somewhat higher than 95%: perhaps we would like a confidence level of 99%. To create a 99% confidence interval, change 1.96 in the 95% confidence interval formula to be 2.58 (it can be shown that 99% of the time a normal random variable will be within 2.58 standard deviations of the mean).

In general, if the point estimate follows the normal model with standard error SE, then a confidence interval for the population parameter is

$$\Theta \pm z \times SE$$

where z corresponds to the confidence level selected:

Confidence Level	90%	95%	99%	99.9%
z Value	1.65	1.96	2.58	3.291

This is how we would compute a 95% confidence interval of the sample mean using bootstrapping:

1. Repeat the following steps for a large number, s, of times:

   a. Draw n observations with replacement from the original data to create a bootstrap sample or resample.
   b. Calculate the mean for the resample.

2. Calculate the mean of your s values of the sample statistic. This process gives you a "bootstrapped" estimate of the sample statistic.
3. Calculate the standard deviation of your s values of the sample statistic. This process gives you a "bootstrapped" estimate of the SE of the sample statistic.
4. Obtain the 2.5th and 97.5th percentiles of your s values of the sample statistic.

In [6]:
```
m = meanBootstrap(accidents, 10000)
sample_mean = np.mean(m)
sample_se =  np.std(m)

print "Mean estimate:", sample_mean
print "SE of the estimate:",  sample_se

ci = [np.percentile(m, 2.5), np.percentile(m, 97.5)]
print  "Confidence interval:", ci
```

Out[6]:
```
Mean estimate: 25.9039
SE of the estimate: 0.4705
Confidence interval: [24.9834, 26.8219]
```

4.3.2.1 But What Does "95% Confident" Mean?

The real meaning of "confidence" is not evident and it must be understood from the point of view of the generating process.

Suppose we took many (infinite) samples from a population and built a 95% confidence interval from each sample. Then about 95% of those intervals would contain the actual parameter. In Fig. 4.3 we show how many confidence intervals computed from 100 different samples of 100 elements from our dataset contain the real population mean. If this simulation could be done with infinite different samples, 5% of those intervals would not contain the true mean.

So, when faced with a sample, the correct interpretation of a confidence interval is as follows:

> In 95% of the cases, when I compute the 95% confidence interval from this sample, the true mean of the population will fall within the interval defined by these bounds: $\pm 1.96 \times SE$.

We cannot say either that our specific sample contains the true parameter or that the interval has a 95% chance of containing the true parameter. That interpretation would not be correct under the assumptions of traditional statistics.

4.4 Hypothesis Testing

Giving a measure of the variability of our estimates is one way of producing a statistical proposition about the population, but not the only one. R.A. Fisher (1890–1962) proposed an alternative, known as *hypothesis testing*, that is based on the concept of *statistical significance*.

Let us suppose that a deeper analysis of traffic accidents in Barcelona results in a difference between 2010 and 2013. Of course, the difference could be caused only by chance, because of the variability of both estimates. But it could also be the case that traffic conditions were very different in Barcelona during the two periods and, because of that, data from the two periods can be considered as belonging to two different populations. Then, the relevant question is: Are the observed effects real or not?

Technically, the question is usually translated to: *Were the observed effects statistically significant*?

The process of determining the statistical significance of an effect is called *hypothesis testing*.

This process starts by simplifying the options into two competing hypotheses:

- H_0: The mean number of daily traffic accidents is the same in 2010 and 2013 (there is only one population, one true mean, and 2010 and 2013 are just different samples from the same population).
- H_A: The mean number of daily traffic accidents in 2010 and 2013 is different (2010 and 2013 are two samples from two different populations).

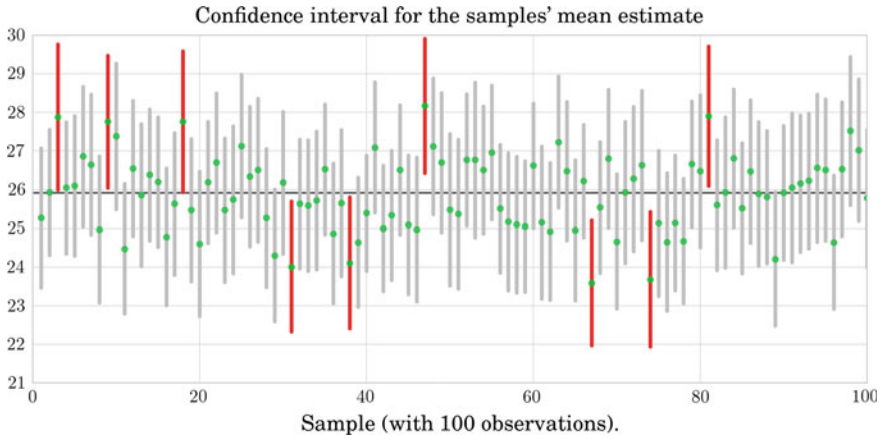


Fig. 4.3 This graph shows 100 sample means (*green points*) and its corresponding confidence intervals, computed from 100 different samples of 100 elements from our dataset. It can be observed that a few of them (those in *red*) do not contain the mean of the population (*black horizontal line*)

We call H_0 the *null hypothesis* and it represents a *skeptical* point of view: the effect we have observed is due to chance (due to the specific sample bias). H_A is the *alternative hypothesis* and it represents the other point of view: the effect is real.

The general rule of frequentist hypothesis testing: we will not *discard* H_0 (and hence we will not consider H_A) unless the observed effect is *implausible* under H_0.

4.4.1 Testing Hypotheses Using Confidence Intervals

We can use the concept represented by *confidence intervals* to measure the *plausibility* of a hypothesis.

We can illustrate the evaluation of the hypothesis setup by comparing the mean rate of traffic accidents in Barcelona during 2010 and 2013:

In [7]:
```
data = pd.read_csv("files/ch04/ACCIDENTS_GU_BCN_2010.csv",
                   encoding='latin-1')

# Create a new column which is the date
data['Date'] = data['Dia de mes'].apply(lambda x: str(x))
               + '-' +
               data['Mes de any'].apply(lambda x: str(x))
data2 = data['Date']
counts2010 = data['Date'].value_counts()
print '2010: Mean', counts2010.mean()

data = pd.read_csv("files/ch04/ACCIDENTS_GU_BCN_2013.csv",
                   encoding='latin-1')

# Create a new column which is the date
data['Date'] = data['Dia de mes'].apply(lambda x: str(x))
               + '-' +
               data['Mes de any'].apply(lambda x: str(x))
data2 = data['Date']
counts2013 = data['Date'].value_counts()
print '2013: Mean', counts2013.mean()
```

Out[7]:
```
2010: Mean 24.8109
2013: Mean 25.9095
```

This estimate suggests that in 2013 the mean rate of traffic accidents in Barcelona was higher than it was in 2010. But is this effect statistically significant?

Based on our sample, the 95% confidence interval for the mean rate of traffic accidents in Barcelona during 2013 can be calculated as follows:

In [8]:
```
n = len(counts2013)
mean = counts2013.mean()
s = counts2013.std()
ci = [mean - s*1.96/np.sqrt(n),  mean + s*1.96/np.sqrt(n)]
print '2010 accident rate estimate:', counts2010.mean()
print '2013 accident rate estimate:', counts2013.mean()
print 'CI for 2013:',ci
```

```
Out[8]: 2010 accident rate estimate: 24.8109
        2013 accident rate estimate: 25.9095
        CI for 2013: [24.9751, 26.8440]
```

Because the 2010 accident rate estimate does not fall in the range of plausible values of 2013, we say the alternative hypothesis cannot be discarded. That is, it cannot be ruled out that in 2013 the mean rate of traffic accidents in Barcelona was higher than in 2010.

Interpreting CI Tests

Hypothesis testing is built around rejecting or failing to reject the null hypothesis. That is, we do not reject H_0 unless we have strong evidence against it. But what precisely does strong evidence mean? As a general rule of thumb, for those cases where the null hypothesis is actually true, we do not want to incorrectly reject H_0 more than 5% of the time. This corresponds to a *significance level* of $\alpha = 0.05$. In this case, the correct interpretation of our test is as follows:

> If we use a 95% confidence interval to test a problem where the null hypothesis is true, we will make an error whenever the point estimate is at least 1.96 standard errors away from the population parameter. This happens about 5% of the time (2.5% in each tail).

4.4.2 Testing Hypotheses Using p-Values

A more advanced notion of *statistical significance* was developed by R.A. Fisher in the 1920s when he was looking for a test to decide whether variation in crop yields was due to some specific intervention or merely random factors beyond experimental control.

Fisher first assumed that fertilizer caused no difference (*null hypothesis*) and then calculated P, the probability that an observed yield in a fertilized field would occur if fertilizer had no real effect. This probability is called the *p-value*.

The p-value is the probability of observing data at least as favorable to the alternative hypothesis as our current dataset, if the null hypothesis is true. We typically use a summary statistic of the data to help compute the p-value and evaluate the hypotheses.

Usually, if P is less than 0.05 (the chance of a fluke is less than 5%) the result is declared *statistically significant*.

It must be pointed out that this choice is rather arbitrary and should not be taken as a scientific truth.

The goal of classical hypothesis testing is to answer the question, "*Given a sample and an apparent effect, what is the probability of seeing such an effect by chance?*" Here is how we answer that question:

- The first step is to quantify the size of the apparent effect by choosing a test statistic. In our case, the apparent effect is a difference in accident rates, so a natural choice for the test statistic is the **difference in means between the two periods**.

- The second step is to define a *null hypothesis*, which is a model of the system based on the assumption that the apparent effect is not real. In our case, the null hypothesis is that there is no difference between the two periods.
- The third step is to compute a *p-value*, which is the probability of seeing the apparent effect if the null hypothesis is true. In our case, we would compute the difference in means, then compute the probability of seeing a difference as big, or bigger, under the null hypothesis.
- The last step is to *interpret the result*. If the p-value is low, the effect is said to be *statistically significant*, which means that it is unlikely to have occurred by chance. In this case we infer that the effect is more likely to appear in the larger population.

In our case, the test statistic can be easily computed:

In [9]:
```
m= len(counts2010)
n= len(counts2013)
p = (counts2013.mean() - counts2010.mean())
print 'm:', m, 'n:', n
print 'mean difference: ', p
```

Out[9]:
```
m: 365 n: 365
mean difference: 1.0986
```

To approximate the p-value , we can follow the following procedure:

1. Pool the distributions, generate samples with size n and compute the difference in the mean.
2. Generate samples with size n and compute the difference in the mean.
3. Count how many differences are larger than the observed one.

In [10]:
```
# pooling distributions
x = counts2010
y = counts2013
pool = np.concatenate([x, y])
np.random.shuffle(pool)

#sample generation
import random
N = 10000 # number of samples
diff = range(N)
for i in range(N):
    p1 = [random.choice(pool) for _ in xrange(n)]
    p2 = [random.choice(pool) for _ in xrange(n)]
    diff[i] = (np.mean(p1) - np.mean(p2))
```

In [11]:
```
# counting differences larger than the observed one
diff2 = np.array(diff)
w1 = np.where(diff2 > p)[0]

print 'p-value (Simulation)=', len(w1)/float(N),
      '(', len(w1)/float(N)*100 ,'%)', 'Difference =', p
if (len(w1)/float(N)) < 0.05:
    print 'The effect is likely'
else:
    print 'The effect is not likely'
```

Out[11]:
```
p-value (Simulation)= 0.0485 ( 4.85%) Difference = 1.098
The effect is likely
```

Interpreting P-Values

A p-value is the probability of an observed (or more extreme) result arising only from chance.

If P is less than 0.05, there are two possible conclusions: there is a real effect or the result is an improbable fluke. *Fisher's method offers no way of knowing which is the case*.

We must not confuse the odds of getting a result (if a hypothesis is true) with the odds of favoring the hypothesis if you observe that result. If P is less than 0.05, we cannot say that this means that it is 95% certain that the observed effect is real and could not have arisen by chance. Given an observation E and a hypothesis H, $P(E|H)$ and $P(H|E)$ are not the same!

Another common error equates *statistical significance* to *practical importance/ relevance*. When working with large datasets, we can detect statistical significance for small effects that are meaningless in practical terms.

We have defined the effect as *a difference in mean as large or larger than* δ, *considering the sign*. A test like this is called *one sided*.

If the relevant question is whether *accident rates are different*, then it makes sense to test the absolute difference in means. This kind of test is called two sided because it counts both sides of the distribution of differences.

Direct Approach

The formula for the standard error of the absolute difference in two means is similar to the formula for other standard errors. Recall that the standard error of a single mean can be approximated by:

$$SE_{\bar{x}_1} = \frac{\sigma_1}{\sqrt{n_1}}$$

The standard error of the difference of two sample means can be constructed from the standard errors of the separate sample means:

$$SE_{\bar{x}_1 - \bar{x}_2} = \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}$$

This would allow us to define a direct test with the 95% confidence interval.

4.5 But Is the Effect E Real?

We do not yet have an answer for this question! We have defined a null hypothesis H_0 (the effect is not real) and we have computed the probability of the observed effect under the null hypothesis, which is $P(E|H_0)$, where E is an effect as big as or bigger than the apparent effect and a p-value .

We have stated that from the frequentist point of view, we cannot consider H_A unless $P(E|H_0)$ is less than an arbitrary value. But the real answer to this question must be based on comparing $P(H_0|E)$ to $P(H_A|E)$, not on $P(E|H_0)$! One possible solution to these problems is to use *Bayesian reasoning*; an alternative to the frequentist approach.

No matter how many data you have, you will still depend on intuition to decide how to interpret, explain, and use that data. Data cannot speak by themselves. Data scientists are interpreters, offering one interpretation of what the useful narrative story derived from the data is, if there is one at all.

4.6 Conclusions

In this chapter we have seen how we can approach the problem of making probable propositions regarding population parameters.

We have learned that in some cases, there are theoretical results that allow us to compute a measure of the variability of our estimates. We have called this approach the "traditional approach". Within this framework, we have seen that the sampling distribution of our parameter of interest is the most important concept when understanding the real meaning of propositions concerning parameters.

We have also learned that the traditional approach is not the only alternative. The "computationally intensive approach", based on the bootstrap method, is a relatively new approach that, based on intensive computer simulations, is capable of computing a measure of the variability of our estimates by applying a resampling method to our data sample. Bootstrapping can be used for computing variability of almost any function of our data, with its only downside being the need for greater computational resources.

We have seen that propositions about parameters can be classified into three classes: propositions about point estimates, propositions about set estimates, and propositions about the acceptance or the rejection of a hypothesis. All these classes are related; but today, set estimates and hypothesis testing are the most preferred.

Finally, we have shown that the production of probable propositions is not error free, even in the presence of big data. For these reason, data scientists cannot forget that after any inference task, they must take decisions regarding the final interpretation of the data.

Acknowledgements This chapter was co-written by Jordi Vitrià and Sergio Escalera.

References

1. M.I. Jordan. Are you a Bayesian or a frequentist? [Video Lecture]. Published: Nov. 2, 2009, Recorded: September 2009. Retrieved from: http://videolectures.net/mlss09uk_jordan_bfway/
2. B. Efron, R.J. Tibshirani, *An introduction to the bootstrap* (CRC press, 1994)

Supervised Learning

5

5.1 Introduction

Machine learning involves coding programs that automatically adjust their performance in accordance with their exposure to information in data. This learning is achieved via a parameterized model with tunable parameters that are automatically adjusted according to different performance criteria. Machine learning can be considered a subfield of artificial intelligence (AI) and we can roughly divide the field into the following three major classes.

1. Supervised learning: Algorithms which learn from a training set of labeled examples (exemplars) to generalize to the set of all possible inputs. Examples of techniques in supervised learning: logistic regression, support vector machines, decision trees, random forest, etc.
2. Unsupervised learning: Algorithms that learn from a training set of unlabeled examples. Used to explore data according to some statistical, geometric or similarity criterion. Examples of unsupervised learning include k-means clustering and kernel density estimation. We will see more on this kind of techniques in Chap. 7.
3. Reinforcement learning: Algorithms that learn via reinforcement from criticism that provides information on the quality of a solution, but not on how to improve it. Improved solutions are achieved by iteratively exploring the solution space.

This chapter focuses on a particular class of supervised machine learning: *classification*. As a data scientist, the first step you apply given a certain problem is to identify the question to be answered. According to the type of answer we are seeking, we are directly aiming for a certain set of techniques.

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_5

- If our question is answered by YES/NO, we are facing a classification problem. Classifiers are also the tools to use if our question admits only a discrete set of answers, i.e., we want to select from a finite number of choices.

  – Given the results of a clinical test, e.g., does this patient suffer from diabetes?
  – Given a magnetic resonance image, is it a tumor shown in the image?
  – Given the past activity associated with a credit card, is the current operation fraudulent?

- If our question is a prediction of a real-valued quantity, we are faced with a *regression* problem. We will go into details of regression in Chap. 6.

  – Given the description of an apartment, what is the expected market value of the flat? What will the value be if the apartment has an elevator?
  – Given the past records of user activity on Apps, how long will a certain client be connected to our App?
  – Given my skills and marks in computer science and maths, what mark will I achieve in a data science course?

Observe that some problems can be solved using both regression and classification. As we will see later, many classification algorithms are thresholded regressors. There is a certain skill involved in designing the correct question and this dramatically affects the solution we obtain.

5.2 The Problem

In this chapter we use data from the Lending Club[1] to develop our understanding of machine learning concepts. The Lending Club is a peer-to-peer lending company. It offers loans which are funded by other people. In this sense, the Lending Club acts as a hub connecting borrowers with investors. The client applies for a loan of a certain amount, and the company assesses the risk of the operation. If the application is accepted, it may or may not be fully covered. We will focus on the prediction of whether the loan will be fully funded, based on the scoring of and information related to the application.

We will use the partial dataset of period 2007–2011. Framing the problem a little bit more, based on the information supplied by the customer asking for a loan, we want to predict whether it will be granted up to a certain threshold thr. The attributes we use in this problem are related to some of the details of the loan application, such as amount of the loan applied for the borrower, monthly payment to be made by the borrower if the loan is accepted, the borrower's annual income, the number of

[1] https://www.lendingclub.com/info/download-data.action.

incidences of delinquency in the borrower's credit file, and interest rate of the loan, among others.

In this case we would like to predict unsuccessful accepted loans. A loan application is unsuccessful if the funded amount (`funded_amnt`) or the amount funded by investors (`funded_amnt_inv`) falls far short of the requested loan amount (`loan_amnt`). That is,

$$\frac{loan - funded}{loan} \geq 0.95.$$

5.3 First Steps

Note that in this problem we are predicting a binary value: either the loan is fully funded or not. Classification is the natural choice of machine learning tools for prediction with discrete known outcomes. According to the cardinality of the target set, one usually distinguishes between *binary* classifiers when the target output only takes two values, i.e., the classifier answers questions with a yes or a no; or *multiclass* classifiers, for a larger number of classes. This issue is important in that not all methods can naturally handle the multiclass setting.[2]

In a formal way, classification is regarded as the problem of finding a function $h(\mathbf{x}) : \mathbb{R}^d \to \mathbb{K}$ that maps an input space in $\mathbb{R}^d$ onto a discrete set of k target outputs or classes $\mathbb{K} = \{1, \ldots, k\}$. In this setting, the features are arranged as a vector $\mathbf{x}$ of d real-valued numbers.[3]

We can encode both target states in a numerical variable, e.g., a successful loan target can take value $+1$; and it is -1, otherwise.

Let us check the dataset,[4]

In [1]:
```
import pickle
ofname = open('./files/ch05/dataset_small.pkl','rb')
# x stores input data and y target values
(x,y) = pickle.load(ofname)
```

[2]Several well-known techniques such as support vector machines or adaptive boosting (adaboost) are originally defined in the binary case. Any binary classifier can be extended to the multiclass case in two different ways. We may either change the formulation of the learning/optimization process. This requires the derivation of a new learning algorithm capable of handling the new modeling. Alternatively, we may adopt ensemble techniques. The idea behind this latter approach is that we may divide the multiclass problem into several binary problems; solve them; and then aggregate the results. If the reader is interested in these techniques, it is a good idea to look for: one-versus-all, one-versus-one, or error correcting output codes methods.

[3]Many problems are described using categorical data. In these cases either we need classifiers that are capable of coping with this kind of data or we need to change the representation of those variables into numerical values.

[4]The notebook companion shows the preprocessing steps, from reading the dataset, cleaning and imputing data, up to saving a subsampled clean version of the original dataset.

A problem in Scikit-learn is modeled as follows:

- Input data is structured in Numpy arrays. The size of the array is expected to be [n_samples, n_features]:
  - n_samples: The number of samples (n). Each sample is an item to process (e.g., classify). A sample can be a document, a picture, an audio file, a video, an astronomical object, a row in a database or CSV file, or whatever you can describe with a fixed set of quantitative traits.
  - n_features: The number of features (d) or distinct traits that can be used to describe each item in a quantitative manner. Features are generally real-valued, but may be Boolean, discrete-valued or even categorical.

$$\text{feature matrix}: \quad \mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ x_{31} & x_{32} & \cdots & x_{3d} \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{nd} \end{bmatrix}$$

$$\text{label vector}: \quad \mathbf{y}^T = [y_1, y_2, y_3, \cdots y_n]$$

The number of features must be fixed in advance. However, it can be very great (e.g., millions of features).

In [2]:
```
dims = x.shape[1]
N = x.shape[0]
print 'dims: ' + str(dims) + ', samples: ' + str(N)
```

Out[2]:
```
dims: 15, samples: 4140
```

Considering data arranged as in the previous matrices we refer to:

- the columns as features, attributes, dimensions, regressors, covariates, predictors, or independent variables;
- the rows as instances, examples, or samples;
- the target as the label, outcome, response, or dependent variable.

All objects in Scikit-learn share a uniform and limited API consisting of three complementary interfaces:

- an estimator interface for building and fitting models (`fit()`);
- a predictor interface for making predictions (`predict()`);
- a transformer interface for converting data (`transform()`).

Let us apply a classifier using Python's Scikit-learn libraries,

In [3]:
```
from sklearn import neighbors
from sklearn import datasets
# Create an instance of K-nearest neighbor classifier
knn = neighbors.KNeighborsClassifier(n_neighbors = 11)
# Train the classifier
knn.fit(x, y)
# Compute the prediction according to the model
yhat = knn.predict(x)
# Check the result on the last example
print 'Predicted value: ' + str(yhat[-1]),
      ', real target: ' + str(y[-1])
```

Out[3]:
```
Predicted value: -1.0 , real target: -1.0
```

The basic measure of performance of a classifier is its *accuracy*. This is defined as the number of correctly predicted examples divided by the total amount of examples. Accuracy is related to the error as follows: $acc = 1 - err$.

$$acc = \frac{\text{Number of correct predictions}}{n}$$

Each estimator has a `score()` method that invokes the default scoring metric. In the case of k-nearest neighbors, this is the classification accuracy.

In [4]:
```
knn.score(x,y)
```

Out[4]:
```
0.83164251207729467
```

It looks like a really good result. But how good is it? Let us first understand a little bit more about the problem by checking the distribution of the labels.

Let us load the dataset and check the distribution of labels:

In [5]:
```
plt.pie(np.c_[np.sum(np.where(y == 1, 1, 0)),
        np.sum(np.where(y == -1, 1, 0))][0],
        labels = ['Not fully funded','Full amount'],
        colors = ['r', 'g'],shadow = False,
        autopct = '%.2f' )
plt.gcf().set_size_inches((7, 7))
```

with the result observed in Fig. 5.1.

Note that there are far more positive labels than negative ones. In this case, the dataset is referred to as *unbalanced*.[5] This has important consequences for a classifier as we will see later on. In particular, a very simple rule such as always predict the

[5]The term unbalanced describes the condition of data where the ratio between positives and negatives is a small value. In these scenarios, always predicting the majority class usually yields accurate performance, though it is not very informative. This kind of problems is very common when we want to model unusual events such as rare diseases, the occurrence of a failure in machinery, fraudulent credit card operations, etc. In these scenarios, gathering data from usual events is very easy but collecting data from unusual events is difficult and results in a comparatively small dataset.

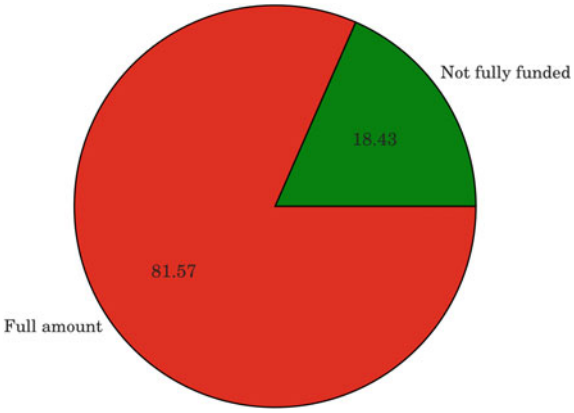


Fig. 5.1 Pie chart showing the distribution of labels in the dataset

majority class, will give us good performance. In our problem, always predicting that the loan will be fully funded correctly predicts 81.57% of the samples. Observe that this value is very close to that obtained using the classifier.

Although accuracy is the most normal metric for evaluating classifiers, there are cases when the business value of correctly predicting elements from one class is different from the value for the prediction of elements of another class. In those cases, accuracy is not a good performance metric and more detailed analysis is needed. The *confusion matrix* enables us to define different metrics considering such scenarios. The confusion matrix considers the concepts of the classifier outcome and the actual ground truth or gold standard. In a binary problem, there are four possible cases:

- *True positives (TP)*: When the classifier predicts a sample as positive and it really is positive.
- *False positives (FP)*: When the classifier predicts a sample as positive but in fact it is negative.
- *True negatives (TN)*: When the classifier predicts a sample as negative and it really is negative.
- *False negatives (FN)*: When the classifier predicts a sample as negative but in fact it is positive.

We can summarize this information in a matrix, namely the confusion matrix, as follows:

		Gold Standard: Positive	Gold Standard: Negative	
Prediction	Positive	TP	FP	→ Precision
	Negative	FN	TN	→ Negative Predictive Value
		↓	↓	
		Sensitivity (Recall)	Specificity	

The combination of these elements allows us to define several performance metrics:

- *Accuracy:*

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

- Column-wise we find these two partial performance metrics:
  - *Sensitivity or Recall:*

$$\text{sensitivity} = \frac{\text{TP}}{\text{Real Positives}} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

  - *Specificity:*

$$\text{specificity} = \frac{\text{TN}}{\text{Real Negatives}} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

- Row-wise we find these two partial performance metrics:
  - *Precision or Positive Predictive Value:*

$$\text{precision} = \frac{\text{TP}}{\text{Predicted Positives}} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

  - *Negative predictive value:*

$$\text{NPV} = \frac{\text{TN}}{\text{Predicted Negative}} = \frac{\text{TN}}{\text{TN} + \text{FN}}$$

These partial performance metrics allow us to answer questions concerning how often a classifier predicts a particular class, e.g., what is the rate of predictions for not fully funded loans that have actually not been fully funded? This question is answered by recall. In contrast, we could ask: Of all the fully funded loans predicted by the classifier, how many have been fully funded? This is answered by the precision metric.

Let us compute these metrics for our problem.

In [6]:
```
yhat = knn.predict(x)
TP = np.sum(np.logical_and(yhat == -1, y == -1))
TN = np.sum(np.logical_and(yhat == 1, y == 1))
FP = np.sum(np.logical_and(yhat == -1, y == 1))
FN = np.sum(np.logical_and(yhat == 1, y == -1))
print 'TP: '+ str(TP), ', FP: '+ str(FP)
print 'FN: '+ str(FN), ', TN: '+ str(TN)
```

Out[6]:
```
TP: 3370 , FP: 690
 FN: 7 ,  TN: 73
```

Scikit-learn provides us with the confusion matrix,

In [7]:
```
from sklearn import metrics
metrics.confusion_matrix(yhat, y)
# sklearn uses a transposed convention for the confusion
# matrix thus I change targets and predictions
```

Out[7]:
```
3370, 690
 7,   73
```

Let us check the following example. Let us select a nearest neighbor classifier with the number of neighbors equal to one instead of eleven, as we did before, and check the training error.

In [8]:
```
# Train a classifier using .fit()
knn = neighbors.KNeighborsClassifier(n_neighbors = 1)
knn.fit(x, y)
yhat = knn.predict(x)

print "classification accuracy:" +
      str(metrics.accuracy_score(yhat, y))
print "confusion matrix: \n" +
      str(metrics.confusion_matrix(yhat, y))
```

Out[8]:
```
classification accuracy: 1.0 confusion matrix:
3377  0
 0   763
```

The performance measure is perfect! 100% accuracy and a diagonal confusion matrix! This looks good. However, up to this point we have checked the classifier performance on the same data it has been trained with. During exploitation, in real applications, we will use the classifier on data not previously seen. Let us simulate this effect by splitting the data into two sets: one will be used for learning (*training set*) and the other for testing the accuracy (*test set*).

In [9]:
```
# Simulate a real case: Randomize and split data into
# two subsets PRC*100\% for training and the rest
# (1-PRC)*100\% for testing
perm = np.random.permutation(y.size)
PRC = 0.7
split_point = int(np.ceil(y.shape[0]*PRC))

X_train = x[perm[:split_point].ravel(),:]
y_train = y[perm[:split_point].ravel()]

X_test = x[perm[split_point:].ravel(),:]
y_test = y[perm[split_point:].ravel()]
```

If we check the shapes of the training and test sets we obtain,

Out[9]:
```
Training shape: (2898, 15), training targets shape: (2898,)
Testing shape: (1242, 15), testing targets shape: (1242,)
```

With this new partition, let us train the model

In [10]:
```
#Train a classifier on training data
knn = neighbors.KNeighborsClassifier(n_neighbors = 1)
knn.fit(X_train, y_train)
yhat = knn.predict(X_train)

print "\n TRAINING STATS:"
print "classification accuracy:" +
      str(metrics.accuracy_score(yhat, y_train))
print "confusion matrix: \n" +
      str(metrics.confusion_matrix(y_train, yhat))
```

Out[10]:
```
TRAINING STATS:
classification accuracy: 1.0
confusion matrix:
2355  0
  0   543
```

As expected from the former experiment, we achieve a perfect score. Now let us see what happens in the simulation with previously unseen data.

In [11]:
```
#Check on the test set
yhat = knn.predict(X_test)
print "TESTING STATS:"
print "classification accuracy:",
      metrics.accuracy_score(yhat, y_test)
print "confusion matrix: \n" +
      str(metrics.confusion_matrix(yhat, y_test))
```

Out[11]:
```
TESTING STATS:
classification accuracy: 0.754428341385
confusion matrix:
865 148
157  72
```

Observe that each time we run the process of randomly splitting the dataset and train a classifier we obtain a different performance. A good simulation for approximating the test error is to run this process many times and average the performances. Let us do this![6]

In [12]:
```
# Spitting done by using the tools provided by sklearn:
from sklearn.cross_validation import train_test_split

PRC = 0.3
acc = np.zeros((10,))
for i in xrange(10):
    X_train, X_test, y_train, y_test =
        train_test_split(x, y, test_size = PRC)
    knn = neighbors.KNeighborsClassifier(n_neighbors = 1)
    knn.fit(X_train, y_train)
    yhat = knn.predict(X_test)
    acc[i] = metrics.accuracy_score(yhat, y_test)
acc.shape = (1, 10)
print "Mean expected error:" + str(np.mean(acc[0]))
```

Out[12]:
```
Mean expected error: 0.754669887279
```

As we can see, the resulting error is below 81%, which was the result of the most naive decision process. What is wrong with this result?

Let us introduce the nomenclature for the quantities we have just computed and define the following terms.

- *In-sample error* E_{in}: The in-sample error or training error is the error measured over all the observed data samples in the training set, i.e.,

$$E_{\text{in}} = \frac{1}{N} \sum_{i=1}^{N} e(x_i, y_i)$$

- *Out-of-sample error* E_{out}: The out-of-sample error or generalization error measures the expected error on unseen data. We can approximate/simulate this quantity by holding back some training data for testing purposes.

$$E_{\text{out}} = \mathbb{E}_{x,y}(e(x, y))$$

Note that the definition of the instantaneous error $e(x_i, y_i)$ is still missing. For example, in classification we could use the indicator function to account for a correctly classified sample as follows:

$$e(x_i, y_i) = I[h(x_i) = y_i] = \begin{cases} 1, \text{ if } \ h(x_i) = y_i \\ 0 \text{ otherwise.} \end{cases}$$

[6] *sklearn* allows us to easily automate the train/test splitting using the function `train_test_split(...)`.

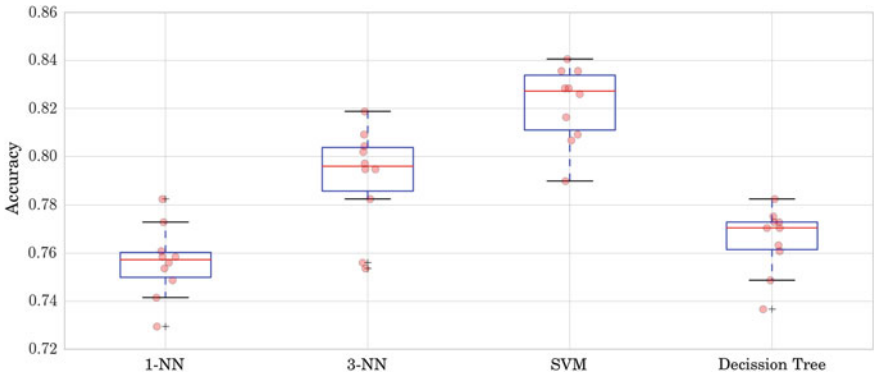


Fig. 5.2 Comparison of the methods using the accuracy metric

Observe that:

$$E_{\text{out}} \geq E_{\text{in}}$$

Using the expected error on the test set, we can select the best classifier for our application. This is called model selection. In this example we cover the most simplistic setting. Suppose we have a set of different classifiers and want to select the "best" one. We may use the one that yields the lowest error rate.

In [13]:
```
from sklearn import tree
from sklearn import svm
PRC = 0.1
acc_r = np.zeros((10, 4))
for i in xrange(10):
    X_train, X_test, y_train, y_test =
        train_test_split(x, y, test_size = PRC)
    nn1 = neighbors.KNeighborsClassifier(n_neighbors = 1)
    nn3 = neighbors.KNeighborsClassifier(n_neighbors = 3)
    svc = svm.SVC()
    dt = tree.DecisionTreeClassifier()

    nn1.fit(X_train, y_train)
    nn3.fit(X_train, y_train)
    svc.fit(X_train, y_train)
    dt.fit(X_train, y_train)

    yhat_nn1 = nn1.predict(X_test)
    yhat_nn3 = nn3.predict(X_test)
    yhat_svc = svc.predict(X_test)
    yhat_dt = dt.predict(X_test)

    acc_r[i][0] = metrics.accuracy_score(yhat_nn1, y_test)
    acc_r[i][1] = metrics.accuracy_score(yhat_nn3, y_test)
    acc_r[i][2] = metrics.accuracy_score(yhat_svc, y_test)
    acc_r[i][3] = metrics.accuracy_score(yhat_dt, y_test)
```

Figure 5.2 shows the results of applying the code.

This process is one particular form of a general model selection technique called *cross-validation*. There are other kinds of cross-validation, such as *leave-one-out* or *K-fold cross-validation*.

- In leave-one-out, given N samples, the model is trained with $N - 1$ samples and tested with the remaining one. This is repeated N times, once per training sample and the result is averaged.
- In K-fold cross-validation, the training set is divided into K nonoverlapping splits. K-1 splits are used for training and the remaining one used to assess the mean. This process is repeated K times leaving one split out each time. The results are then averaged.

5.4 What Is Learning?

Let us recall the two basic values defined in the last section. We talk of *training error* or in-sample error, E_{in}, which refers to the error measured over all the observed data samples in the training set. We also talk of *test error* or *generalization error*, E_{out}, as the error expected on unseen data.

We can empirically estimate the generalization error by means of cross-validation techniques and observe that:

$$E_{\text{out}} \geq E_{\text{in}}.$$

The goal of learning is to minimize the generalization error; but how can we guarantee this minimization using only training data?

From the above inequality it is easy to derive a couple of very intuitive ideas.

- Because E_{out} is greater than or equal to E_{in}, it is desirable to have
$$E_{\text{in}} \to 0.$$
- Additionally, we also want the training error behavior to track the generalization error so that if one minimizes the in-sample error the out-of-sample error follows, i.e.,
$$E_{\text{out}} \approx E_{\text{in}}.$$

We can rewrite the second condition as

$$E_{\text{in}} \leq E_{\text{out}} \leq E_{\text{in}} + \Omega,$$

with $\Omega \to 0$.

We would like to characterize Ω in terms of our problem parameters, i.e., the number of samples (N), dimensionality of the problem (d), etc.

Statistical analysis offers an interesting characterization of this quantity[7]

[7]The reader should note that there are several bounds in machine learning to characterize the generalization error. Most of them come from variations of Hoeffding's inequality.

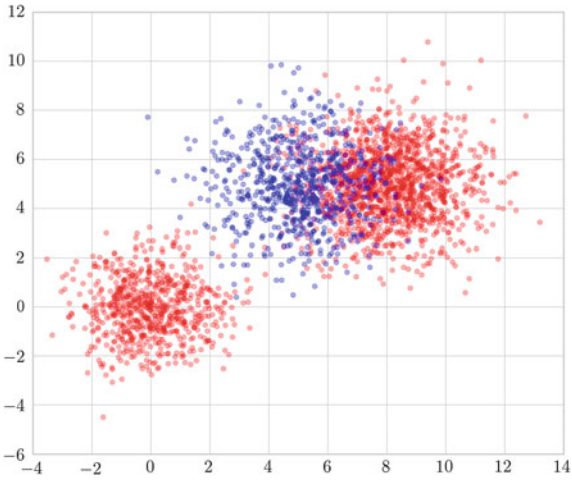


Fig. 5.3 Toy problem data

$$E_{\text{out}} \leq E_{\text{in}}(C) + \mathcal{O}\left(\sqrt{\frac{\log C}{N}}\right),$$

where C is a measure of the complexity of the model class we are using. Technically, we may also refer to this model class as the hypothesis space.

5.5 Learning Curves

Let us simulate the effect of the number of examples on the training and test errors for a given complexity. This curve is called the *learning curve*. We will focus for a moment in a more simple case. Consider the toy problem in Fig. 5.3.

Let us take a classifier and vary the number of examples we feed it for training purposes, then check the behavior of the training and test accuracies as the number of examples grows. In this particular case, we will be using a decision tree with fixed maximum depth.

Observing the plot in Fig. 5.4, we can see that:

- As the number of training samples increases, both errors tend to the same value.
- When we have few training data, the training error is very small but the test error is very large.

Now check the learning curve when the degree of complexity is greater in Fig. 5.5. We simulate this effect by increasing the maximum depth of the tree.

And if we put both curves together, we have the results shown in Fig. 5.6.

Although both show similar behavior, we can note several differences:

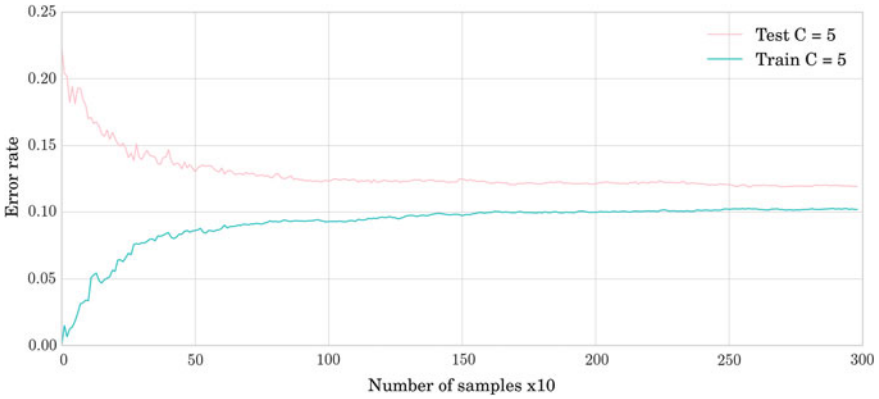


Fig. 5.4 Learning curves (training and test errors) for a model with a high degree of complexity

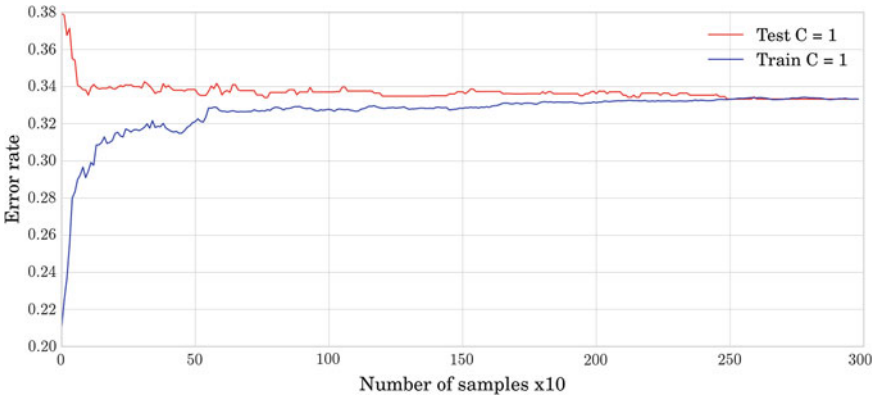


Fig. 5.5 Learning curves (training and test errors) for a model with a low degree of complexity

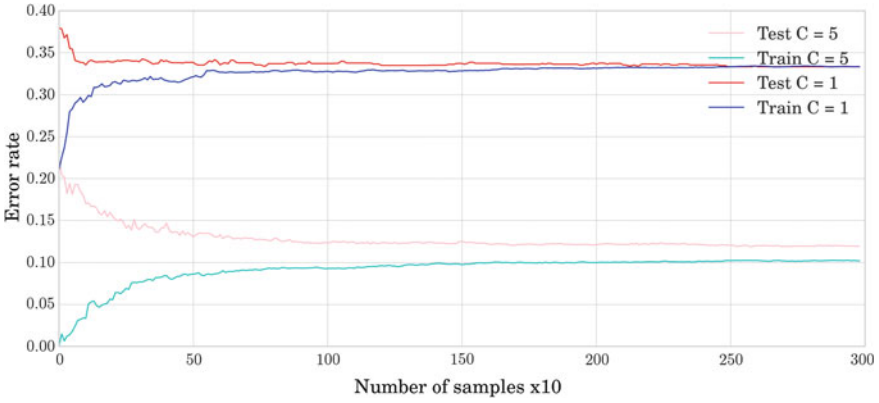


Fig. 5.6 Learning curves (training and test errors) for models with a low and a high degree of complexity

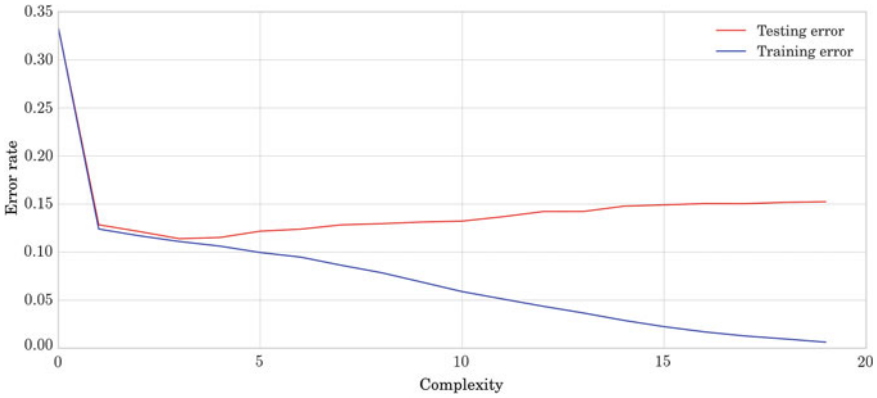


Fig. 5.7 Learning curves (training and test errors) for a fixed number of data samples, as the complexity of the decision tree increases

- With a low degree of complexity, the training and test errors converge to the bias sooner/with fewer data.
- Moreover, with a low degree of complexity, the error of convergence is larger than with increased complexity.

The value both errors converge towards is also called the *bias*; and the difference between this value and the test error is called the *variance*. The *bias/variance* decomposition of the learning curve is an alternative approach to the training and generalization view.

Let us now plot the learning behavior for a fixed number of examples with respect to the complexity of the model. We may use the same data but now we will change the maximum depth of the decision tree, which governs the complexity of the model.

Observe in Fig. 5.7 that as the complexity increases the training error is reduced; but above a certain level of complexity, the test error also increases. This effect is called *overfitting*. We may enact several cures for overfitting:

- Observe that models are usually parameterized by some hyperparameters. Selecting the complexity is usually governed by some such parameters. Thus, we are faced with a model selection problem. A good heuristic for selecting the model is to choose the value of the hyperparameters that yields the smallest estimated test error. Remember that this can be done using cross-validation.
- We may also change the formulation of the objective function to penalize complex models. This is called *regularization*. Regularization accounts for estimating the value of Ω in our out-of-sample error inequality. In other words, it models the complexity of the technique. This usually becomes implicit in the algorithm but has huge consequences in real applications. The most common regularization strategies are as follows:

  - L2 weight regularization: Adding an L2 penalization term to the weights of a weight-controlled model implies looking for solutions with small weight values. Intuitively, adding an L2 penalization term can be seen as a surrogate for the notion of smoothness. In this sense, a low complexity model means a very smooth model.
  - L1 weight regularization: Adding an L1 regularization term forces sparsity in the weights of the model. In this sense, a low complexity model means a model with few components or few active terms.

  These terms are added to the objective function. They trade off with the error function in the objective and are governed by a hyperparameter. Thus, we still have to select this parameter by means of model selection.
- We can use "ensemble techniques". A third cure for overfitting is to use ensemble techniques. The best known are *bagging* and *boosting*.

5.6 Training, Validation and Test

Going back to our problem, we have to select a model and control its complexity according to the number of training data. In order to do this, we can start by using a model selection technique. We have seen model selection before when we wanted to compare the performance of different classifiers. In that case, our best bet was to select the classifier with the smallest E_{out}. Analogous to model selection, we may think of selecting the best hyperparameters as choosing the classifier with parameters that performs the best. Thus, we may select a set of hyperparameter values and use cross-validation to select the best configuration.

The process of selecting the best hyperparameters is called *validation*. This introduces a new set into our simulation scheme; we now need to divide the data we have into three sets: training, validation, and test sets. As we have seen, the process of assessing the performance of the classifier by estimating the generalization error is called testing. And the process of selecting a model using the estimation of the generalization error is called validation. There is a subtle but critical difference between the two and we have to be aware of it when dealing with our problem.

- Test data is used exclusively for assessing performance at the end of the process and will never be used in the learning process.[8]
- Validation data is used explicitly to select the parameters/models with the best performance according to an estimation of the generalization error. This is a form of learning.
- Training data are used to learn the instance of the model from a model class.

[8]This set cannot be used to select a classifier, model or hyperparameter; nor can it be used in any decision process.

In practice, we are just given training data, and in the most general case we explicitly have to tune some hyperparameter. Thus, how do we select the different splits?

How we do this will depend on the questions regarding the method that we want to answer:

- Let us say that our customer asks us to deliver a classifier for a given problem. If we just want to provide the best model, then we may use cross-validation on our training dataset and select the model with the best performance. In this scenario, when we return the trained classifier to our customer, we know that it is the one that achieves the best performance. But if the customer asks about the expected performance, we cannot say anything.
  A practical issue: once we have selected the model, we use the complete training set to train the final model.
- If we want to know about the performance of our model, we have to use unseen data. Thus, we may proceed in the following way:

1. Split the original dataset into training and test data. For example, use 30% of the original dataset for testing purposes. This data is held back and will only be used to assess the performance of the method.
2. Use the remaining training data to select the hyperparameters by means of cross-validation.
3. Train the model with the selected parameter and assess the performance using the test dataset.

A practical issue: Observe that by splitting the data into three sets, the classifier is trained with a smaller fraction of the data.

- If we want to make a good comparison of classifiers but we do not care about the best parameters, we may use *nested cross-validation*. Nested cross-validation runs two cross-validation processes. An external cross-validation is used to assess the performance of the classifier and in each loop of the external cross-validation another cross-validation is run with the remaining training set to select the best parameters.

If we want to select the best complexity of a decision tree, we can use tenfold cross-validation checking for different complexity parameters. If we change the maximum depth of the method, we obtain the results in Fig. 5.8.

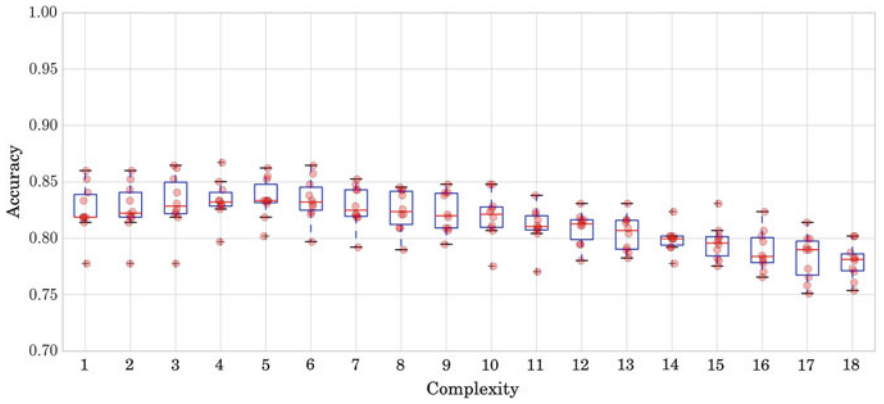


Fig. 5.8 Box plot showing accuracy for different complexities of the decision tree

In [14]:

```
# Create a 10-fold  cross-validation set
kf = cross_validation.KFold(n = y.shape[0],
                            n_folds = 10,
                            shuffle = True,
                            random_state = 0)

# Search for the parameter among the following:
C = np.arange(2, 20,)

acc = np.zeros((10, 18))
i = 0
for train_index, val_index in kf:
    X_train, X_val = X[train_index], X[val_index]
    y_train, y_val = y[train_index], y[val_index]
    j = 0
    for c in C:
        dt = tree.DecisionTreeClassifier(
            min_samples_leaf = 1,
            max_depth = c)
        dt.fit(X_train, y_train)
        yhat = dt.predict(X_val)
        acc[i][j] = metrics.accuracy_score(yhat, y_val)
        j = j + 1
    i = i + 1
```

Checking Fig. 5.8, we can see that the best average accuracy is obtained by the fifth model, a maximum depth of 6. Although we can report that the best accuracy is estimated to be found with a complexity value of 6, we cannot say anything about the value it will achieve. In order to have an estimation of that value, we need to run the model on a new set of data that are completely unseen, both in training and in model selection (the model selection value is positively biased). Let us put everything together. We will be considering a simple train_test split for testing purposes and then run cross-validation for model selection.

In [15]:
```
# Train_test split
X_train, X_test, y_train, y_test = cross_validation
    .train_test_split(X, y, test_size = 0.20)

# Create a 10-fold  cross-validation set
kf = cross_validation.KFold(n = y_train.shape[0],
                            n_folds = 10,
                            shuffle = True,
                            random_state = 0)

# Search the parameter among the following
C = np.arange(2, 20,)
acc = np.zeros((10, 18))
i = 0
for train_index, val_index in kf:
    X_t, X_val = X_train[train_index], X_train[val_index]
    y_t, y_val = y_train[train_index], y_train[val_index]
    j = 0
    for c in C:
        dt = tree.DecisionTreeClassifier(
            min_samples_leaf = 1,
            max_depth = c)
        dt.fit(X_t, y_t)
        yhat = dt.predict(X_val)
        acc[i][j] = metrics.accuracy_score(yhat, y_val)
        j = j + 1
    i = i + 1
print 'Mean accuracy: ' + str(np.mean(acc, axis = 0))
print 'Selected model index: ' +
      str(np.argmax(np.mean(acc, axis = 0)))
```

Out[15]:
```
Mean accuracy: [0.8254832 0.83031158 0.83091854 0.83423816
0.83363939 0.83303516 0.82759983 0.82337022 0.82034725
0.81642795 0.80947567 0.79951316 0.80162614 0.79226695
0.79589324 0.785928 0.78049267 0.78320988]
Selected model index: 3
```

If we run the output of this code, we observe that the best accuracy is provided by the fourth model. In this example it is a model with complexity 5.[9] The selected model achieves a success rate of 0.83423816 in validation. We then train the model with the complete training set and verify its test accuracy.

[9] This reduction in the complexity of the best model should not surprise us. Remember that complexity and the number of examples are intimately related for the learning to succeed. By using a test set we perform model selection with a smaller dataset than in the former case.

In [16]:
```
# Train the model with the complete training set with the
    selected complexity
dt = tree.DecisionTreeClassifier(
    min_samples_leaf = 1,
    max_depth = C[np.argmax(np.mean(acc, axis = 0))])
dt.fit(X_train,y_train)

# Test the model with the test set
yhat = dt.predict(X_test)
print 'Test accuracy: ' +
      str(metrics.accuracy_score(yhat, y_test))
```

Out[16]:
```
Test accuracy: 0.826086956522
```

As expected, the value is slightly reduced; it achieves 0.82608. Finally, the model is trained with the complete dataset. This will be the model used in exploitation and we expect to at least achieve an accuracy rate of 0.82608.

In [17]:
```
# Train the final model
dt = tree.DecisionTreeClassifier(min_samples_leaf = 1,
    max_depth = C[np.argmax(np.mean(acc, axis = 0))])
dt.fit(X, y)
```

5.7 Two Learning Models

Let us return to our problem and check the performance of different models. There are many learning models in the machine learning literature. However, in this short introduction we focus on two of the most important and pragmatically effective approaches[10]: support vector machines (SVM) and random forests (RF).

5.7.1 Generalities Concerning Learning Models

Before going into some of the details of the models selected, let us check the components of any learning algorithm. In order to be able to learn, an algorithm has to define at least three components:

- *The model class/hypothesis space* defines the family of mathematical models that will be used. The target decision boundary will be approximated from one element of this space. For example, we can consider the class of linear models. In this case our decision boundary will be a line if the problem is defined in $\mathbf{R}^2$ and the model class is the space of all possible lines in $\mathbf{R}^2$.

[10] These techniques have been shown to be two of the most powerful families for classification [1].

Model classes define the geometric properties of the decision function. There are different taxonomies but the best known are the families of *linear* and *nonlinear* models. These families usually depend on some parameters; and the solution to a learning problem is the selection of a particular set of parameters, i.e., the selection of an instance of a model from the model class space. The model class space is also called the *hypothesis space*.
The selection of the best model will depend on our problem and what we want to obtain from the problem. The primary goal in learning is usually to achieve the minimum error/maximum performance; but according to what else we want from the algorithm, we can come up with different algorithms. Other common desirable properties are interpretability, behavior when faced with missing data, fast training, etc.

- *The problem model* formalizes and encodes the desired properties of the solution. In many cases, this formalization takes the form of an optimization problem. In its most basic instantiation, the problem model can be the *minimization of an error function*. The error function measures the difference between our model and the target. Informally speaking, in a classification problem it measures how "irritated" we are when our model misses the right label for a training sample. For example, in classification, the ideal error function is the *0–1 loss*. This function takes value 1 when we incorrectly classify a training sample and zero otherwise. In this case, we can interpret it by saying that we are only irritated by "one unit of irritation" when one sample is misclassified.
  The problem model can also be used to impose other constraints on our solution,[11] such as finding a smooth approximation, a model with a low degree of small complexity, a sparse solution, etc.
- *The learning algorithm* is an optimization/search method or algorithm that, given a model class, fits it to the training data according to the error function. According to the nature of our problem there are many different algorithms. In general, we are talking about finding the minimum error approximation or maximum probable model. In those cases, if the problem is convex/quasi-convex we will typically use first- or second-order methods (i.e., gradient descent, coordinate descent, Newton's method, interior point methods, etc.). Other searching techniques such as genetic algorithms or Monte Carlo techniques can be used if we do not have access to the derivatives of the objective function.

5.7.2 Support Vector Machines

SVM is a learning technique initially designed to fit a linear boundary between the samples of a binary problem, ensuring the maximum robustness in terms of tolerance to isotropic uncertainty. This effect is observed in Fig. 5.9. Note that the boundary displayed has the largest distance to the closest point of both classes. Any other

[11] Remember the regularization cure for overfitting.

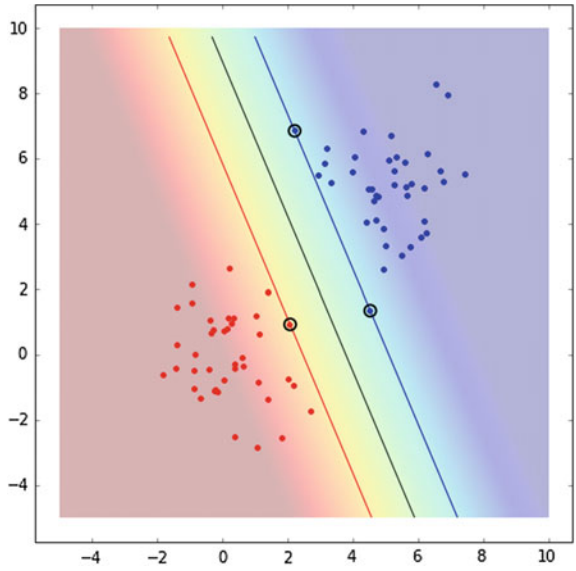


Fig. 5.9 Support vector machine decision boundary and the support vectors

separating boundary will have a point of a class closer to it than this one. The figure also shows the closest points of the classes to the boundary. These points are called *support vectors*. In fact, the boundary only depends on those points. If we remove any other point from the dataset, the boundary remains intact. However, in general, if any of these special points is removed the boundary will change.

5.7.2.1 A Brief Note on Deriving Hard Margin Support Vector Machines

In order to understand the model, we have to be able to approximately derive its formulation. For this purpose it is important to understand a couple of things about basic geometry of a hyperplane. A hyperplane in $\mathbf{R}^d$ is defined as an affine combination of the variables: $\pi \equiv a^T x + b = 0$. A hyperplane splits the space into two half-spaces. The evaluation of the equation of the hyperplane on any element belonging to one of the half-spaces is a positive value. It is a negative value for all the elements in the other half-space. The distance of a point $x \in \mathbf{R}^d$ to the hyperplane π is

$$d(x, \pi) = \frac{|a^T x + b|}{\|a\|_2}$$

Given a binary classification problem with training data $\mathcal{D} = \{(x_i, y_i)\}$, $i = 1 \ldots N$, $y_i \in \{+1, -1\}$, consider $\mathcal{S} \subseteq \mathcal{D}$ the subset of all data points belonging to class $+1$, $\mathcal{S} = \{x_i | y_i = +1\}$, and $\mathcal{R} = \{x_i | y_i = -1\}$ its complement.

Then the problem of finding a separating hyperplane consists of fulfilling the following constraints[12]

$$a^T s_i + b > 0 \text{ and } a^T r_i + b < 0 \quad \forall s_i \in \mathcal{S}, r_i \in \mathcal{R}.$$

This is a *feasibility problem* and it is usually written in the following way in optimization standard notation:

$$\begin{aligned} &\text{minimize} \quad 1 \\ &\text{subject to} \quad y_i(a^T x_i + b) \geq 1, \ \forall x_i \in \mathcal{D} \end{aligned}$$

The solution of this problem is not unique. Selecting the maximum margin hyperplane requires us to add a new constraint to our problem. Remember from the geometry of the hyperplane that the distance of any point to a hyperplane is given by: $d(x, \pi) = \frac{a^T x + b}{\|a\|_2}$.

Recall also that we want positive data to be beyond value 1 and negative data below -1. Thus, what is the distance value we want to maximize?

The positive point closest to the boundary is at $1/\|a\|_2$ and the negative point closest to the boundary data point is also at $1/\|a\|_2$. Thus, data points from different classes are at least $2/\|a\|_2$ apart.

Recall that our goal is to find the separating hyperplane with maximum margin, i.e., with maximum distance between elements in the different classes. Thus, we can complete the former formulation with our last requirement as follows:

$$\begin{aligned} &\text{minimize} \quad \|a\|_2/2 \\ &\text{subject to} \quad y_i(a^T x_i + b) \geq 1, \ \forall x_i \in \mathcal{D} \end{aligned}$$

This formulation has a solution as long as the problem is linearly separable.

In order to deal with misclassifications, we are going to introduce a new set of variables ξ_i, that represents the amount of violation in the i-th constraint. If the constraint is already satisfied, then $\xi_i = 0$; while $\xi_i > 0$ otherwise. Because ξ_i is related to the errors, we would like to keep this amount as close to zero as possible. This makes us introduce an element in the objective trade-off with the maximum margin.

[12]Note the strict inequalities in the formulation. Informally, we can consider the smallest satisfied constraint, and observe that the rest must be satisfied with a larger value. Thus, we can arbitrarily set that value to 1 and rewrite the problem as

$$a^T s_i + b \geq 1 \text{ and } a^T r_i + b \leq -1.$$

The new model becomes:

$$\begin{aligned} \text{minimize} \quad & \|a\|_2/2 + C\sum_{i=1}^{N}\xi_i \\ \text{subject to} \quad & y_i(a^T x_i + b) \geq 1 - \xi_i,\ i = 1 \ldots N \\ & \xi_i \geq 0 \end{aligned}$$

where C is the trade-off parameter that roughly balances the rates of margin and misclassification. This formulation is also called *soft-margin SVM*.

The larger the C value is, the more importance one gives to the error, i.e., the method will be more accurate according to the data at hand, at the cost of being more sensitive to variations of the data.

The decision boundary of most problems cannot be well approximated by a linear model. In SVM, the extension to the nonlinear case is handled by means of kernel theory. In a pragmatic way, a kernel can be referred to as any function that captures the similarity between any two samples in the training set. The kernel has to be a positive semi-definite function as follows:

- *Linear kernel*:
$$k(x_i, x_j) = x_i^T x_j$$
- *Polynomial kernel*:
$$k(x_i, x_j) = (1 + x_i^T x_j)^p$$
- *Radial Basis Function kernel*:
$$k(x_i, x_j) = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$$

Note that selecting a polynomial or a Radial Basis Function kernel means that we have to adjust a second parameter p or σ, respectively. As a practical summary, the SVM method will depend on two parameters (C, γ) that have to be chosen carefully using cross-validation to obtain the best performance.

5.7.3 Random Forest

Random Forest (RF) is the other technique that is considered in this work. RF is an ensemble technique. Ensemble techniques rely on combining different classifiers using some aggregation technique, such as majority voting. As pointed out earlier, ensemble techniques usually have good properties for combating overfitting. In this case, the aggregation of classifiers using a voting technique reduces the variance of the final classifier. This increases the robustness of the classifier and usually achieves a very good classification performance. A critical issue in the ensemble of classifiers is that for the combination to be successful, the errors made by the members of the ensemble should be as uncorrelated as possible. This is sometimes referred to in the

literature as the diversity of the classifiers. As the name suggests, the base classifiers in RF are decision trees.

5.7.3.1 A Brief Note on Decision Trees

A decision tree is one of the most simple and intuitive techniques in machine learning, based on the divide and conquer paradigm. The basic idea behind decision trees is to partition the space into patches and to fit a model to a patch. There are two questions to answer in order to implement this solution:

- How do we partition the space?
- What model shall we use for each patch?

Tackling the first question leads to different strategies for creating decision tree. However, most techniques share the axis-orthogonal hyperplane partition policy, i.e., a threshold in a single feature. For example, in our problem "Does the applicant have a home mortgage?". This is the key that allows the results of this method to be interpreted. In decision trees, the second question is straightforward, each patch is given the value of a label, e.g., the majority label, and all data falling in that part of the space will be predicted as such.

The RF technique creates different trees over the same training dataset. The word "random" in RF refers to the fact that only a subset of features is available to each of the trees in its building process. The two most important parameters in RF are the number of trees in the ensemble and the number of features each tree is allowed to check.

5.8 Ending the Learning Process

With both techniques in mind, we are going to optimize and check the results using nested cross-validation. Scikit-learn allows us to do this easily using several model selection techniques. We will use a grid search, `GridSearchCV` (a cross-validation using an exhaustive search over all combinations of parameters provided).

In [16]:
```
parameters = {'C': [1e4, 1e5, 1e6],
              'gamma': [1e-5, 1e-4, 1e-3]}
N_folds = 5

kf=cross_validation.KFold(n = y.shape[0],
                          n_folds = N_folds,
                          shuffle = True,
                          random_state = 0)

acc = np.zeros((N_folds,))
i = 0
# We will build the predicted y from the partial predictions
     on the test of each of the folds
yhat = y.copy()
for train_index, test_index in kf:
    X_train, X_test = X[train_index,:], X[test_index,:]
    y_train, y_test = y[train_index], y[test_index]
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    clf = svm.SVC(kernel = 'rbf')
    clf = grid_search.GridSearchCV(clf, parameters, cv = 3)
    clf.fit(X_train, y_train.ravel())
    X_test = scaler.transform(X_test)
    yhat[test_index] = clf.predict(X_test)

print metrics.accuracy_score(yhat, y)
print metrics.confusion_matrix(yhat, y)
```

Out[16]:
```
classification accuracy: 0.856038647343
confusion matrix:
3371 590
   6   173
```

The result obtained has a large error in the non-fully funded class (negative). This is because the default scoring for cross-validation grid-search is mean accuracy. Depending on our business, this large error in recall for this class may be unacceptable. There are different strategies for diminishing the impact of this effect. On the one hand, we may change the default scoring and find the parameter setting that corresponds to the maximum average recall. On the other hand, we could mitigate this effect by imposing a different weight on an error on the critical class. For example, we could look for the best parameterization such than one error on the critical class is equivalent to one thousand errors on the noncritical class. This is important in business scenarios where monetization of errors can be derived.

5.9 A Toy Business Case

Consider that clients using our service yield a profit of 100 units per client (we will use abstract units but keep in mind that this will usually be accounted in euros/dollars). We design a campaign with the goal of attracting investors in order to cover all non-fully funded loans. Let us assume that the cost of the campaign is α units per client. With this policy we expect to keep our customers satisfied and engaged with our service, so they keep using it. Analyzing the confusion matrix we can

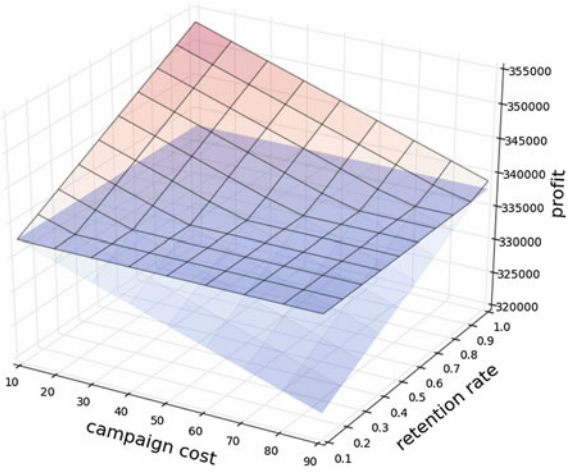


Fig. 5.10 Surfaces for two different campaign and attraction factors. The horizontal plane corresponds to the profit if no campaign is launched. The slanted plane is the profit for a certain confusion matrix

give precise meaning to different concepts in this campaign. The real positive set $(TP + FN)$ consists of the number of clients that are fully funded. According to our assumption, each of these clients generates a profit of 100 units. The total profit is $100 \cdot (TP + FN)$. The campaign to attract investors will be cast considering all the clients we predict are not fully funded. These are those that the classifier predict as negative, i.e., $(FN + TN)$. However, the campaign will only have an effect on the investors/clients that are actually not funded, i.e., TN; and we expect to attract a certain fraction β of them. After deploying our campaign, a simplified model of the expected profit is as follows:

$$100 \cdot (TP + FN) - \alpha(TN + FN) + 100\beta TN$$

When optimizing the classifier for accuracy, we do not consider the business needs. In this case, optimizing an SVM using cross-validation for different parameters of the C and γ, we have an accuracy of 85.60% and a confusion matrix with the following values:

$$\begin{pmatrix} 3371. & 590. \\ 6. & 173. \end{pmatrix}$$

If we check how the profit changes for different values of α and β, we obtain the plot in Fig. 5.10. The figure shows two hyperplanes. The horizontal plane is the expected profit if the campaign is not launched, i.e., $100 \cdot (TP + FN)$. The other hyperplane represents the profit of the campaign for different values of α and β using a particular classifier. Remember that the cost of the campaign is given by α, and the success rate of the campaign is represented by β. For the campaign to be successful we would like to select values for both parameters so that the profit of the campaign is larger than the cost of launching it. Observe in the figure that certain costs and attraction rates result in losses.

We may launch different classifiers with different configurations and toy with different weights (2, 4, 8, 16) for elements of different classes in order to bias the classi-

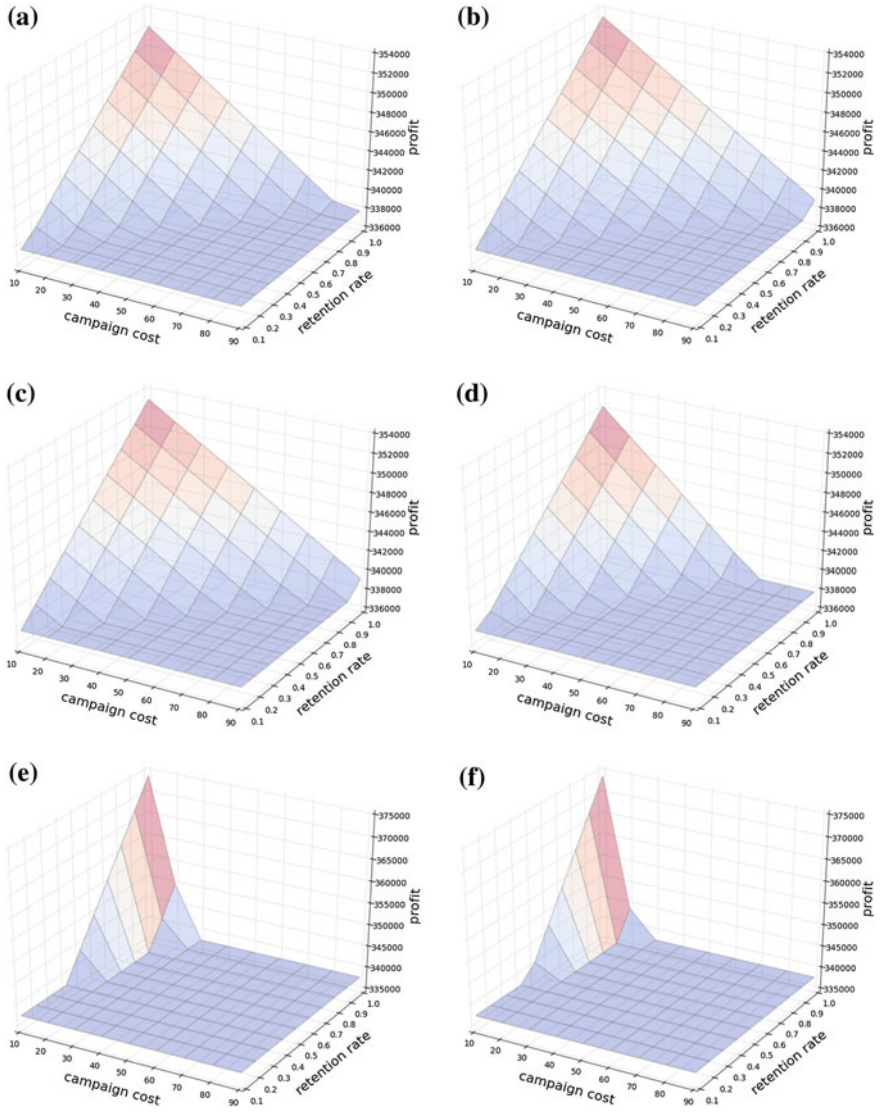


Fig. 5.11 3D surfaces of the profit obtained for different classifiers and configurations of retention campaign cost and retention rate. **a** RF, **b** SVM with the same cost per class, **c** SVM with double cost for the target class, **d** SVM with a cost for the target class equal to 4, **e** SVM with a cost for the target class equal to 8, **f** SVM with a cost for the target class equal to 16

fier towards obtaining different values for the confusion matrix.[13] The weights define

[13]It is worth mentioning that another useful tool for visualizing the trade-off between true positives and false positives in order to choose the operating point of the classifier is the receiver-operating

Table 5.1 Different configurations of classifiers and their respective profit rates and accuracies

	Max profit rate (%)	Profit rate at 60% (%)	Accuracy (%)
Random forest	4.41	2.41	87.87
SVM {1 : 1}	4.59	2.54	85.60
SVM {1 : 2}	4.52	2.50	85.60
SVM {1 : 4}	4.30	2.28	83.81
SVM {1 : 8}	10.69	3.57	52.51
SVM {1 : 16}	10.68	2.88	41.40

how much a misclassification in one class counts with respect to a misclassification in another. Figure 5.11 shows the different landscapes for different configurations of the SVM classifier and RF.

In order to frame the problem, we consider a very successful campaign with a 60% investor attraction rate. We can ask several questions in this scenario:

- What is the maximum amount to be spent on the campaign?
- How much will I gain?
- From all possible configurations of the classifier, which is the most profitable?
- Is it the one with the best accuracy?

Checking the values in Fig. 5.11, we find the results collected in Table 5.1. Observe that the most profitable campaign with 60% corresponds to a classifier that considers the cost of mistaking a sample from the non-fully funded class eight times larger than the one from the other class. Observe also that the accuracy in that case is much worse than in other configurations.

The take-home idea of this section is that business needs are often not aligned with the notion of accuracy. In such scenarios, the confusion matrix values have specific meanings. This must be taken into account when tuning the classifier.

5.10 Conclusion

In this chapter we have seen the basics of machine learning and how to apply learning theory in a practical case using Python. The example in this chapter is a basic one in which we can safely assume the data are independent and identically distributed, and that they can be readily represented in vector form. However, machine learning

(Footnote 13 continued)
characteristic (ROC) curve. This curve plots the true positive rate/sensitivity/recall (TP/(TP+FN)) with respect to the false positive rate (FP/(FP+TN)).

may tackle many more different settings. For example, we may have different target labels for a single example; this is called multilabel learning. Or, data can come from streams or be time dependent; in these settings, sequential learning or sequence learning can be the methods of choice. Moreover, each data example can be a non-vector or have a variable size, such as a graph, a tree, or a string. In such scenarios kernel learning or structural learning may be used. During these last years we are also seeing the revival of neural networks under the name of deep learning and achieving impressive results in different domains such as computer vision or natural language processing. Nonetheless, all of these methods will behave as explained in this chapter and most of the lessons learned here can be readily applied to these techniques.

Acknowledgements This chapter was co-written by Oriol Pujol and Petia Radeva.

Reference

1. M. Fernández-Delgado, E. Cernadas, S. Barro, D. Amorim, Do we Need Hundreds of Classifiers to Solve Real World Classification Problems? Journal of Machine Learning Research **15**, 3133 (2014). http://jmlr.org/papers/v15/delgado14a.html

Regression Analysis

6

6.1 Introduction

In this chapter, we introduce *regression analysis* and some of its applications in data science. Regression is related to how to make predictions about real-world quantities such as, for instance, the predictions alluded to in the following questions. How does sales volume change with changes in price? How is sales volume affected by the weather? How does the title of a book affect its sales? How does the amount of a drug absorbed vary with the patient's body weight; and does this relationship depend on blood pressure? How many customers can I expect today? At what time should I go home to avoid traffic jams? What is the chance of rain on the next two Mondays; and what is the expected temperature?

All these questions have a common structure: they ask for a *response* that can be expressed as a combination of one or more (independent) *variables* (also called *covariates* or *predictors*). The role of regression is to build a model to predict the response from the variables. This process involves the transition *from data to model*.

More specifically, the model can be useful in different tasks, such as the following: (1) analyzing the behavior of data (the relation between the response and the variables), (2) predicting data values (whether continuous or discrete), and (3) finding important variables for the model.

In order to understand how a regression model can be suitable for tackling these tasks, we will introduce three practical cases for which we use three real datasets and solve different questions. These practical cases will motivate *simple linear regression*, *multiple linear regression*, and *logistic regression*, as presented in the following sections.

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_6

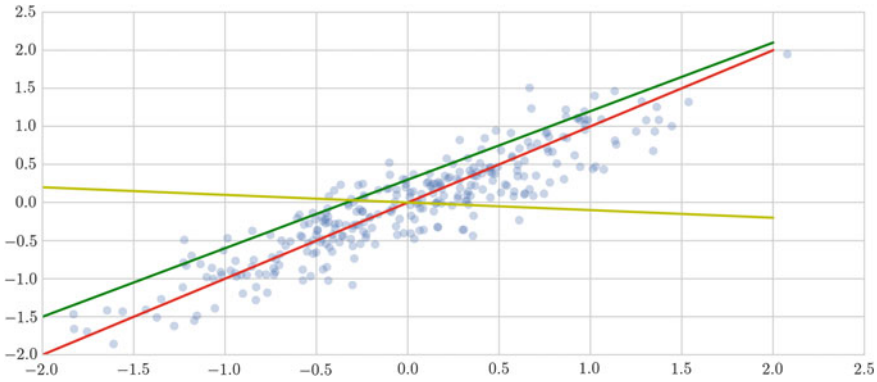


Fig. 6.1 Illustration of different simple linear regression models. *Blue points* correspond to a set of random points sampled from a univariate normal (Gaussian) distribution. *Red*, *green* and *yellow lines* are three different simple linear regression models

6.2 Linear Regression

The objective of performing a regression is to build a model to express the relation between the response $\mathbf{y} \in \mathbb{R}^n$ and a combination of one or more (independent) variables $\mathbf{x}_i \in \mathbb{R}^n$. [1] The model allows us to predict the response $\mathbf{y}$ from the variables. The simplest model which can be considered is a *linear model*, where the response $\mathbf{y}$ depends linearly on the d variables $\mathbf{x}_i$:

$$\mathbf{y} = a_1\mathbf{x}_1 + \cdots + a_d\mathbf{x}_d. \tag{6.1}$$

The variables a_i are termed the *parameters* or *coefficients* of the model. This equation can be rewritten in a more compact matrix form: $\mathbf{y} = \mathbf{X}\mathbf{w}$, where

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \mathbf{X} = \begin{pmatrix} x_{11} \ldots x_{1d} \\ x_{21} \ldots x_{2d} \\ \vdots \\ x_{n1} \ldots x_{nd} \end{pmatrix}, \mathbf{w} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_d \end{pmatrix}.$$

Linear regression is the technique for creating these linear models.

6.2.1 Simple Linear Regression

Simple linear regression considers n samples of a single variable $\mathbf{x} \in \mathbb{R}^n$ and describes the relationship between the variable and the response with the model:

$$\mathbf{y} = a_0 + a_1\mathbf{x}, \tag{6.2}$$

where the parameter a_0 is called the *intercept* or the *constant term*.

Given a set of samples $(\mathbf{x}, \mathbf{y})$, such as the set illustrated in Fig. 6.1, we can create a linear model to explain the data, as in Eq. (6.2). But how do we know which is the

best model (best parameters) for this particular set of samples? See the three different models (straight lines in different colors) in Fig. 6.1.

Ordinary least squares (OLS) is the simplest and most common *estimator* in which the parameters (a's) are chosen to minimize the square of the distance between the predicted values and the actual values with respect to a_0, a_1:

$$||a_0 + a_1\mathbf{x} - \mathbf{y}||_2^2 = \sum_{j=1}^{n}(a_0 + a_1 x_j - y_j)^2.$$

We are concerned here with the y-axis distance, since it does not consider the error in the variables. This error expression is often called the *sum of squared errors of prediction* (SSE). The SSE function is quadratic in the parameters, $\mathbf{w}$, with positive-definite Hessian, and therefore this function possesses a unique global minimum at $\hat{\mathbf{w}} = (\hat{a}_0, \hat{a}_1)$. The resulting model is represented as follows: $\hat{\mathbf{y}} = \hat{a}_0 + \hat{a}_1\mathbf{x}$, where the hats on the variables represent the fact that they are estimated from the data available.

OLS is a popular approach for several reasons. It makes it computationally cheap to calculate the coefficients. It is also easier to interpret than the other more sophisticated models. In situations where the goal is to understand a simple model in detail, rather than to estimate the response well, it can provide insight into what the model captures. Finally, in situations where there is a lot of noise, as in many real scenarios, it may be hard to find the true functional form, so a constrained model can perform quite well compared to a complex model which can be more affected by noise.

Practical Case: Sea Ice Data and Climate Change

In this practical case, we pose the question: Is the climate really changing? More concretely, we want to show the effect of the climate change by determining whether the *sea ice area* (or *extent*) has decreased over the years. Sea ice area refers to the total area covered by ice, whereas sea ice extent is the area of ocean with at least 15% sea ice. Reliable measurement of sea ice edges began with the satellite era in the late 1970s. Before then, sea ice area and extent were monitored less precisely by a combination of ships, buoys, and aircraft.

We will use the sea ice data from the National Snow & Ice Data Center[1] which provides measurements of the area and extend of sea ice at the poles over the last 36 years. The center has given access to the archived monthly Sea Ice Index images and data since 1979 [2]. The archived data reside at an FTP location[2] (web-page instructions can be followed easily to access and download the files). The ASCII data files tabulate sea ice extent and area (in millions of square kilometers) by year for a given month.

In order to check whether there is an anomaly in the evolution of sea ice extent over recent years, we want to build a simple linear regression model and analyze the fitting; but before we need to perform several processing steps.

[1] https://nsidc.org/data/seaice_index/archives.html.
[2] ftp://sidads.colorado.edu/DATASETS/NOAA/G02135/.

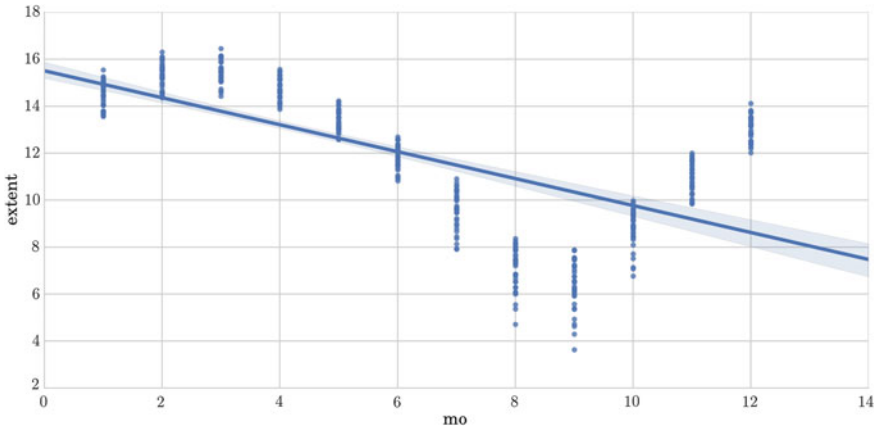


Fig. 6.2 Ice extent data by month

First, we read the data, previously downloaded, and create a `DataFrame` (*Pandas*) as follows:

In [1]:
```
ice = pd.read_csv('files/ch06/SeaIce.txt',
                  delim_whitespace=True)
print 'shape:', ice.shape
```

Out[1]:
```
shape: (424, 6)
```

For data cleaning, we check the values of all the fields to detect any potential error. We find that there is a '−9999' value in the `data_type` field which should contain 'Goddard' or 'NRTSI-G' (the type of the input dataset). So we can easily clean the data, removing these instances.

In [2]:
```
ice2 = ice[ice.data_type != '-9999']
```

Next, we visualize the data. The `lmplot()` function from the *Seaborn* toolbox is intended for exploring linear relationships of different forms in multidimensional datasets. For instance, we can illustrate the relationship between the `month` of the year (variable) and the `extent` (response) as follows:

In [3]:
```
import Seaborn as sns
sns.lmplot("mo", "extent", ice2)
```

This outputs Fig. 6.2. We can observe a monthly fluctuation of the sea ice extent, as would be expected for the different seasons of the year.

We should normalize the data before performing the regression analysis to avoid this fluctuation and be able to study the evolution of the extent over the years. To capture the variation for a given interval of time (month), we can compute the mean

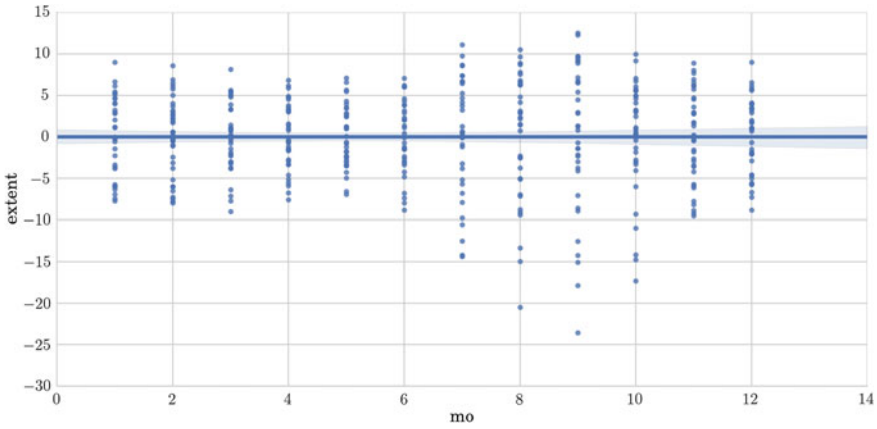


Fig. 6.3 Ice extent data by month after the normalization

for the i-th interval of time (using the period from 1979 through 2014 for the mean extent) μ_i, and subtract it from the set of extent values for that month $\{e^i_j\}$. This value can be converted to a relative percentage difference by dividing it by the total average (1979–2014) μ, and then multiplying by 100:

$$\tilde{e}^i_j = 100 * \frac{e^i_j - \mu_i}{\mu}, i = 1, \ldots, 12.$$

We implement this normalization and plot the relationship again as follows:

In [4]:
```
for i in range(12):
    ice2.extent[ice2.mo == i+1] =
        100*(ice2.extent[ice2.mo == i+1]
            - month_means[i+1])
        /month_means.mean()
sns.lmplot("mo", "extent", ice2)
```

The new output is in Fig. 6.3. We now observe a comparable range of values for all months.

Next, the normalized values can be plotted for the entire time series to analyze the tendency. We compute the trend as a simple linear regression. We use the `lmplot()` function for visualizing linear relationships between the year (variable) and the extent (response).

In [5]:
```
sns.lmplot("year", "extent", ice2)
```

This outputs Fig. 6.4 showing the regression model fitting the extent data. This plot has two main components. The first is a scatter plot, showing the observed data points. The second is a regression line, showing the estimated linear model relating

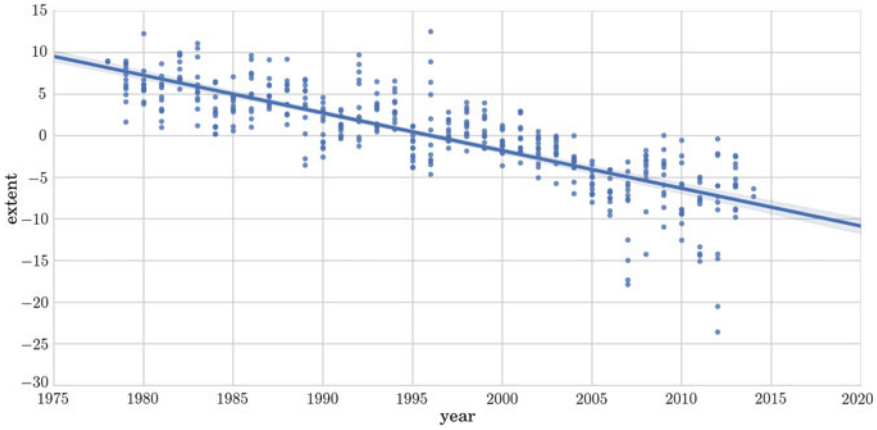


Fig. 6.4 Regression model fitting sea ice extent data for all months by year using `lmplot`

the two variables. The regression line is plotted with a 95% confidence band to give an impression of the uncertainty in the model.

In this figure, we can observe that the data show a long-term negative trend over years. The negative trend can be attributed to global warming, although there is also a considerable amount of variation from year to year.

Up until here, we have qualitatively shown the linear regression using a useful visualization tool. We can also analyze the linear relationship in the data using the *Scikit-learn* library, which allows a quantitative evaluation. As was explained in the previous chapter, Scikit-learn provides an object-oriented interface centered around the concept of an estimator. The `sklearn.linear_model.LinearRegression` estimator sets the state of the estimator based on the training data using the function `fit`. Moreover, it allows the user to specify whether to fit an intercept term in the object construction. This is done by setting the corresponding constructor arguments of the estimator object as follows:

In [6]:
```
from sklearn.linear_model import LinearRegression
est = LinearRegression(fit_intercept = True)
```

During the fitting process, the state of the estimator is stored in instance attributes that have a trailing underscore (`'_'`). For example, the coefficients of a `LinearRegression` estimator are stored in the attribute `coef_`. We fit a regression model using years as variables (**x**) and the extent values as the response (**y**).

In [7]:
```
x = ice2[['year']]
y = ice2[['extent']]
est.fit(x, y)
print "Coefficients:", est.coef_
print "Intercept:", est.intercept_
```

Out[7]:
```
Coefficients: [[-0.45275459]]
Intercept: [ 903.71640207]
```

Estimators that can generate predictions provide an `Estimator.predict` method. In the case of regression, `Estimator.predict` will return the predicted regression values. We can evaluate the model fitting by computing the mean squared error (MSE) and the coefficient of determination (R^2) of the model. The coefficient R^2 is defined as $(1 - \mathbf{u}/\mathbf{v})$, with $\mathbf{u} = \sum(\mathbf{y} - \hat{\mathbf{y}})^2$ and $\mathbf{v} = \sum(\mathbf{y} - \bar{\mathbf{y}})^2$, where $\bar{\mathbf{y}}$ is the mean. The best possible score for R^2 is 1.0, lower values are worse (it can also be negative). These measures can provide a quantitative answer to the question we are facing: Is there a negative trend in the evolution of sea ice extent over recent years? We can perform this analysis for a particular month or for all months together, as done in the following lines:

In [8]:
```
from sklearn import metrics
y_hat = est.predict(x)
print "MSE:", metrics.mean_squared_error(y_hat, y)
print "R^2:", metrics.r2_score(y_hat, y)
print 'var:', y.var()
```

Out[8]:
```
MSE: 10.5391316398
R^2: 0.50678703821
var: 31.98324
```

The negative trend seen in Fig. 6.4 is validated by the MSE value which is small, 0.1%, and the R^2 value which is acceptable, given the variance of the data, 0.3%.

Given the model, we can also predict the extent value for the coming years. For instance, the predicted extent for January 2025 can be computed as follows:

In [9]:
```
x = [2025]
y_hat = model.predict(x)
m = 1 # January
y_hat = (y_hat*month_means.mean()/100) + month_means[m]
print "Prediction of extent for January 2025
      (in millions of square km):", y_hat
```

Out[9]:
```
Prediction of extent for January 2025 (in millions of square
km): [12.93603933].
```

6.2.2 Multiple Linear Regression and Polynomial Regression

As we have seen in the previous section, with simple linear regression we describe the relationship between the variable and the response with a straight line. In the case of *multiple linear regression*, we extend this idea by fitting a d-dimensional hyperplane to our d variables, as defined in Eq. (6.1).

Multiple linear regression may seem a very simple model, but even when the response depends on the variables in nonlinear ways, this model can still be used by

considering nonlinear transformations $\phi(\cdot)$ of the variables:

$$\mathbf{y} = a_1\phi(\mathbf{x}_1) + \cdots + a_d\phi(\mathbf{x}_d)$$

This model is called *polynomial regression* and it is a popular nonlinear regression technique which models the relationship between the response and the variables as an p-th order polynomial. The higher the order of the polynomial, the more complex the functions you can fit. However, using higher-order polynomial can involve *computational complexity* and *overfitting*. Overfitting occurs when a model fits the characteristics of the training data and loses the capacity to generalize from the seen to predict the unseen.

6.2.3 Sparse Model

Often, in real problems, there are uninformative variables in the data which prevent proper modeling of the problem and thus, the building of a correct regression model. In such cases, a feature selection process is crucial to select only the informative features and discard non-informative ones. This can be achieved by *sparse methods* which use a penalization approach, such as *LASSO* (least absolute shrinkage and selection operator) to set some model coefficients to zero (thereby discarding those variables). Sparsity can be seen as an application of Occam's razor: prefer simpler models to complex ones.

Given the set of samples $(\mathbf{X}, \mathbf{y})$, the objective of a sparse model is to minimize the SSE through a restriction (or penalty):

$$\frac{1}{2n}||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2 + \alpha||\mathbf{w}||_1,$$

where $||\mathbf{w}||_1$ is the $L1$-norm of the parameter vector $\mathbf{w} = (a_0, \ldots, a_d)$.

Practical Case: Prediction of the Price of a New Housing Market

In this practical case we want to solve the question: Can we predict the price of a new market given any of its attributes?

We will use the Boston housing dataset from Scikit-learn, which provides recorded measurements of 13 attributes of housing markets around Boston, as well as the median house price.[3] Once we load the dataset (506 instances), the description of the dataset can easily be shown by printing the field `DESCR`. The data ($\mathbf{x}$), feature names, and target ($\mathbf{y}$) are stored in other fields of the dataset.

We first consider the task of predicting median house values in the Boston area using as the variable one of the attributes, for instance, `LSTAT`, defined as the "proportion of lower status of the population".

Seaborn visualization can be used to show this linear relationships easily:

[3]Copy of UCI ML housing dataset: http://archive.ics.uci.edu/ml/datasets/Housing.

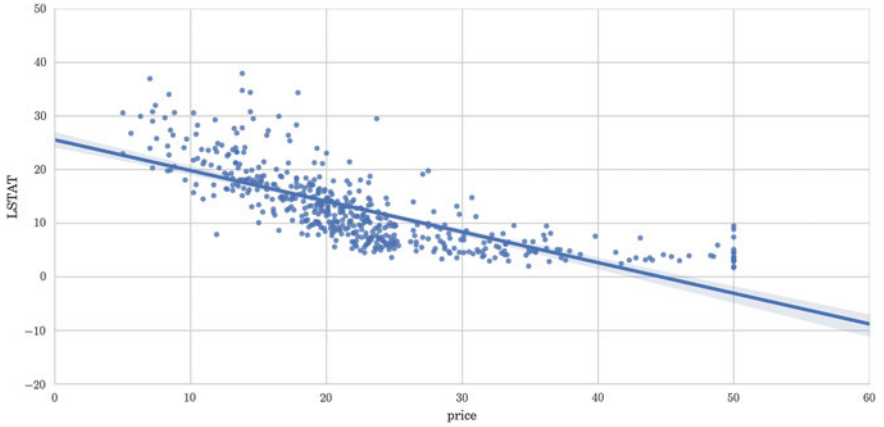


Fig. 6.5 Scatter plot of Boston data (`LSTAT` versus `price`) and their linear relationship (using `lmplot`)

In [10]:
```
from sklearn import datasets
boston = datasets.load_boston()
X_boston, y_boston = boston.data, boston.target
print 'Shape of data:', X_boston.shape, y_boston.shape
print 'Feature names:',boston.feature_names
df_boston = pd.DataFrame(boston.data,
                         columns = boston.feature_names)
df_boston['price'] = boston.target
sns.lmplot("price", "LSTAT", df_boston)
```

Out[10]:
```
Shape of data: (506L, 13L) (506L,)
Feature names: ['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE'
'DIS' 'RAD' 'TAX' 'PTRATIO' 'B' 'LSTAT']
```

In Fig. 6.5, we can clearly see that the relationship between `price` and `LSTAT` is nonlinear, since the straight line is a poor fit. We can examine whether a better fit can be obtained by including higher-order terms. For example, a quadratic model:

$$\mathbf{y}_i \approx a_0 + a_1 \mathbf{x}_i + a_2 \mathbf{x}_i^2$$

The `lmplot` function allows to easily change the order of the model as is done in the next code, which outputs Fig. 6.6, where we observe a better fit.

In [11]:
```
sns.lmplot("price", "LSTAT", df_boston, order = 2)
```

To study the relation among multiple variables in a dataset, there are different options. We can study the relationship between several variables in a dataset by using the functions `corr` and `heatmap` which allow to calculate a correlation matrix for a dataset and draws a *heat map* with the correlation values. The heat map is a matricial image which helps to interpret the correlations among variables. For the sake of visualization, we do not consider all the 13 variables in the Boston housing data, but six: `CRIM`, per capita crime rate by town; `INDUS`, proportion of non-retail

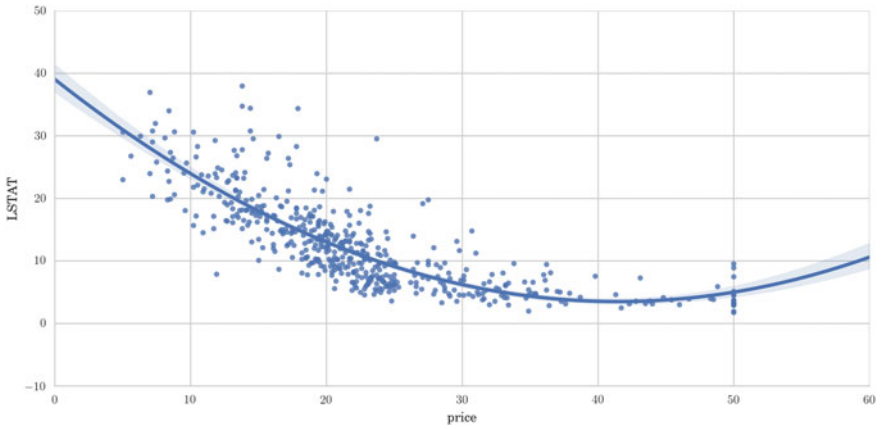


Fig. 6.6 Scatter plot of Boston data (`LSTAT` versus `price`) and their polynomial relationship (using `lmplot` with order 2)

business acres per town; `NOX`, nitric oxide concentrations (parts per 10 million); `RM`, average number of rooms per dwelling; `AGE`, proportion of owner-occupied units built prior to 1940; and `LSTAT`. These variables are indicated by their indexes in the following code:

In [12]:
```
indexes = [0,2,4,5,6,12]
df2 = pd.DataFrame(boston.data[:,indexes],
                   columns = boston.feature_names[indexes])
df2['price'] = boston.target
corrmat = df2.corr()
sns.heatmap(corrmat, vmax = .8, square = True)
```

Figure 6.7 shows a heat map representing the correlation between pairs of variables; specifically, the six variables selected and the price of houses. The color bar shows the range of values used in the matrix. This plot is a useful way of summarizing the correlation of several variables. It can be seen that `LSTAT` and `RM` are the variables that are most correlated with `price`.

Another good way to explore multiple variables is the *scatter plot* from Pandas. The scatter plot is a grid of plots of multiple variables one against the others, illustrating the relationship of each variable with the rest. For the sake of visualization, we do not consider all the variables, but just three: `RM`, `AGE`, and `LSTAT` defined by `indexes` in the following code:

In [13]:
```
indexes=[5,6,12]
df2 = pd.DataFrame(boston.data[:,indexes],
                   columns = boston.feature_names[indexes])
df2['price'] = boston.target
pd.scatter_matrix(df2, figsize = (12.0, 12.0))
```

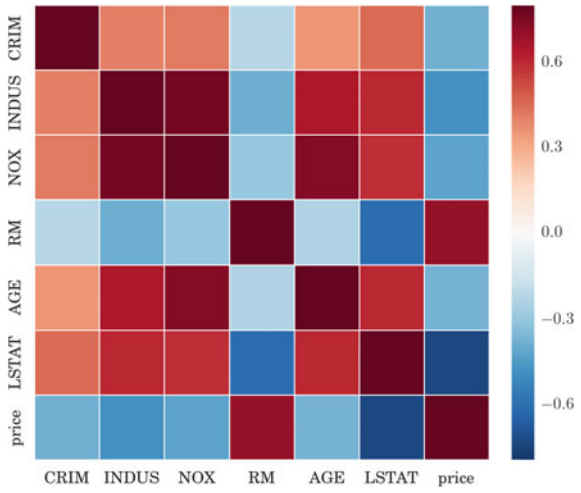


Fig. 6.7 Correlation plot: heat map representing the correlation between seven pairs of variables in the Boston housing dataset

This code outputs Fig. 6.8, where we obtain visual information concerning the density function for every variable, in the diagonal, as well as the scatter plots of the data points for pairs of variables. In the last column, we can appreciate the relation between the three variables selected and house prices. It can be seen that `RM` follows a linear relation with `price`; whereas `AGE` does not. `LSTAT` follows a higher-order relation with `price`. This plot gives us an indication of how good or bad every attribute would be as a variable in a linear model.

For the evaluation of the prediction power of the model with new samples, we split the data into a training set and a testing set, and we compute the linear regression score, which returns the coefficient of determination R^2 of the prediction. We can also calculate the MSE.

In [14]:

```
from sklearn import linear_model
train_size = X_boston.shape[0]/2
X_train = X_boston[:train_size]
X_test = X_boston[train_size:]
y_train = y_boston[:train_size]
y_test = y_boston[train_size:]
print 'Training and testing set sizes',
       X_train.shape, X_test.shape
regr = LinearRegression()
regr.fit(X_train, y_train)
print 'Coeff and intercept:',
      regr.coef_, regr.intercept_
print 'Testing Score:', regr.score(X_test, y_test) print '
    Training
MSE: ',
      np.mean((regr.predict(X_train) - y_train)**2)
print 'Testing MSE: ',
      np.mean((regr.predict(X_test) - y_test)**2)
```

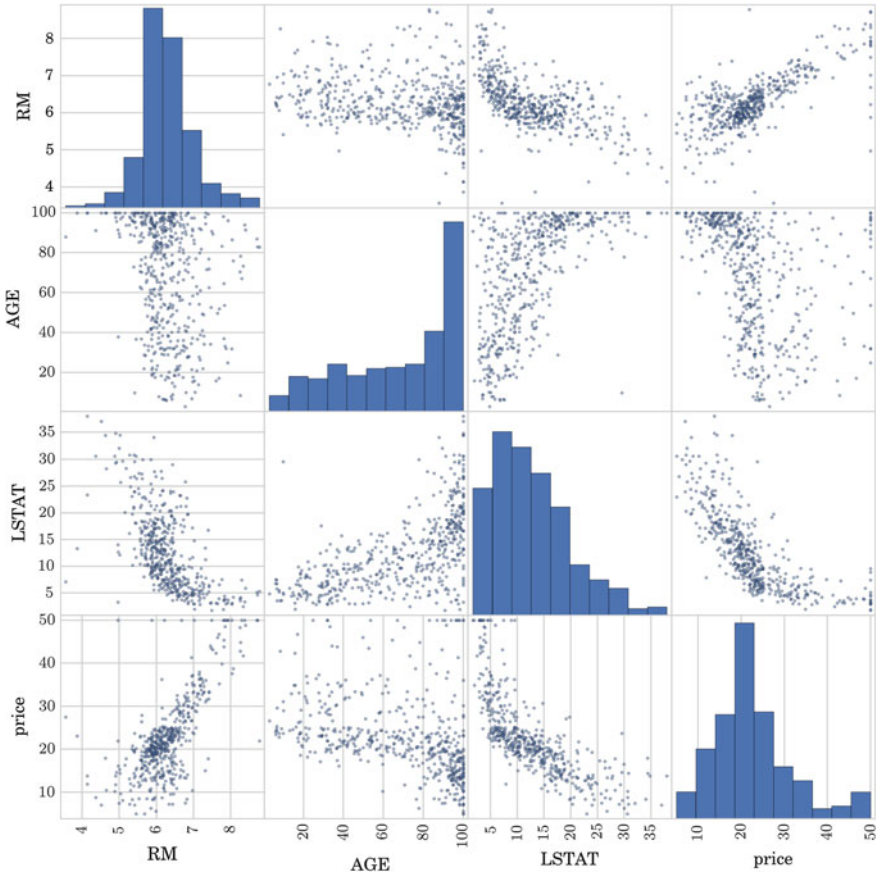


Fig. 6.8 Scatter plot of Boston housing dataset

```
Out[14]: Training and testing set sizes (253, 13) (253, 13)
Coeff and intercept: [ 1.20133313 0.02449686 0.00999508
0.42548672 -8.44272332 8.87767164 -0.04850422 -1.11980855
0.20377571 -0.01597724 -0.65974775 0.01777057 -0.11480104]
-10.0174305829
Testing Score: -2.24420202674
Training MSE: 9.98751732546
Testing MSE: 302.64091133
```

We can see that all the coefficients obtained are different from zero, meaning that no variable is discarded. Next, we try to build a sparse model to predict the price using the most important factors and discarding the non-informative ones. To do this, we can create a LASSO regressor, forcing zero coefficients.

In [15]:
```
regr_lasso = linear_model.Lasso(alpha = .3)
regr_lasso.fit(X_train, y_train) print 'Coeff and intercept:
    ',regr_lasso.coef_
print 'Tesing Score:', regr_lasso.score(X_test,
y_test) print 'Training MSE: ',
      np.mean((regr_lasso.predict(X_train)  - y_train)**2)
print 'Testing MSE: ',
      np.mean((regr_lasso.predict(X_test) - y_test)**2)
```

Out[15]:
```
Coeff and intercept: [ 0. 0.01996512 -0. 0. -0. 7.69894744
-0.03444803 -0.79380636 0.0735163 -0.0143421 -0.66768539
0.01547437 -0.22181817] -6.18324183615
Testing Score: 0.501127529021
Training MSE: 10.7343110095
Testing MSE: 46.5381680949
```

It can now be seen that the result of the model fitting for a set of sparse coefficients is much better than before (using all the variables), with the score increasing from -2.24 to 0.5. This demonstrates that four of the initial variables are not important for the prediction and in fact they confuse the regressor.

With the LASSO result, we can also emphasize the most important factors for determining the price of a new market, based on the coefficient values:

In [16]:
```
ind = np.argsort(np.abs(regr_lasso.coef_))
print 'Ordered variable (from less to more important):',
      boston.feature_names[ind]
```

Out[16]:
```
Ordered variable (from less to more important): ['CRIM' 'INDUS'
'CHAS' 'NOX' 'TAX' 'B' 'ZN' 'AGE' 'RAD' 'LSTAT' 'PTRATIO' 'DIS'
'RM']
```

There are also other strategies for feature selection. For instance, we can select the k = 5 best features, according to the k highest scores, using the function `SelectKBest` from Scikit-learn:

In [17]:
```
import sklearn.feature_selection as fs
selector = fs.SelectKBest(score_func = fs.f_regression,
                          k = 5)
selector.fit_transform(X_train, y_train) per
selector.fit(X_train,y_train)
print 'Selected features:',
      zip(selector.get_support(), boston.feature_names)
```

Out[17]:
```
Selected features: [(False, 'CRIM'), (False, 'ZN'), (True,
'INDUS'), (False, 'CHAS'), (False, 'NOX'), (True, 'RM'), (True,
'AGE'), (False, 'DIS'), (False, 'RAD'), (False, 'TAX'), (True,
'PTRATIO'), (False, 'B'), (True, 'LSTAT')]
```

The set of selected features is now different, since the criterion has changed. However, three of the most important features: `RM`, `PTRATIO`, and `LSTAT`.

In order to evaluate the prediction, it could be interesting to visualize the target and predicted responses in a scatter plot, as it is done in the next code:

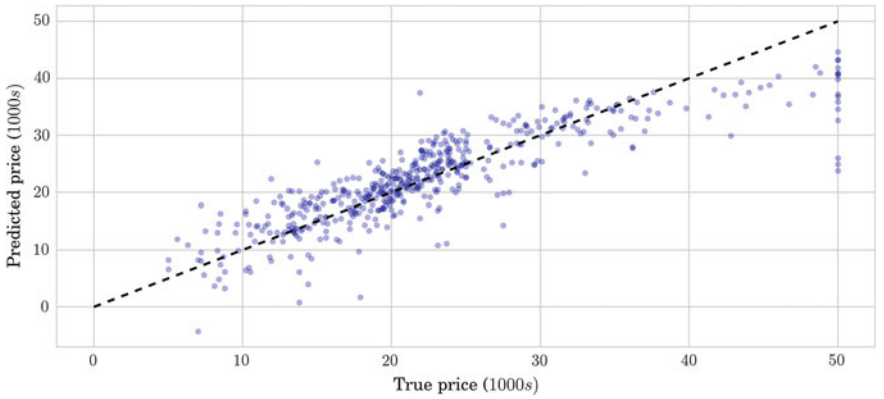


Fig. 6.9 Relation between true (x-axis) and predicted (y-axis) prices

In [18]:
```
clf = LinearRegression()
clf.fit(boston.data, boston.target)
predicted = clf.predict(boston.data)
plt.scatter(boston.target, predicted, alpha = 0.3)
plt.plot([0, 50], [0, 50], '--k')
plt.axis('tight')
plt.xlabel('True price ($1000s)')
plt.ylabel('Predicted price ($1000s)')
```

The output is shown in Fig. 6.9, where we can observe that the original prices are properly estimated by the predicted ones, except for the higher values, around $50.000 (points in the top right corner).

Finally, it is worth noting that we can work with statistical evaluation of a linear regression with the *OLS* toolbox of the *Stats Model* toolbox.[4] This toolbox is useful to study several statistics concerning the regression model. To know more about the toolbox, go to the Documentation related to Stats Models.

6.3 Logistic Regression

Logistic regression is a type of model of probabilistic statistical classification. It is used as a binary model to predict a binary response, the outcome of a categorical dependent variable (i.e., a class label), based on one or more variables.

The form of the logistic function is:

$$f(x) = \frac{1}{1 + e^{-\lambda x}}$$

[4] http://statsmodels.sourceforge.net/devel/examples/notebooks/generated/ols.html.

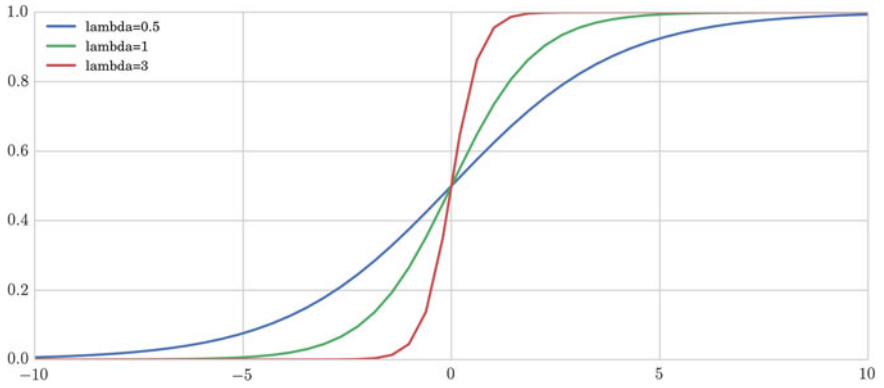


Fig. 6.10 Logistic function for different lambda values

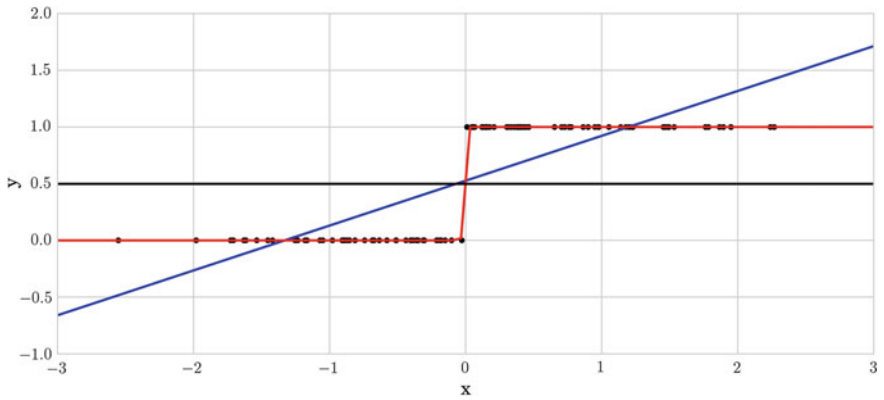


Fig. 6.11 Linear regression (*blue*) versus logistic regression (*red*) for fitting a set of data (*black points*) normally distributed across the 0 and 1 y-values

Figure 6.10 illustrates the logistic function with different values of λ. This function is useful because it can take as its input any value from negative infinity to positive infinity, whereas the output is restricted to values between 0 and 1 and hence can be interpreted as a probability.

The set of samples ($\mathbf{X}$, $\mathbf{y}$), illustrated as black points in Fig. 6.11, defines a fitting problem suitable for a logistic regression. The blue and red lines show the fitting result for linear and logistic models, respectively. In this case, a logistic model can clearly explain the data; whereas a linear model cannot.

Practical Case: Winning or Losing Football Team

Now, we pose the question: What number of goals makes a football team the winner or the loser? More concretely, we want to predict victory or defeat in a football match when we are given the number of goals a team scores. To do this we consider

the set of results of the football matches from the Spanish league[5] and we build a classification model with it.

We first read the data file in a `DataFrame` and select the following columns in a new `DataFrame`: `HomeTeam`, `AwayTeam`, `FTHG` (home team goals), `FTAG` (away team goals), and `FTR` (H=home win, D = draw, A = away win). We then build a d-dimensional vector of variables with all the scores, $\mathbf{x}$, and a binary response indicating victory or defeat, $\mathbf{y}$. For that, we create two extra columns containing `W` the number of goals of the winning team and `L` the number of goals of the losing team and we concatenate these data. Finally, we can compute and visualize a logistic regression model to predict the discrete value (victory or defeat) using these data.

In [19]:
```
from sklearn.linear_model import LogisticRegression
data = pd.read_csv('files/ch06/SP1.csv')
s = data[['HomeTeam','AwayTeam', 'FTHG', 'FTAG', 'FTR']]
def my_f1(row):
    return max(row['FTHG'], row['FTAG'])
def my_f2(row):
    return min(row['FTHG'], row['FTAG'])
s['W'] = s.apply(my_f1, axis = 1)
s['L'] = s.apply(my_f2, axis = 1)
x1 = s['W'].values
y1 = np.ones(len(x1), dtype = np.int)
x2 = s['L'].values
y2 = np.zeros(len(x2), dtype = np.int)
x = np.concatenate([x1, x2])
x = x[:, np.newaxis]
y = np.concatenate([y1, y2])
logreg = LogisticRegression()
logreg.fit(x, y)
X_test = np.linspace(-5, 10, 300)
def lr_model(x):
    return 1 / (1+np.exp(-x))
loss = lr_model(X_test*logreg.coef_ + logreg.intercept_)
       .ravel()
X_test2 = X_test[:,np.newaxis]
losspred = logreg.predict(X_test2)
plt.scatter(x.ravel(), y,
            color = 'black',
            s = 100, zorder = 20,
            alpha = 0.03)
plt.plot(X_test, loss, color = 'blue', linewidth = 3)
plt.plot(X_test, losspred, color = 'red', linewidth = 3)
```

Figure 6.12 shows a scatter plot with transparency so we can appreciate the overlapping in the discrete positions of the total numbers of victories and defeats. It also shows the fitting of the logistic regression model, in blue, and prediction of the logistic regression model, in red, for the Spanish football league results. With this information we can estimate that the cutoff value is 1. This means that a team, in general, has to score more than one goal to win.

[5] http://www.football-data.co.uk/mmz4281/1213/SP1.csv.

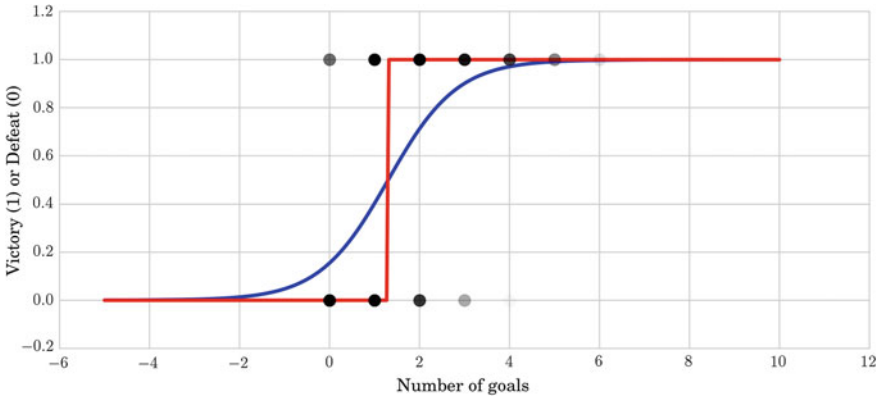


Fig. 6.12 Fitting of the logistic regression model (*blue*) and prediction of the logistic regression model (*red*) for the Spanish football league results

6.4 Conclusions

In this chapter, we have focused on regression analysis and the different Python tools that are useful for performing it. We have shown how regression analysis allows us to better understand data by means of building a model from it. We have formally presented four different regression models: simple linear regression, multiple linear regression, polynomial regression, and logistic regression. We have also emphasized the properties of sparse models in the selection of variables.

The different models have been used in three real problems dealing with different types of datasets. In these practical cases, we solve different questions regarding the behavior of the data, the prediction of data values (continuous or discrete), and the importance of variables for the model. In the first case, we showed that there is a decreasing tendency in the sea ice extent over the years, and we also predicted the amount of ice for the next 20 years. In the second case, we predicted the price of a market given a set of attributes and distinguished which of the attributes were more important in the prediction. Moreover, we presented a useful way to show the correlation between pairs of variables, as well as a way to plot the relationship between pairs of variables. In the third case, we faced the problem of predicting victory or defeat in a football match given the score of a team. We posed this problem as a classification problem and solved it using a logistic regression model; and we estimated the minimum number of goals a team has to score to win.

Acknowledgements This chapter was co-written by Laura Igual and Jordi Vitrià.

References

1. D. Freedman, *Statistical Models: Theory and Practice*. Cambridge University Press, (2009)
2. J. Maslanik, J. Stroeve. Near-Real-Time DMSP SSMIS Daily Polar Gridded Sea Ice Concentrations. Sea ice index data: Monthly sea ice extent and area data files, (1999, updated daily). http://dx.doi.org/10.5067/U8C09DWVX9LM

Unsupervised Learning

7

7.1 Introduction

In machine learning, the problem of unsupervised learning is that of trying to find hidden structure in unlabeled data. Since the examples given to the learner are unlabeled, there is no error or reward signal to evaluate the goodness of a potential solution. This distinguishes unsupervised from supervised learning. Unsupervised learning is defined as the task performed by algorithms that learn from a training set of unlabeled or unannotated examples, using the features of the inputs to categorize them according to some geometric or statistical criteria.

Unsupervised learning encompasses many techniques that seek to summarize and explain key features or structures of the data. Many methods employed in unsupervised learning are based on data mining methods used to preprocess data. Most unsupervised learning techniques can be summarized as those that tackle the following four groups of problems:

- *Clustering*: has as a goal to partition the set of examples into groups.
- *Dimensionality reduction*: aims to reduce the dimensionality of the data. Here, we encounter techniques such as Principal Component Analysis (PCA), independent component analysis, and nonnegative matrix factorization.
- *Outlier detection*: has as a purpose to find unusual events (e.g., a malfunction), that distinguish part of the data from the rest according to certain criteria.
- *Novelty detection*: deals with cases when changes occur in the data (e.g., in streaming data).

The most common unsupervised task is clustering, which we focus on in this chapter.

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_7

7.2 Clustering

Clustering is a process of grouping similar objects together; i.e., to partition unlabeled examples into disjoint subsets of clusters, such that:

- Examples within a cluster are similar (in this case, we speak of *high intraclass similarity*).
- Examples in different clusters are different (in this case, we speak of *low interclass similarity*).

When we denote data as similar and dissimilar, we should define a measure for this similarity/dissimilarity. Note that grouping similar data together can help in discovering new categories in an unsupervised manner, even when no sample category labels are provided. Moreover, two kinds of inputs can be used for grouping:

(a) in *similarity-based clustering*, the input to the algorithm is an $n \times n$ *dissimilarity matrix* or *distance matrix*;
(b) in *feature-based clustering*, the input to the algorithm is an $n \times D$ *feature matrix* or *design matrix*, where n is the number of examples in the dataset and D the dimensionality of each sample.

Similarity-based clustering allows easy inclusion of domain-specific similarity, while feature-based clustering has the advantage that it is applicable to potentially noisy data.

Therefore, several questions regarding the clustering process arise.

- What is a natural grouping among the objects? We need to define the "groupness" and the "similarity/distance" between data.
- How can we group samples? What are the best procedures? Are they efficient? Are they fast? Are they deterministic?
- How many clusters should we look for in the data? Shall we state this number a priori? Should the process be completely data driven or can the user guide the grouping process? How can we avoid "trivial" clusters? Should we allow final clustering results to have very large or very small clusters? Which methods work when the number of samples is large? Which methods work when the number of classes is large?
- What constitutes a good grouping? What objective measures can be defined to evaluate the quality of the clusters?

There is not always a single or optimal answer to these questions. It used to be said that clustering is a "subjective" issue. Clustering will help us to describe, analyze, and gain insight into the data, but the quality of the partition depends to a great extent on the application and the analyst.

7.2.1 Similarity and Distances

To speak of similar and dissimilar data, we need to introduce a notion of the similarity of data. There are several ways for modeling of similarity. A simple way to model this is by means of a Gaussian kernel:

$$s(a, b) = e^{-\gamma d(a,b)}$$

where $d(a, b)$ is a metric function, and γ is a constant that controls the decay of the function. Observe that when $a = b$, the similarity is maximum and equal to one. On the contrary, when a is very different to b, the similarity tends to zero. The former modeling of the similarity function suggests that we can use the notion of distance as a surrogate. The most widespread distance metric is the *Minkowski distance*:

$$d(a, b) = (\sum_{i=1}^{d} |a_i - b_i|^p)^{1/p}$$

where $d(a, b)$ stands for the distance between two elements $a, b \in \mathbb{R}^d$, d is the dimensionality of the data, and p is a parameter.

The best-known instantiations of this metric are as follows:

- when $p = 2$, we have the *Euclidean distance*,
- when $p = 1$, we have the *Manhattan distance*, and
- when $p = \inf$, we have the *max-distance*. In this case, the distance corresponds to the component $|a_i - b_i|$ with the highest value.

7.2.2 What Constitutes a Good Clustering? Defining Metrics to Measure Clustering Quality

When performing clustering, the question normally arises: How do we measure the quality of the clustering result? Note that in unsupervised clustering, we do not have groundtruth labels that would allow us to compute the accuracy of the algorithm. Still, there are several procedures for assessing quality. We find two families of techniques: those that allow us to compare clustering techniques, and those that check on specific properties of the clustering, for example "compactness".

7.2.2.1 Rand Index, Homogeneity, Completeness and V-measure Scores

One of the best-known methods for comparing the results in clustering techniques in statistics is the *Rand index* or Rand measure (named after William M. Rand). The Rand index evaluates the similarity between two results of data clustering. Since in unsupervised clustering, class labels are not known, we use the Rand index to compare the coincidence of different clusterings obtained by different approaches or criteria. As an alternative, we later discuss the *Silhouette coefficient*: instead of

comparing different clusterings, this evaluates the compactness of the results of applying a specific clustering approach.

Given a set of n elements $S = \{o_1, \ldots, o_n\}$, we can compare two partitions of S[1]: $X = \{X_1, \ldots, X_r\}$, a partition of S into r subsets; and $Y = \{Y_1, \ldots, , Y_s\}$, a partition of S into s subsets. Let us use the annotations as follows:

- a is the number of pairs of elements in S that are in the same subset in both X and Y;
- b is the number of pairs of elements in S that are in different subsets in both X and Y;
- c is the number of pairs of elements in S that are in the same subset in X, but in different subsets in Y; and
- d is the number of pairs of elements in S that are in different subsets in X, but in the same subset in Y.

The Rand index, R, is defined as follows:

$$R = \frac{a+b}{a+b+c+d},$$

ensuring that its value is between 0 and 1.

One of the problems of the Rand index is that when given two datasets with random labelings, it does not take a constant value (e.g., zero) as expected. Moreover, when the number of clusters increases it is desirable that the upper limit tends to the unity. To solve this problem, a form of the Rand index, called the *Adjusted Rand index*, is used that adjusts the Rand index with respect to chance grouping of elements. It is defined as follows:

$$AR = \frac{\binom{n}{2}(a+d) - [(a+b)(a+c)+(c+d)(b+d)]}{\binom{n}{2}^2 [(a+b)(a+c)+(c+d)(b+d)]}.$$

Another way for comparing clustering results is the V-measure. Let us first introduce some concepts. We say that a clustering result satisfies a *homogeneity* criterion if all of its clusters contain only data points which are members of the same original (single) class. A clustering result satisfies a *completeness* criterion if all the data points that are members of a given class are elements of the same predicted cluster. Note that both scores have real positive values between 0.0 and 1.0, larger values being desirable. For example, if we consider two toy clustering sets (e.g., original and predicted) with four samples and two labels, we get:

In [1]:
```
print("%.3f" % metrics.homogeneity_score([0, 0, 1, 1],
                                         [0, 0, 0, 0]))
```

Out[1]:
```
0.000
```

[1] https://en.wikipedia.org/wiki/Rand_index.

The homogeneity is 0 since the samples in the predicted cluster 0 come from original cluster 0 and cluster 1.

In [2]:
```
print metrics.completeness_score([0, 0, 1, 1],
                                 [1, 1, 0, 0])
```

Out[2]:
```
1.0
```

The completeness is 1 since all the samples from the original cluster with label 0 go into the same predicted cluster with label 1, and all the samples from the original cluster with label 1 go into the same predicted cluster with label 0.

However, how can we define a measure that takes into account the completeness as well as the homogeneity? The *V-measure* is the harmonic mean between the homogeneity and the completeness defined as follows:

$$v = 2 * (\text{homogeneity} * \text{completeness})/(\text{homogeneity} + \text{completeness}).$$

Note that this metric is not dependent of the absolute values of the labels: a permutation of the class or cluster label values will not change the score value in any way. Moreover, the metric is symmetric with respect to switching between the predicted and the original cluster label. This is very useful to measure the agreement of two independent label assignment strategies applied to the same dataset even when the real groundtruth is not known. If class members are completely split across different clusters, the assignment is totally incomplete, hence the V-measure is null:

In [3]:
```
print("%.3f" % metrics.v_measure_score([0, 0, 0, 0],
                                       [0, 1, 2, 3]))
```

Out[3]:
```
0.000
```

In contrast, clusters that include samples from different classes destroy the homogeneity of the labeling, hence:

In [4]:
```
print("%.3f" % metrics.v_measure_score([0, 0, 1, 1],
                                       [0, 0, 0, 0]))
```

Out[4]:
```
0.000
```

In summary, we can say that the advantages of the V-measure include that it has bounded scores: 0.0 means the clustering is extremely bad; 1.0 indicates a perfect clustering result. Moreover, it can be interpreted easily: when analyzing the V-measure, low completeness or homogeneity explain in which direction the clustering is not performing well. Furthermore, we do not assume anything about the cluster structure. Therefore, it can be used to compare clustering algorithms such as *K-means*, which assume isotropic blob shapes, with results of other clustering algorithms such as spectral clustering (see Sect. 7.2.3.2), which can find clusters with "folded" shapes. As a drawback, the previously introduced metrics are not normalized with regard to random labeling. This means that depending on the number of samples, clusters and groundtruth classes, a completely random labeling will

not always yield the same values for homogeneity, completeness and hence, the V-measure. In particular, random labeling will not yield a zero score, and they will tend further from zero as the number of clusters increases. It can be shown that this problem can reliably be overcome when the number of samples is high, i.e., more than a thousand, and the number of clusters is less than 10. These metrics require knowledge of the groundtruth classes, while in practice this information is almost never available or requires manual assignment by human annotators. Instead, as mentioned before, these metrics can be used to compare the results of different clusterings.

7.2.2.2 Silhouette Score

An alternative to the former scores is to evaluate the final 'shape' of the clustering result. This is the underlying idea behind the Silhouette coefficient. It is defined as a function of the intracluster distance of a sample in the dataset, a and the nearest-cluster distance, b for each sample.[2] Later, we will discuss different ways to compute the distance between clusters. The Silhouette coefficient for a sample i can be written as follows:

$$Silhouette(i) = \frac{b - a}{\max(a, b)}.$$

Hence, if the Silhouette $s(i)$ is close to 0, it means that the sample is on the border of its cluster and the closest one from the rest of the dataset clusters. A negative value means that the sample is closer to the neighbor cluster. The average of the Silhouette coefficients of all samples of a given cluster defines the "goodness" of the cluster. A high positive value, i.e., close to 1 would mean a compact cluster, and vice versa. And the average of the Silhouette coefficients of all clusters gives idea of the quality of the clustering result. Note that the Silhouette coefficient only makes sense when the number of labels predicted is less than the number of samples clustered.

The advantage of the Silhouette coefficient is that it is bounded between -1 and $+1$. Moreover, it is easy to show that the score is higher when clusters are dense and well separated; a logical feature when speaking about clusters. Furthermore, the Silhouette coefficient is generally higher when clusters are compact.

7.2.3 Taxonomies of Clustering Techniques

Within different clustering algorithms, one can find *soft partition* algorithms, which assign a probability of the data belonging to each cluster, and also *hard partition* algorithms, where each datapoint is assigned precise membership of one cluster. A typical example of a soft partition algorithm is the Mixture of Gaussians [1], which can be viewed as a density estimator method that assigns a confidence or

[2]The intracluster distance of sample i is obtained by the distance of the sample to the nearest sample from the same class, and the nearest-cluster distance is given by the distance to the closest sample from the cluster nearest to the cluster of sample i.

probability to each point in the space. A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. The universally used generative unsupervised clustering using a Gaussian mixture model is also known as *EM Clustering*. Each point in the dataset has a soft assignment to the K clusters. One can convert this soft probabilistic assignment into membership by picking out the most likely clusters (those with the highest probability of assignment).

An alternative to soft algorithms are the *hard partition* algorithms, which assign a unique cluster value to each element in the feature space. According to the grouping process of the hard partition algorithm, there are two large families of clustering techniques:

- *Partitional algorithms*: these start with a random partition and refine it iteratively. That is why sometimes these algorithms are called "flat" clustering. In this chapter, we will consider two partitional algorithms in detail: K-means and *spectral clustering*.
- *Hierarchical algorithms*: these organize the data into hierarchical structures, where data can be agglomerated in the bottom-up direction, or split in a top-down manner. In this chapter, we will discuss and illustrate *agglomerative clustering*.

A typical hard partition algorithm is K-means clustering. We will now discuss it in some detail.

7.2.3.1 K-means Clustering

K-means algorithm is a hard partition algorithm with the goal of assigning each data point to a single cluster. K-means algorithm divides a set of n samples X into k disjoint clusters c_i, $i = 1, \ldots, k$, each described by the mean μ_i of the samples in the cluster. The means are commonly called cluster *centroids*. The K-means algorithm assumes that all k groups have equal variance.

K-means clustering solves the following minimization problem:

$$\arg min_c \sum_{j=1}^{k} \sum_{x \in c_j} d(x, \mu_j) = \arg min_c \sum_{j=1}^{k} \sum_{x \in c_j} ||x - \mu_j||_2^2 \tag{7.1}$$

where c_i is the set of points that belong to cluster i and μ_i is the center of the class c_i. K-means clustering objective function uses the square of the Euclidean distance $d(x, \mu_j) = ||x - \mu_j||^2$, that is also referred to as the *inertia* or *within-cluster sum-of-squares*. This problem is not trivial to solve (in fact, it is NP-hard problem), so the algorithm only hopes to find the global minimum, but may become stuck at a different solution.

In other words, we may wonder whether the centroids should belong to the original set of points:

$$inertia = \sum_{i=0}^{n} min_{\mu_j \in c}(||x_i - \mu_j||^2)). \tag{7.2}$$

The K-means algorithm, also known as Lloyd's algorithm, is an iterative procedure that searches for a solution of the K-means clustering problem and works as follows. First, we need to decide the number of clusters, k. Then we apply the following procedure:

1. Initialize (e.g., randomly) the k cluster centers, called *centroids*.
2. Decide the class memberships of the n data samples by assigning them to the nearest-cluster centroids (e.g., the center of gravity or mean).
3. Re-estimate the k cluster centers, c_i, by assuming the memberships found above are correct.
4. If none of the n objects changes its membership from the last iteration, then exit. Otherwise go to step 2.

Let us illustrate the algorithm in Python. First, we will create three sample distributions:

In [5]:
```
MAXN = 40
X = np.concatenate([
    1.25*np.random.randn(MAXN, 2),
    5 + 1.5*np.random.randn(MAXN, 2)])
X = np.concatenate([
    X, [8, 3] + 1.2*np.random.randn(MAXN, 2)])
```

The sample distributions generated are shown in Fig. 7.1 (left). However, the algorithm is not aware of their distribution. Figure 7.1 (right) shows what the algorithm sees. Let us assume that we expect to have three clusters ($k = 3$) and apply the K-means command from the Scikit-learn library:

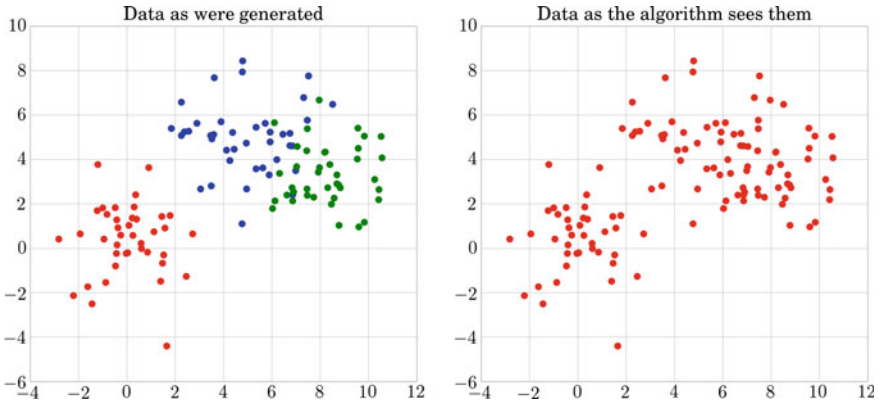


Fig. 7.1 Initial samples as generated (*left*), and samples seen by the algorithm (*right*)

In [6]:
```
from sklearn import cluster

K = 3 # Assuming we have 3 clusters!
clf = cluster.KMeans(init = 'random', n_clusters = K)
clf.fit(X)
```

Out[6]:
```
KMeans(copy_x=True, init='random', max_iter=300,
n_clusters=3, n_init=10, n_jobs=1, precompute_distances=True,
random_state=None, tol=0.0001, verbose=0)
```

Each clustering algorithm in Scikit-learn is used as follows. First, an object from the clustering technique is instantiated. Then we can use the `fit` method to adjust the learning parameters. We also find the method `predict` that, given new data, returns the cluster they belong to. For the class, the labels over the training data can be found in the `labels_` attribute or alternatively they can be obtained using the predict method.

How many "mis-clusterings" do we have? In order to see this, we tessellate the space and color all grid points from the same cluster with the same color. Then, we overlay the initial sample distributions (see Fig. 7.2). In the ideal case, we expect that in each partitioned subspace the sample points are of the same color. However, as shown in Fig. 7.2, the resulting clustering, which is represented in the figure by the color subspace in gray, does not usually coincide exactly with the initial distribution, which is represented by the color of the data. For example, in the same figure, if most of the blue points belong to the same cluster, there are a few ones that belong to the space occupied by the green data.

When computing the Rand index, we get:

In [7]:
```
print ('The Adjusted Rand index is: %.2f' %
       metrics.adjusted_rand_score(y.ravel(), clf.labels_))
```

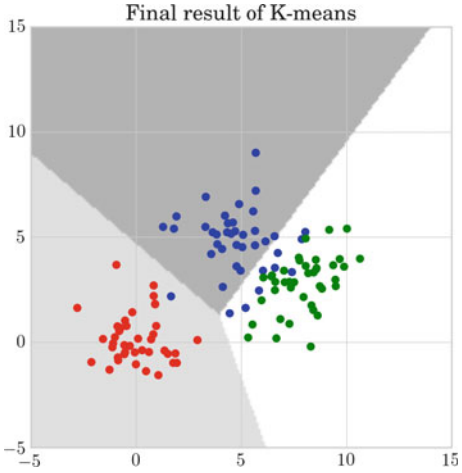


Fig. 7.2 Original samples (*dots*) generated by three distributions and the partition of the space according to the K-means clustering

```
Out[7]: The Adjusted Rand index is: 0.66
```

Taking into account that the Adjusted Rand index belongs to the interval [0, 1], the result of 0.66 in our example means that although most of the clusters were discovered, not 100% of them were; as confirmed by Fig. 7.2.

The inertia can be seen as a measure of how internally coherent the clusters are. Several issues should be taken into account:

- The inertia assumes that clusters are isotropic and convex, since the Euclidean distance is applied, which is isotropic with regard to the different dimensions of the data. However, we cannot expect that the data fulfill this assumption by default. Hence, the K-means algorithm responds poorly to elongated clusters or manifolds with irregular shapes.
- The algorithm may not ensure convergence to the global minimum. It can be shown that K-means will always converge to a local minimum of the inertia (Eq. (7.2)). It depends on the random initialization of the seeds, but some seeds can result in a poor convergence rate, or convergence to suboptimal clustering. To alleviate the problem of local minima, the K-means computation is often performed several times, with different centroid initializations. One way to address this issue is the `k-means++` initialization scheme, which has been implemented in `Scikit-learn` (use the `init='kmeans++'` parameter). This parameter initializes the centroids to be (generally) far from each other, thereby probably leading to better results than random initialization.
- This algorithm requires the number of clusters to be specified. Different heuristics can be applied to predetermine the number of seeds of the algorithm.
- It scales well to a large number of samples and has been used across a large range of application areas in many different fields.

In summary, we can conclude that K-means has the advantages of allowing the easy use of heuristics to select good seeds; initialization of seeds by other methods; multiple points to be tried. However, in contrast, it still cannot ensure that the local minima problem is overcome; it is iterative and hence slow when there are a lot of high-dimensional samples; and it tends to look for spherical clusters.

7.2.3.2 Spectral Clustering

Up to this point, the clustering procedure has been considered as a way to find data groups following a notion of compactness. Another way of looking at what a cluster is is provided by connectivity (or similarity). Spectral clustering [2] refers to a family of methods that use spectral techniques. Specifically, these techniques are related to the eigendecomposition of an affinity or similarity matrix and solve the problem of clustering according to the connectivity of the data. Let us consider an ideal similarity matrix of two clear sets.

Let us denote the similarity matrix, S, as the matrix $S_{ij} = s(x_i, x_j)$ which gives the similarity between observations x_i and x_j. Remember that we can model similarity

using the Euclidean distance, $d(x_i, x_j) = ||x_i - x_j||^2$, by means of a Gaussian Kernel as follows:

$$s(x_i, x_j) = exp(-\alpha||x_i - x_j||^2),$$

where α is a parameter. We expect two points from different clusters to be far away from each other. However, if there is a sequence of points within the cluster that forms a "path" between them, this also would lead to big distance among some of the points from the same cluster. Hence, we define an affinity matrix A based on the similarity matrix S, where A contains positive values and is symmetric. This can be done, for example, by applying a k-nearest neighbor that builds a graph connecting just the k closest data points. The symmetry comes from the fact that A_{ij} and A_{ji} give the distance between the same points. Considering the affinity matrix, the clustering can be seen as a graph partition problem, where connected graph components correspond to clusters. The graph obtained by spectral clustering will be partitioned so that graph edges connecting different clusters have low weights, and vice versa. Furthermore, we define a degree matrix D, where each diagonal value is the degree of the respective graph node and all other elements are 0. Finally, we can compute the unnormalized graph Laplacian ($U = D - A$) and/or a normalized version of the Laplacian (L), as follows:

- *Simple Laplacian*: $L = I - D^{-1}A$, which corresponds to a random walk, being D^{-1} the transition matrix. Spectral clustering obtains groups of nodes such that the random walk corresponds to seldom transitions from one group to another.
- *Normalized Laplacian*: $L = D^{-\frac{1}{2}}UD^{-\frac{1}{2}}$.
- *Generalized Laplacian*: $L = D^{-1}U$.

If we assume that there are k clusters, the next step is to find the k smallest eigenvectors, without considering the trivial constant eigenvector. Each row of the matrix formed by the k smallest eigenvectors of the Laplacian matrix defines a transformation of the data x_i. Thus, in this transformed space, we can apply K-means clustering in order to find the final clusters. If we do not know in advance the number of clusters, k, we can look for sudden changes in the sorted eigenvalues of the matrix, U, and keep the smallest ones.

7.2.3.3 Hierarchical Clustering

Another well-known clustering technique of particular interest is hierarchical clustering. Hierarchical clustering is comprised of a general family of clustering algorithms that construct nested clusters by successive merging or splitting of data. The hierarchy of clusters is represented as a tree. The tree is usually called a *dendrogram*. The root of the dendrogram is the single cluster that contains all the samples; the leaves are the clusters containing only one sample each. This is a nice tool, since it can be straightforwardly interpreted: it "explains" how clusters are formed and visualizes clusters at different scales. The tree that results from the technique shows

the similarity between the samples. Partitioning is computed by selecting a cut on the tree at a certain level.

In general, there are two types of hierarchical clustering:

- *Top-down* divisive clustering applies the following algorithm:
  - Start with all the data in a single cluster.
  - Consider every possible way to divide the cluster into two.
  - Choose the best division.
  - Recursively, it operates on both sides until a stopping criterion is met. That can be something as follows: there are as much clusters as data; the predetermined number of clusters has been reached; the maximum distance between all possible partition divisions is smaller than a predetermined threshold; etc.
- *Bottom-up* agglomerative clustering applies the following algorithm:
  - Start with each data point in a separate cluster.
  - Repeatedly join the closest pair of clusters.
  - At each step, a stopping criterion is checked: there is only one cluster; a predetermined number of clusters has been reached; the distance between the closest clusters is greater than a predetermined threshold; etc.

  This process of merging forms a binary tree or hierarchy.

When merging two clusters, a question naturally arises: How to measure the similarity of two clusters? There are different ways to define this with different results for the agglomerative clustering. The linkage criterion determines the metric used for the cluster merging strategy:

- *Maximum* or *complete* linkage minimizes the maximum distance between observations of pairs of clusters. Based on the similarity of the two least similar members of the clusters, this clustering tends to give tight spherical clusters as a final result.
- *Average* linkage averages similarity between members, i.e., minimizes the average of the distances between all observations of pairs of clusters.
- *Ward* linkage minimizes the sum of squared differences within all clusters. It is thus a variance-minimizing approach and in this sense is similar to the K-means objective function, but tackled with an agglomerative hierarchical approach.

Let us illustrate how the different linkages work with an example. Let us generate three clusters as follows:

In [8]:
```
MAXN1 = 500
MAXN2 = 400
MAXN3 = 300
X1 = np.concatenate([
     2.25 * np.random.randn(MAXN1, 2),
     4 + 1.7*np.random.randn(MAXN2, 2)])
X1 = np.concatenate([
     X1, [8, 3] + 1.9*np.random.randn(MAXN3, 2)])

y1 = np.concatenate([
     np.ones((MAXN1, 1)),
     2 * np.ones((MAXN2, 1))])
y1 = np.concatenate([
     y1, 3 * np.ones((MAXN3, 1))]).ravel()
y1 = np.int_(y1)
labels_y1 = ['+', '*', 'o']
colors = ['r', 'g', 'b']
```

Let us apply agglomerative clustering using the different linkages:

In [9]:
```
from sklearn.cluster import AgglomerativeClustering

for linkage in ('ward', 'complete', 'average'):
    clustering = AgglomerativeClustering(linkage = linkage,
                                         n_clusters =3)
    clustering.fit(X1)

    x_min , x_max = np.min (X1, axis = 0) , np.max (X1,axis
        = 0)
    X1 = (X1 - x_min ) / ( x_max - x_min )
    plt.figure ( figsize =(5 , 5))
    for i in range (X1.shape [0]) :
        plt.text(X1[i, 0], X1[i, 1], labels_y1[y1[i]-1],
                 color = colors[y1[i]-1])
    plt.title("\%s linkage " \% linkage, size  = 20)
    plt.tight_layout()

plt.show()
```

The results of the agglomerative clustering using the different linkages: complete, average, and Ward are given in Fig. 7.3. Note that agglomerative clustering exhibits "rich get richer" behavior that can sometimes lead to uneven cluster sizes, with average linkage being the worst strategy in this respect and Ward linkage giving the most regular sizes. Ward linkage is an attempt to form clusters that are as compact as possible, since it considers inter- and intra-distances of the clusters. Meanwhile, for non-Euclidean metrics, average linkage is a good alternative. Average linkage can produce very unbalanced clusters, it can even separate a single data point into a separate cluster. This fact would be useful if we want to detect outliers, but it may be undesirable when two clusters are very close to each other, since it would tend to merge them.

Agglomerative clustering can scale to a large number of samples when it is used jointly with a connectivity matrix, but it is computationally expensive when no con-

nectivity constraints are added between samples: it considers all the possible merges at each step.

7.2.3.4 Adding Connectivity Constraints

Sometimes, we are interested in introducing a connectivity constraint into the clustering process so that merging of nonadjacent points is avoided. This can be achieved by constructing a connectivity matrix that defines which are the neighboring samples in the dataset. For instance, in the example in Fig. 7.4, we want to avoid the formation of clusters of samples from the different circles. A sample code to compute agglomerative clustering with connectivity would be as follows:

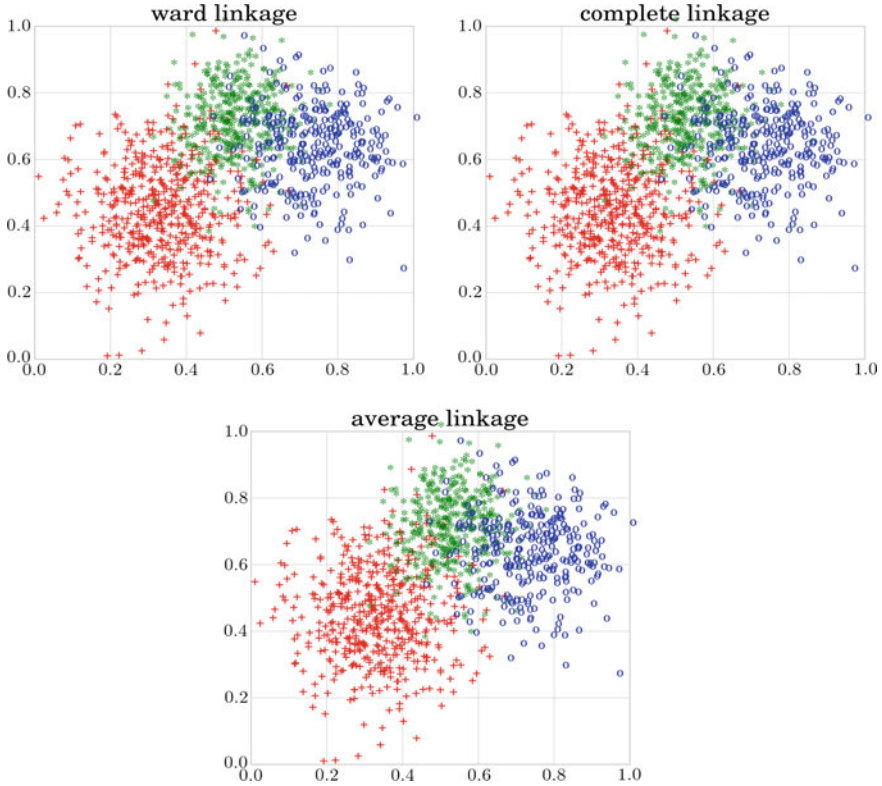


Fig. 7.3 Illustration of agglomerative clustering using different linkages: Ward, complete, and average. The symbol of each data point corresponds to the original class generated and the color corresponds to the cluster obtained

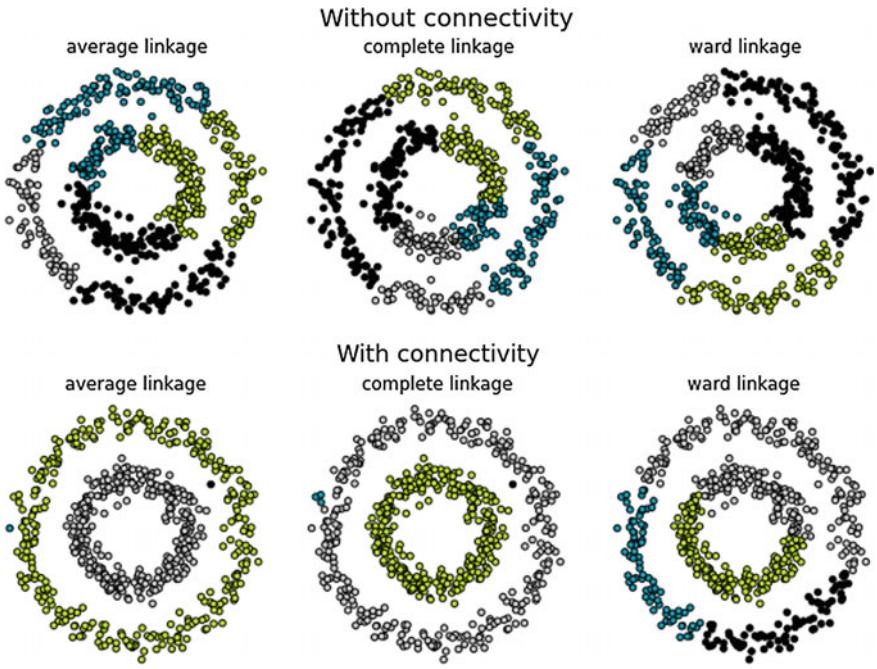


Fig. 7.4 Illustration of agglomerative clustering without (*top row*) and with (*bottom row*) a connectivity graph using the three linkages (from *left* to *right*): average, complete, and Ward. The *colors* correspond to the clusters obtained

In [10]:
```
connectivity = kneighbors_graph(X, 30)
model = AgglomerativeClustering(linkage = 'average',
    connectivity = connectivity, n_clusters = 8)
model.fit(X)
```

A connectivity constraint is useful to impose a certain local structure, but it also makes the algorithm faster, especially when the number of the samples is large. A connectivity constraint is imposed via a *connectivity matrix*: a sparse matrix that only has elements at the intersection of a row and a column with indexes of the dataset that should be connected. This matrix can be constructed from a priori information or can be learned from the data, for instance using `kneighbors_graph` to restrict merging to nearest neighbors or using `image.grid_to_graph` to limit merging to neighboring pixels in an image, both from Scikit-learn. This phenomenon can be observed in Fig. 7.4, where in the first row we see the results of the agglomerative clustering without using a connectivity graph. The clustering can join data from different circles (e.g., the black cluster). At the bottom, the three linkages use a connectivity graph and thus two of them avoid joining data points that belong to different circles (except the Ward linkage that attempts to form compact and isotropic clusters).

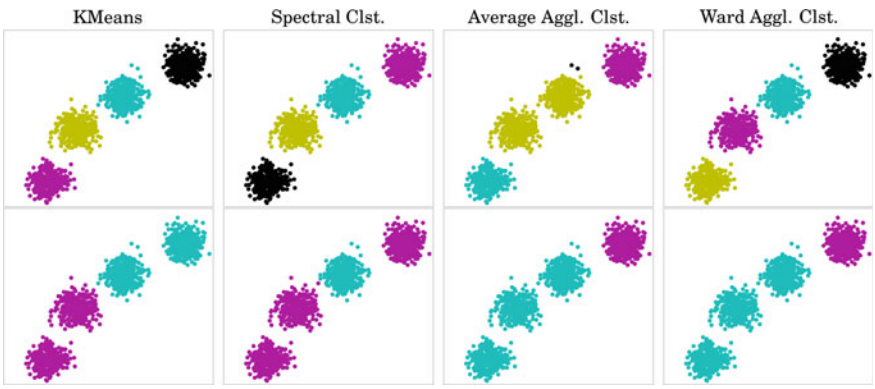


Fig. 7.5 Comparison of the different clustering techniques (from *left* to *right*): K-means, spectral clustering, and agglomerative clustering with average and Ward linkage on simple compact datasets. In the *first row*, the expected number of clusters is $k = 2$ and in the *second row*: $k = 4$

7.2.3.5 Comparison of Different Hard Partition Clustering Algorithms

Let us compare the behavior of the different clustering algorithms discussed so far. For this purpose, we generate three different datasets' configurations:

(a) 4 spherical groups of data;
(b) a uniform data distribution; and
(c) a non-flat configuration of data composed of two moon-like groups of data.

An easy way to generate these datasets is by using `Scikit-learn` that has predefined functions for it: `datasets.make_blobs()`, `datasets.ma- ke_ moons()`, etc.

We apply the clustering techniques discussed above, namely K-means, agglomerative clustering with average linkage, agglomerative clustering with Ward linkage, and spectral clustering. Let us test the behavior of the different algorithms assuming $k = 2$ and $k = 4$. Connectivity is applied in the algorithms where it is applicable.

In the simple case of separated clusters of data and $k = 4$, most of the clustering algorithms perform well, as expected (see Fig. 7.5). The only algorithm that could not discover the four groups of samples is the average agglomerative clustering. Since it allows highly unbalanced clusters, the two noisy data points that are quite separated from the closest two blobs were considered as a different cluster, while the two central blobs were merged in one cluster. In case of $k = 2$, each of the methods is obligated to join at least two blobs in a cluster.

Regarding the uniform distribution of data (see Fig. 7.6), K-means, Ward linkage agglomerative clustering and spectral clustering tend to yield even and compact clusters; while the average linkage agglomerative clustering attempts to join close points as much as possible following the "rich get richer" rule. This results in a

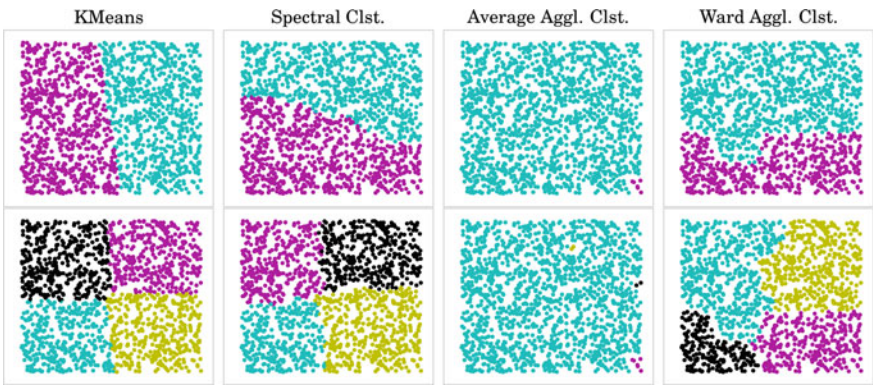


Fig. 7.6 Comparison of the different clustering techniques (from *left* to *right*): K-means, spectral clustering, and agglomerative clustering with average and Ward linkage on uniformly distributed data. In the *first row*, the number of clusters assumed is $k = 2$ and in the *second row*: $k = 4$

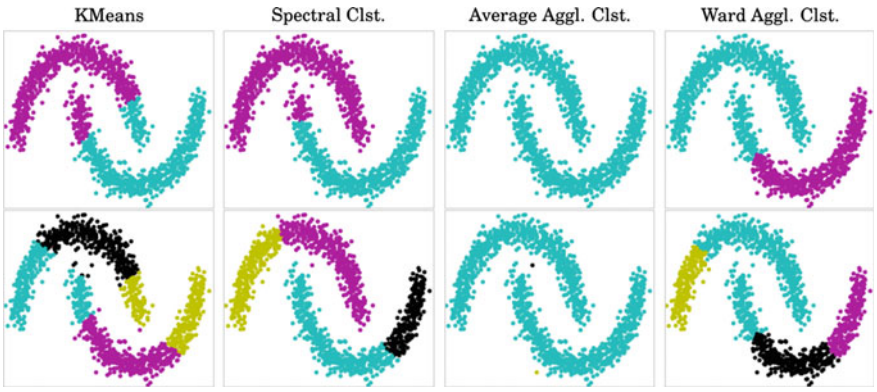


Fig. 7.7 Comparison of the different clustering techniques (from *left* to *right*): K-means, spectral clustering, and agglomerative clustering with average and Ward linkage on non-flat geometry datasets. In the *first row*, the expected number of clusters is $k = 2$ and in the *second row*: $k = 4$

second cluster of a small set of data. This behavior is observed in both cases: $k = 2$ and $k = 4$.

Regarding datasets with more complex geometry, like in the moon dataset (see Fig. 7.7), K-means and Ward linkage agglomerative clustering attempt to construct compact clusters and thus cannot separate the moons. Due to the connectivity constraint, the spectral clustering and the average linkage agglomerative clustering separated both moons in case of $k = 2$, while in case of $k = 4$, the average linkage agglomerative clustering clustered most of datasets correctly separating some of the noisy data points as two separate single clusters. In the case of spectral clustering, looking for four clusters, the method splits each of the two moon datasets into two clusters.

	0	1	2	3	4	5	6	7	8	9	10	11
GEO												
Albania	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Austria	0.52	5.25	9.98	0.64	1.22	0.61	1.01	2.64	1.63	5.89	5.90	11.20
Belgium	0.34	6.25	11.90	0.32	0.61	0.78	1.54	2.79	1.46	6.57	6.44	12.51
Bulgaria	0.63	3.35	8.96	0.74	1.99	0.92	0.80	1.76	0.61	4.10	4.18	10.95
Croatia	0.26	4.24	NaN	0.03	NaN	0.65	1.87	0.97	0.78	4.27	4.42	NaN

Fig. 7.8 Expenditure on different educational indicators for the first five countries in the Eurostat dataset

7.3 Case Study

In order to illustrate clustering with a real dataset, we will now analyze the indicators of spending on education among the European Union member states, provided by the Eurostat data bank.[3] The data are organized by year (TIME) from 2002 until 2011 and country (GEO): ('Albania', 'Austria', 'Belgium', 'Bulgaria', etc.). Twelve indicators (INDIC_ED) of financing of education with their corresponding values (Value) are given: (1) Expenditure on educational institutions from private sources as % of gross domestic product (GDP), for all levels of education combined; (2) Expenditure on educational institutions from public sources as % of GDP, for all levels of government combined, (3) Expenditure on educational institutions from public sources as % of total public expenditure, for all levels of education combined, (4) Public subsidies to the private sector as % of GDP, for all levels of education combined, (5) Public subsidies to the private sector as % of total public expenditure, for all levels of education combined, etc. We can store the 12 indicators for a given year (e.g., 2010) in a table. Figure 7.8 provides visualization of the first five countries in the table.

As we can observe, this is not a clean dataset, since there are values missing. Some countries have very limited information and should be excluded. Other countries may still not collect or have access to a few indicators. For these last cases, we can proceed in two ways: (a) fill in the gaps with some non-informative, non-biasing data; or (b) drop the features with missing values for the analysis. If we have many features and only a few have missing values, then it is not very harmful to drop them. However, if missing values are spread across most of the features, we eventually have to deal with them. In our case, both options seem reasonable, as long as the number of missing features for a country is not too large. We will proceed in both ways at the same time.

We apply both options: filling the gap with the mean value of the feature and the dropping option, ignoring the indicators with missing values. Let us now apply K-means clustering to these data in order to partition the countries according to

[3] http://ec.europa.eu/eurostat.

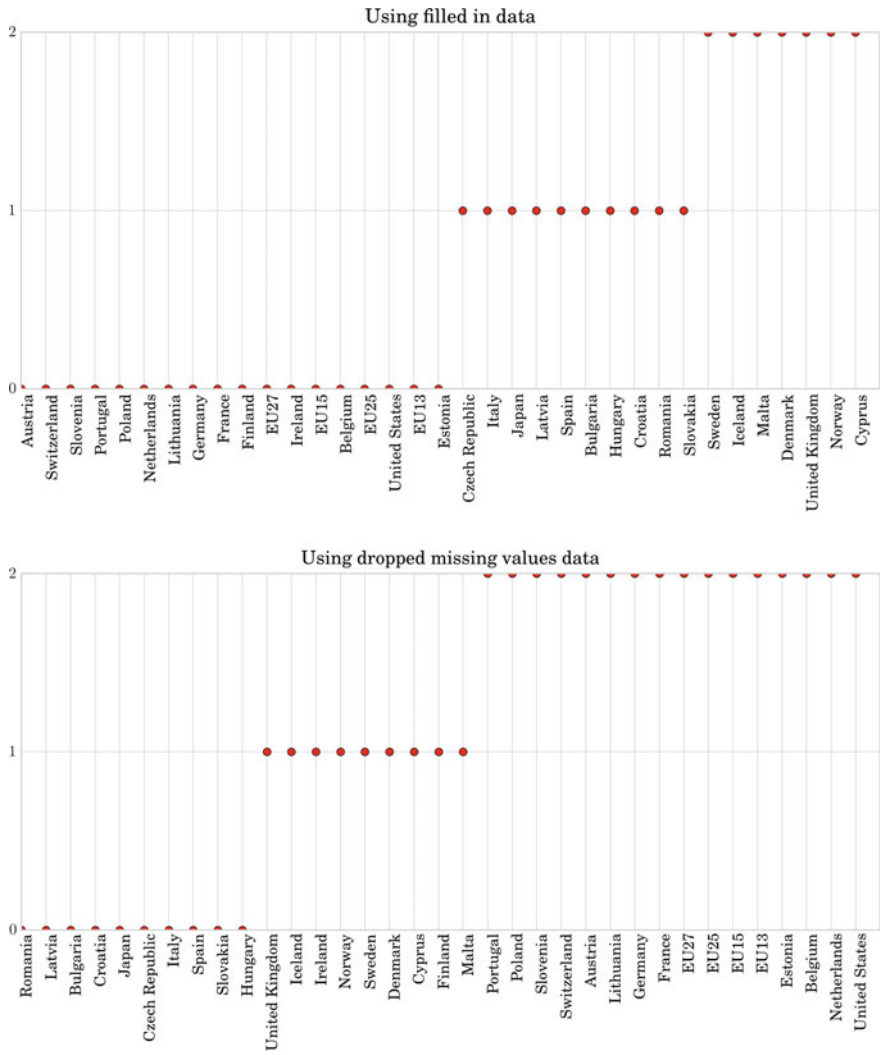


Fig. 7.9 Clustering of the countries according to their educational expenditure using filled-in (*top row*) and dropped (*bottom row*) missing values

their investment in education and check their profiles. Figure 7.9 shows the results of this K-means clustering. We have sorted the data for better visualization. At a simple glance, we can see that the partitions (top and bottom of Fig. 7.9) are different. Most countries in cluster 2 in the filled-in dataset correspond to cluster 0 in the dropped missing values dataset. Analogously, most of cluster 0 in the filled-in dataset correspond to cluster 1 in the dropped missing values dataset; and most countries from cluster 1 in the filled-in dataset correspond to cluster 2 in the dropped

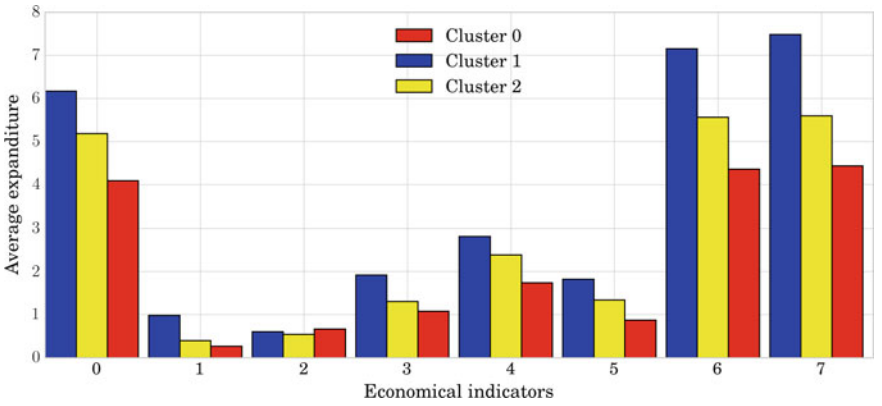


Fig. 7.10 Mean expenditure of the different clusters according to the 8 indicators of the indicators-dropped dataset

set. Still, there are some countries that do not follow this rule. That is, looking at both clusterings, they may yield similar (up to label permutation) results, but they will not necessarily always coincide. This is mainly due to two aspects: the random initialization of the K-means clustering and the fact that each method works in a different space (i.e., dropped data in 8D space *vs* filled-in data, working in 12D space). Note that we should not consider the assigned absolute cluster value, since it is irrelevant. The mean expenditure of the different clusters is shown by different colors according to the 8 indicators of the indicators-dropped dataset (see Fig. 7.10).

So, without loss of generality, we continue analyzing the set obtained by dropping missing values. Let us now check the clusters and check their profile by looking at the centroids. Visualizing the eight values of the three clusters (see Fig. 7.10), we can see that cluster 1 spends more on education for the 8 educational indicators, while cluster 0 is the one with least resources invested in education.

Let us consider a specific country, e.g., Spain and its expenditure on education. If we refine cluster 0 further and check how close members are from this cluster to cluster 1, it may give us a hint as to a possible ordering. When visualizing the distance to cluster 0 and 1, we can observe that Spain, while being from cluster 0, has a smaller distance to cluster 1 (see Fig. 7.11). This should make us realize that using 3 clusters probably does not sufficiently represent the groups of countries. So we redo the process, but applying $k = 4$: we obtain 4 clusters. This time cluster 0 includes the EU members with medium expenditure (Fig. 7.12). This reinforce the intuition about Spain being a limit case in the former clustering. The clusters obtained are as follows:

- Cluster 0: ('Austria', 'Estonia', 'EU13', 'EU15', 'EU25', 'EU27', 'France', 'Germany', 'Hungary', 'Latvia', 'Lithuania', 'Netherlands', 'Poland', 'Portugal', 'Slovenia', 'Spain', 'Switzerland', 'United Kingdom', 'United States')

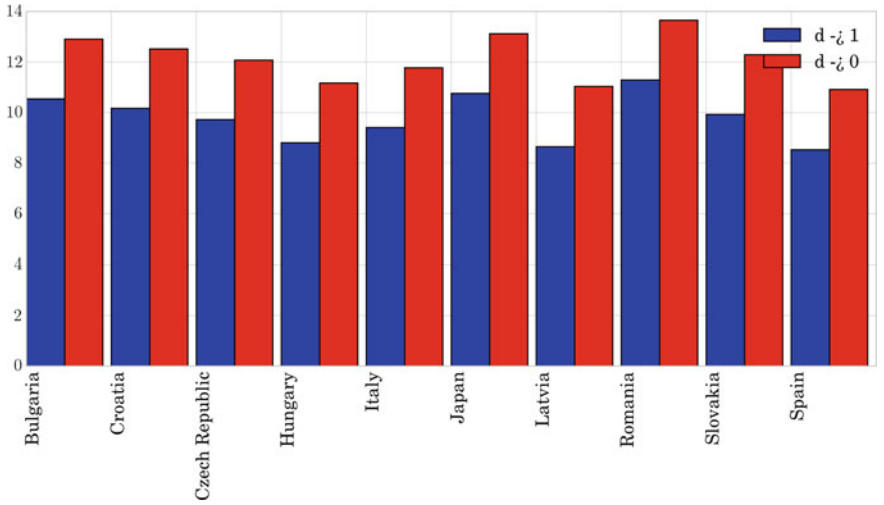


Fig. 7.11 Distance of countries in cluster 0 to centroids of cluster 0 (in *red*) and cluster 1 (in *blue*)

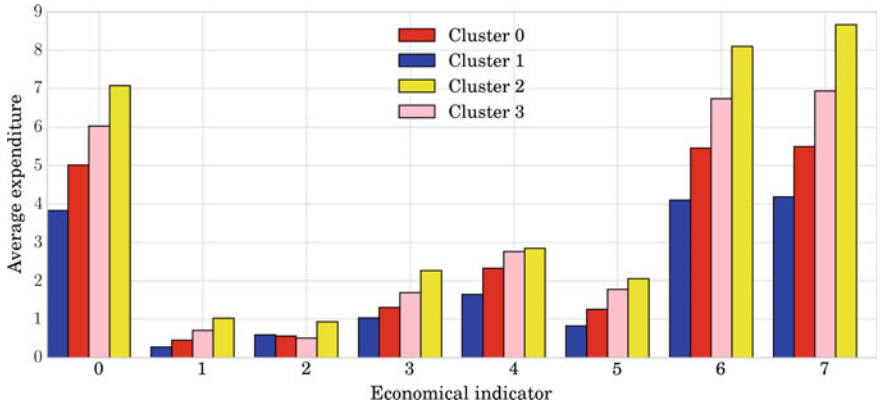


Fig. 7.12 K-means applied to the Eurostat dataset grouping the countries into four clusters

- Cluster 1: ('Bulgaria', 'Croatia', 'Czech Republic', 'Italy', 'Japan', 'Romania', 'Slovakia')
- Cluster 2: ('Cyprus', 'Denmark', 'Iceland')
- Cluster 3: ('Belgium', 'Finland', 'Ireland', 'Malta', 'Norway', 'Sweden')

We can repeat the process using the alternative clustering techniques and compare their results. Let us first apply spectral clustering. The corresponding code will be as follows:

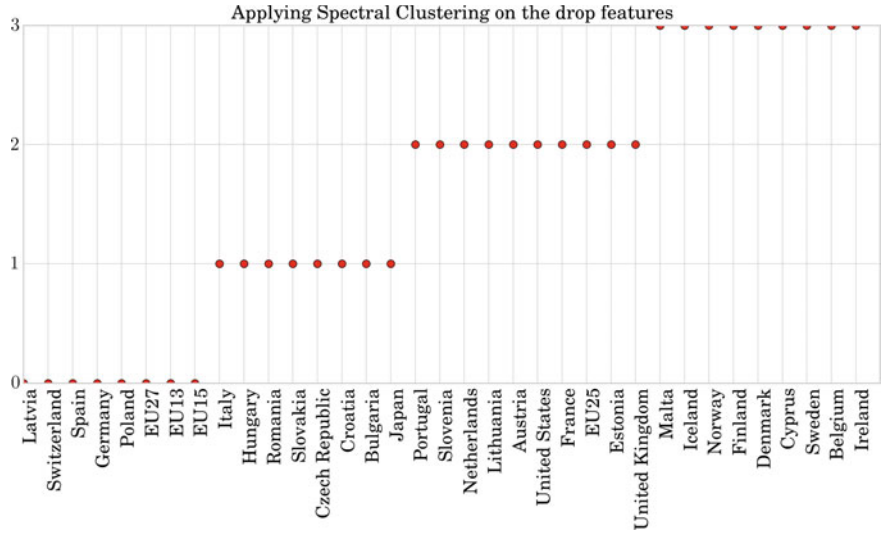


Fig. 7.13 Spectral clustering applied to the European countries according to their expenditure on education

In [11]:
```
X = StandardScaler().fit_transform(edudrop.values)
distances = euclidean_distances(edudrop.values)
spectral = cluster.SpectralClustering(
    n_clusters = 4, affinity = "nearest_neighbors")
spectral.fit(edudrop.values)
y_pred = spectral.labels_.astype(np.int)
```

The result of this spectral clustering is shown in Fig. 7.13. Note that in general, the aim of spectral clustering is to obtain more balanced clusters. In this way, the predicted cluster 1 merges clusters 2 and 3 of the K-means clustering, cluster 2 corresponds to cluster 1 of the K-means clustering, cluster 0 mainly shifts to cluster 2, and cluster 3 corresponds to cluster 0 of the K-means.

Applying agglomerative clustering, not only we do obtain different clusters, but also we can see how different clusters are obtained. Thus, in some way it is giving us information on which the most similar pairs of countries and clusters are. The corresponding code that applies the agglomerative clustering will be as follows:

In [12]:
```
from scipy.cluster.hierarchy import linkage, dendrogram
from scipy.spatial.distance import pdist

X_train = edudrop.values
dist = pdist(X_train, 'euclidean')
linkage_matrix = linkage(dist, method = 'complete');

plt.figure(figsize = (11.3, 11.3))
dendrogram(linkage_matrix, orientation="right",
           color_threshold = 3,
           labels = wrk_countries_names,
           leaf_font_size = 20);
plt.tight_layout()
```

In `Scikit-learn`, the parameter *color_threshold* of the command dendrogram() colors all the descendent links below a cluster node k the same color if k is the first node below the color_threshold. All links connecting nodes with distances greater than or equal to the threshold are colored blue. Hence, using *color_threshold* = 3, the clusters obtained are as follows:

- Cluster 0: ('Cyprus', 'Denmark', 'Iceland')
- Cluster 1: ('Bulgaria', 'Croatia', 'Czech Republic', 'Italy', 'Japan', 'Romania', 'Slovakia')
- Cluster 2: ('Belgium', 'Finland', 'Ireland', 'Malta', 'Norway', 'Sweden')
- Cluster 3: ('Austria', 'Estonia', 'EU13', 'EU15', 'EU25', 'EU27', 'France', 'Germany', 'Hungary', 'Latvia', 'Lithuania', 'Netherlands', 'Poland', 'Portugal', 'Slovenia', 'Spain', 'Switzerland', 'United Kingdom', 'United States')
  Note that, to a high degree, they correspond to the clusters obtained by the K-means (except for permutation of cluster labels, which is irrelevant).

Figure 7.14 shows the construction of the clusters using complete linkage agglomerative clustering. Different cuts at different levels of the dendrogram allow us to obtain different numbers of clusters.

To summarize, we can compare the results of the three clustering approaches. We cannot expect the results to coincide, since the different approaches are based on different criteria for constructing clusters. Nonetheless, we can still observe that in this case, K-means and the agglomerative approaches gave the same results (up to a permutation of the number of cluster, which is irrelevant); while spectral clustering gave more evenly distributed clusters. This later approach fused clusters 0 and 2 of the agglomerative clustering in cluster 1, and split cluster 3 of the agglomerative clustering into its clusters 0 and 3. Note that these results could change when using different distances among data.

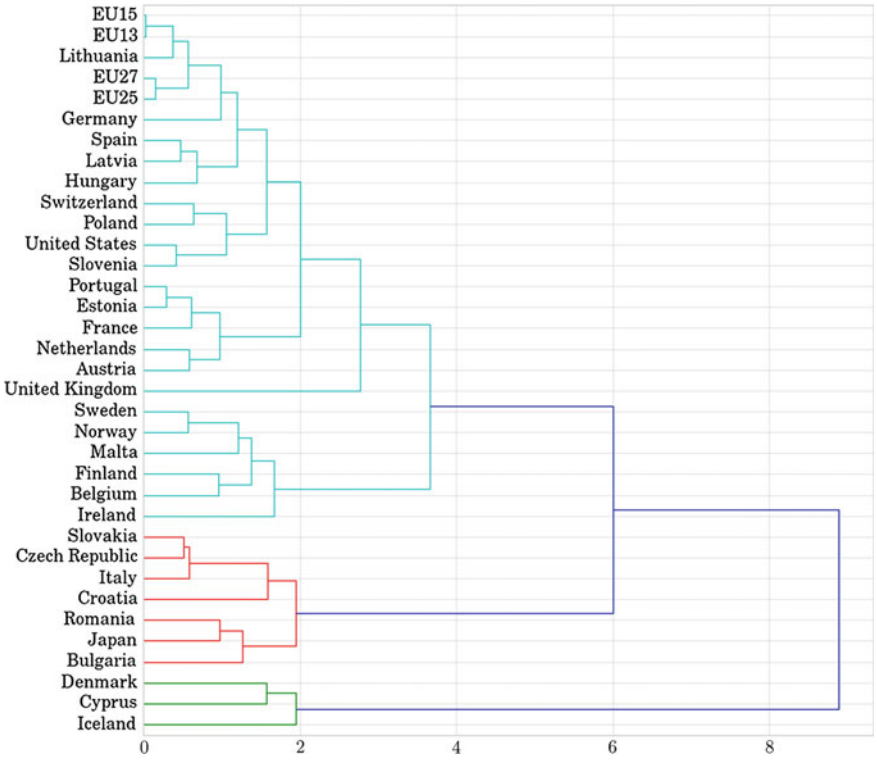


Fig. 7.14 Agglomerative clustering applied to cluster European countries according to their expenditure on education

7.4 Conclusions

In this chapter, we have introduced the unsupervised learning problem as a problem of knowledge or structure discovery from a set of unlabeled data. We have focused on clustering as one of the main problems in unsupervised learning. Basic concepts such as distance, similarity, connectivity, and the quality of the clustering results have been discussed as the main elements to be determined before choosing a specific clustering technique. Three basic clustering techniques have been introduced: K-means, agglomerative clustering, and spectral clustering. We have discussed their advantages and disadvantages and compared them through different examples. One of the important parameters for most clustering techniques is the number of clusters expected.

Regarding scalability, K-means can be applied to very large datasets, but the number of clusters should be as much as medium value, due to its iterative procedure. Spectral clustering can manage datasets that are not very large and a reasonable number of clusters, since it is based on computing the eigenvectors of the affinity matrix. In this aspect, the best option is hierarchical clustering, which allows large

numbers of samples and clusters to be tackled. Regarding uses, K-means is best suited to data with a flat geometry (isotropic and compact clusters), while spectral clustering and agglomerative clustering, with either average or complete linkage, are able to detect patterns in data with non-flat geometry. The connectivity graph is especially helpful in such cases. At the end of the chapter, a case study using a Eurostat database has been considered to show the applicability of the clustering in real problems (with real datasets).

Acknowledgements This chapter was co-written by Petia Radeva and Oriol Pujol.

References

1. Press, WH; Teukolsky, SA; Vetterling, W.T.; Flannery, B.P. (2007). "Section 16.1. Gaussian Mixture Models and k-Means Clustering". Numerical Recipes: The Art of Scientific Computing (3rd ed.). New York: Cambridge University Press. ISBN 978-0-521-88068-8.
2. Meilă, M.; Shi, J. (2001); "Learning Segmentation by Random Walks", Neural Information Processing Systems 13 (NIPS 2000), 2001, pp. 873–879.
3. Székely, G.J.; Rizzo, M.L. (2005). "Hierarchical clustering via Joint Between-Within Distances: Extending Ward's Minimum Variance Method", Journal of Classification 22, 151–183.

Network Analysis

8

8.1 Introduction

Network data are generated when we consider relationships between two or more entities in the data, like the highways connecting cities, friendships between people or their phone calls. In recent years, a huge number of network data are being generated and analyzed in different fields. For instance, in sociology there is interest in analyzing blog networks, which can be built based on their citations, to look for divisions in their structures between political orientations. Another example is infectious disease transmission networks, which are built in epidemiological studies to find the best way to prevent infection of people in a territory, by isolating certain areas. Other examples studied in the field of technology include interconnected computer networks or power grids, which are analyzed to optimize their functioning. We also find examples in academia, where we can build co-authorship networks and citation networks to analyze collaborations among Universities.

Structuring data as networks can facilitate the study of the data for different goals; for example, to discover the weaknesses of a structure. That could be the objective of a biologist studying a community of plants and trying to establish which of its properties promote quick transmission of a disease. A contrasting objective would be to find and exploit structures that work efficiently for the transmission of messages across the network. This may be the goal of an advertising agent trying to find the best strategy for spreading publicity.

How to analyze networks and extract the features we want to study are some of the issues we consider in this chapter. In particular, we introduce some basic concepts related with networks, such as connected components, centrality measures, ego-networks, and PageRank. We present some useful Python tools for the analysis of networks and discuss some of the visualization options. In order to motivate and illustrate the concepts, we perform social network analysis using real data. We present a practical case based on a public dataset which consists of a set of interconnected

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_8

Facebook friendship networks. We formulate multiple questions at different levels: the local/member level, the community level, and the global level.

In general, some of the questions we try to solve are the following:

- What type of network are we dealing with?
- Which is the most representative member of the network in terms of being the most connected to the rest of the members?
- Which is the most representative member of the network in terms of being the most circulated on the paths between the rest of the members?
- Which is the most representative member of the network in terms of proximity to the rest of the members?
- Which is the most representative member of the network in terms of being the most accessible from any location in the network?
- There are many ways of calculating the representativeness or importance of a member, each one with a different meaning, so: how can we illustrate them and compare them?
- Are there different communities in the network? If so, how many?
- Does any member of the network belong to more than one community? That is, is there any overlap between the communities? How much overlap? How can we illustrate this overlap?
- Which is the largest community in the network?
- Which is the most dense community (in terms of connections)?
- How can we automatically detect the communities in the network?
- Is there any difference between automatically detected communities and real ones (manually labeled by users)?

8.2 Basic Definitions in Graphs

Graph is the mathematical term used to refer to a network. Thus, the field that studies networks is called *graph theory* and it provides the tools necessary to analyze networks. Leonhard Euler defined the first graph in 1735, as an abstraction of one of the problems posed by mathematicians of the time regarding Konigsberg, a city with two islands created by the River Pregel, which was crossed by seven bridges. The problem was: is it possible to walk through the town of Konigsberg crossing each bridge once and only once? Euler represented the land areas as nodes and the bridges connecting them as edges of a graph and proved that the walk was not possible for this particular graph.

A graph is defined as a set of *nodes*, which are an abstraction of any entities (parts of a city, persons, etc.), and the connecting links between pairs of nodes called *edges* or relationships. The edge between two nodes can be *directed* or *undirected*. A directed edge means that the edge points from one node to the other and not the other way round. An example of a directed relationship is "a person knows another person". An edge has a direction when person A knows person B, and not the reverse direction

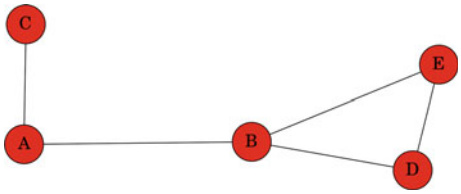


Fig. 8.1 Simple undirected labeled graph with 5 nodes and 5 edges

if B does not know A (which is usual for many fans and celebrities). An undirected edge means that there is a symmetric relationship. An example is "a person shook hands with another person"; in this case, the relationship, unavoidably, involves both persons and there is no directionality. Depending on whether the edges of a graph are directed or undirected, the graph is called a *directed graph* or an *undirected graph*, respectively.

The *degree* of a node is the number of edges that connect to it. Figure 8.1 shows an example of an undirected graph with 5 nodes and 5 edges. The degree of node C is 1, while the degree of nodes A, D and E is 2 and for node B it is 3. If a network is directed, then nodes have two different degrees, the *in-degree*, which is the number of incoming edges, and the *out-degree*, which is the number of outgoing edges.

In some cases, there is information we would like to add to graphs to model properties of the entities that the nodes represent or their relationships. We could add *strengths* or *weights* to the links between the nodes, to represent some real-world measure. For instance, the length of the highways connecting the cities in a network. In this case, the graph is called a *weighted graph*.

Some other elementary concepts that are useful in graph analysis are those we explain in what follows. We define a *path* in a network to be a sequence of nodes connected by edges. Moreover, many applications of graphs require *shortest paths* to be computed. The shortest path problem is the problem of finding a path between two nodes in a graph such that the length of the path or the sum of the weights of edges in the path is minimized. In the example in Fig. 8.1, the paths (C, A, B, E) and (C, A, B, D, E) are those between nodes C and E. This graph is unweighted, so the shortest path between C and E is the one that follows the fewer edges: (C, A, B, E).

A graph is said to be *connected* if for every pair of nodes, there is a path between them. A graph is *fully connected* or *complete* if each pair of nodes is connected by an edge. A *connected component* or simply a *component* of a graph is a subset of its nodes such that every node in the subset has a path to every other one. In the example of Fig. 8.1, the graph has one connected component. A *subgraph* is a subset of the nodes of a graph and all the edges linking those nodes. Any group of nodes can form a subgraph.

8.3 Social Network Analysis

Social network analysis processes social data structured in graphs. It involves the extraction of several characteristics and graphics to describe the main properties of the network. Some general properties of networks, such as the shape of the network degree distribution (defined bellow) or the average path length, determine the type of network, such as a *small-world* network or a *scale-free* network. A small-world network is a type of graph in which most nodes are not neighbors of one another, but most nodes can be reached from every other node in a small number of steps. This is the so-called *small-world phenomenon* which can be interpreted by the fact that strangers are linked by a short chain of acquaintances. In a small-world network, people usually form communities or small groups where everyone knows everyone else. Such communities can be seen as complete graphs. In addition, most the community members have a few relationships with people outside that community. However, some people are connected to a large number of communities. These may be celebrities and such people are considered as the *hubs* that are responsible for the small-world phenomenon. Many small-world networks are also scale-free networks. In a scale-free network the node degree distribution follows a power law (a relationship function between two quantities x and y defined as $y = x^n$, where n is a constant). The name *scale-free* comes from the fact that power laws have the same functional form at all scales, i.e., their shape does not change on multiplication by a scale factor. Thus, by definition, a scale-free network has many nodes with a very few connections and a small number of nodes with many connections. This structure is typical of the World Wide Web and other social networks. In the following sections, we illustrate this and other graph properties that are useful in social network analysis.

8.3.1 Basics in NetworkX

NetworkX[1] is a Python toolbox for the creation, manipulation and study of the structure, dynamics and functions of complex networks. After importing the toolbox, we can create an undirected graph with 5 nodes by adding the edges, as is done in the following code. The output is the graph in Fig. 8.1.

In [1]:
```
import networkx as nx
G = nx.Graph()
G.add_edge('A', 'B');
G.add_edge('A', 'C');
G.add_edge('B', 'D');
G.add_edge('B', 'E');
G.add_edge('D', 'E');
nx.draw_networkx(G)
```

To create a directed graph we would use `nx.DiGraph()`.

[1] https://networkit.iti.kit.edu.

8.3.2 Practical Case: Facebook Dataset

For our practical case we consider data from the Facebook network. In particular, we use the data *Social circles: Facebook*[2] from the Stanford Large Network Dataset[3] (SNAP) collection. The SNAP collection has links to a great variety of networks such as Facebook-style social networks, citation networks, Twitter networks or open communities like Live Journal. The Facebook dataset consists of a network representing friendship between Facebook users. The Facebook data was anonymized by replacing the internal Facebook identifiers for each user with a new value.

The network corresponds to an undirected and unweighted graph that contains users of Facebook (nodes) and their friendship relations (edges). The Facebook dataset is defined by an edge list in a plain text file with one edge per line.

Let us load the Facebook network and start extracting the basic information from the graph, including the numbers of nodes and edges, and the average degree:

In [2]:
```
fb = nx.read_edgelist("files/ch08/facebook_combined.txt")
fb_n, fb_k = fb.order(), fb.size()
fb_avg_deg = fb_k / fb_n
print 'Nodes: ', fb_n
print 'Edges: ', fb_k
print 'Average degree: ', fb_avg_deg
```

Out[2]:
```
Nodes: 4039
Edges: 88234
Average degree: 21
```

The Facebook dataset has a total of 4,039 users and 88,234 friendship connections, with an average degree of 21. In order to better understand the graph, let us compute the degree distribution of the graph. If the graph were directed, we would need to generate two distributions: one for the in-degree and another for the out-degree. A way to illustrate the degree distribution is by computing the histogram of degrees and plotting it, as the following code does with the output shown in Fig. 8.2:

In [3]:
```
degrees = fb.degree().values()
degree_hist = plt.hist(degrees, 100)
```

The graph in Fig. 8.2 is a power-law distribution. Thus, we can say that the Facebook network is a *scale-free network*.

Next, let us find out if the Facebook dataset contains more than one connected component (previously defined in Sect. 8.2):

In [4]:
```
print '# connected components of Facebook network: ',
      nx.number_connected_components(fb)
```

Out[4]:
```
# connected components of Facebook network: 1
```

[2] https://snap.stanford.edu/data/egonets-Facebook.html.
[3] http://snap.stanford.edu/data/.

As it can be seen, there is only one connected component in the Facebook network. Thus, the Facebook network is a connected graph (see definition in Sect. 8.2). We can try to divide the graph into different connected components, which can be potential communities (see Sect. 8.6). To do that, we can remove one node from the graph (this operation also involves removing the edges linking the node) and see if the number of connected components of the graph changes. In the following code, we prune the graph by removing node '0' (arbitrarily selected) and compute the number of connected components of the pruned version of the graph:

In [5]:
```
fb_prun = nx.read_edgelist(
    "files/ch08/facebook_combined.txt")
fb_prun.remove_node('0')
print 'Remaining nodes:', fb_prun.number_of_nodes()
print 'New # connected components:',
      nx.number_connected_components(fb_prun)
```

Out[5]:
```
Remaining nodes: 4038
New # connected components: 19
```

Now there are 19 connected components, but let us see how big the biggest is and how small the smallest is:

In [6]:
```
fb_components = nx.connected_components(fb_prun)
print 'Sizes of the connected components',
      [len(c) for c in fb_components]
```

Out[6]:
```
Sizes of the connected components [4015, 1, 3, 2, 2, 1, 1, 1,
1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1]
```

This simple example shows that removing a node splits the graph into multiple components. You can see that there is one large connected component and the rest are almost all isolated nodes. The isolated nodes in the pruned graph were only

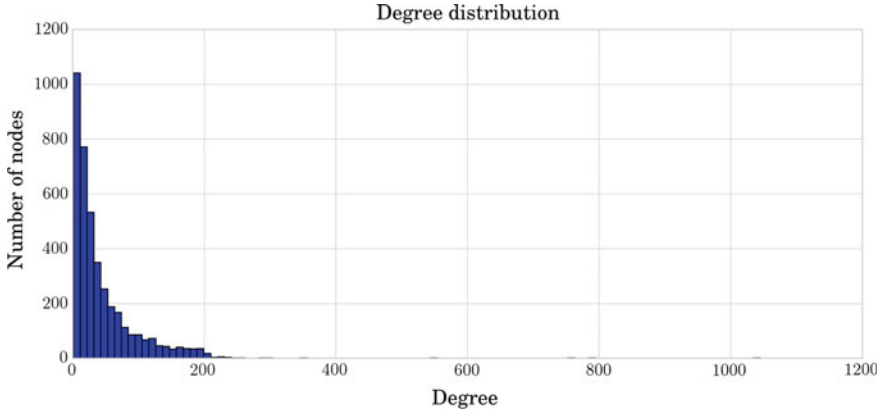


Fig. 8.2 Degree histogram distribution

connected to node '0' in the original graph and when that node was removed they were converted into connected components of size 1. These nodes, only connected to one neighbor, are probably not important nodes in the structure of the graph. We can generalize the analysis by studying the *centrality* of the nodes. The next section is devoted to explore this concept.

8.4 Centrality

The centrality of a node measures its relative importance within the graph. In this section we focus on undirected graphs. Centrality concepts were first developed in social network analysis. The first studies indicated that central nodes are probably more influential, have greater access to information, and can communicate their opinions to others more efficiently [1]. Thus, the applications of centrality concepts in a social network include identifying the most influential people, the most informed people, or the most communicative people. In practice, what centrality means will depend on the application and the meaning of the entities represented as nodes in the data and the connections between those nodes. Various measures of the centrality of a node have been proposed. We present four of the best-known measures: *degree centrality*, *betweenness centrality*, *closeness centrality*, and *eigenvector centrality*.

Degree centrality is defined as the number of edges of the node. So the more ties a node has, the more central the node is. To achieve a normalized degree centrality of a node, the measure is divided by the total number of graph nodes (n) without counting this particular one ($n-1$). The normalized measure provides proportions and allows us to compare it among graphs. Degree centrality is related to the capacity of a node to capture any information that is floating through the network. In social networks, connections are associated with positive aspects such as knowledge or friendship.

Betweenness centrality quantifies the number of times a node is crossed along the shortest path/s between any other pair of nodes. For the normalized measure this number is divided by the total number of shortest paths for every pair of nodes. Intuitively, if we think of a public bus transportation network, the bus stop (node) with the highest betweenness has the most traffic. In social networks, a person with high betweenness has more power in the sense that more people depend on him/her to make connections with other people or to access information from other people. Comparing this measure with degree centrality, we can say that degree centrality depends only on the node's neighbors; thus, it is more local than the betweenness centrality, which depends on the connection properties of every pair of nodes in the graph, except pairs with the node in question itself. The equivalent measure exists for edges. The betweenness centrality of an edge is the proportion of the shortest paths between all node pairs which pass through it.

Closeness centrality tries to quantify the position a node occupies in the network based on a distance calculation. The distance metric used between a pair of nodes is defined by the length of its shortest path. The closeness of a node is inversely proportional to the length of the average shortest path between that node and all the

other nodes in the graph. In this case, we interpret a central node as being close to, and able to communicate quickly with, the other nodes in a social network.

Eigenvector centrality defines a relative score for a node based on its connections and considering that connections from high centrality nodes contribute more to the score of the node than connections from low centrality nodes. It is a measure of the influence of a node in a network, in the following sense: it measures the extent to which a node is connected to influential nodes. Accordingly, an important node is connected to important neighbors.

Let us illustrate the centrality measures with an example. In Fig. 8.3, we show an undirected star graph with $n = 8$ nodes. Node C is obviously important, since it can exchange information with more nodes than the others. The degree centrality measures this idea. In this star network, node C has a degree centrality of 7 or 1 if we consider the normalized measure, whereas all other nodes have a degree of 1 or 1/7 if we consider the normalized measure. Another reason why node C is more important than the others in this star network is that it lies between each of the other pairs of nodes, and no other node lies between C and any other node. If node C wants to contact F, C can do it directly; whereas if node F wants to contact B, it must go through C. This gives node C the capacity to broke/prevent contact among other nodes and to isolate nodes from information. The betweenness centrality is underneath this idea. In this example, the betweenness centrality of the node C is 28, computed as $(n-1)(n-2)/2$, while the rest of nodes have a betweenness of 0. The final reason why we can say node C is superior in the star network is because C is closer to more nodes than any other node is. In the example, node C is at a distance of 1 from all other 7 nodes and each other node is at a distance 2 from all other nodes, except C. So, node C has closeness centrality of 1/7, while the rest of nodes have a closeness of 1/13. The normalized measures, computed by dividing by $n-1$, are 1 for C and 7/13 for the other nodes.

An important concept in social network analysis is that of a *hub* node, which is defined as a node with high degree centrality and betweenness centrality. When a hub governs a very centralized network, the network can be easily fragmented by removing that hub.

Coming back to the Facebook example, let us compute the degree centrality of Facebook graph nodes. In the code below we show the user identifier of the 10 most central nodes together with their normalized degree centrality measure. We also show the degree histogram to extract some more information from the shape of the distribution. It might be useful to represent distributions using logarithmic scale. We

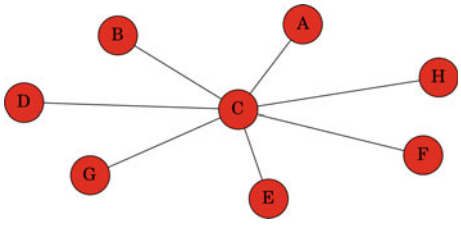


Fig. 8.3 Star graph example

do that with the `matplotlib.loglog()` function. Figure 8.4 shows the degree centrality histogram in linear and logarithmic scales as computed in the box bellow.

In [7]:
```
degree_cent_fb = nx.degree_centrality(fb)
print 'Facebook degree centrality: ',
      sorted(degree_cent_fb.items(),
             key = lambda x: x[1],
             reverse = True)[:10]
degree_hist = plt.hist(list(degree_cent_fb.values()), 100)
plt.loglog(degree_hist[1][1:],
           degree_hist[0], 'b', marker = 'o')
```

Out[7]:
```
Facebook degree centrality: [(u'107', 0.258791480931154),
(u'1684', 0.1961367013372957), (u'1912', 0.18697374938088163),
(u'3437', 0.13546310054482416), (u'0', 0.08593363051015354),
(u'2543', 0.07280832095096582), (u'2347', 0.07206537890044576),
(u'1888', 0.0629024269440317), (u'1800', 0.06067360079247152),
(u'1663', 0.058197127290737984)]
```

The previous plots show us that there is an interesting (large) set of nodes which corresponds to low degrees. The representation using a logarithmic scale (right-hand graphic in Fig. 8.4) is useful to distinguish the members of this set of nodes, which are clearly visible as a straight line at low values for the x-axis (upper left-hand part of the logarithmic plot). We can conclude that most of the nodes in the graph have low degree centrality; only a few of them have high degree centrality. These latter nodes can be properly seen as the points in the bottom right-hand part of the logarithmic plot.

The next code computes the betweenness, closeness, and eigenvector centrality and prints the top 10 central nodes for each measure.

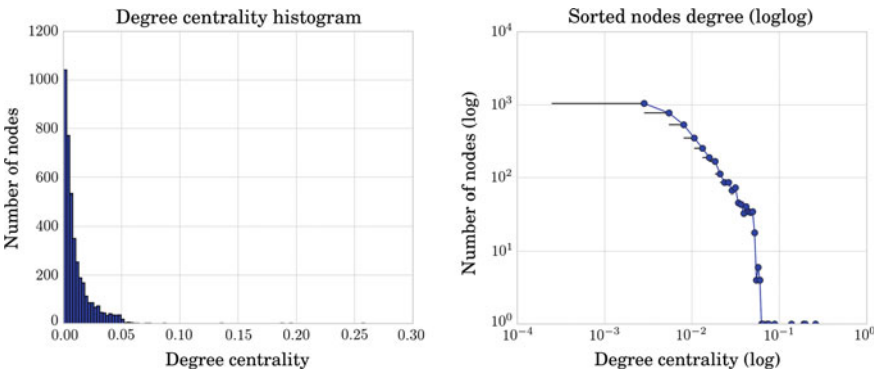


Fig. 8.4 Degree centrality histogram shown using a linear scale (*left*) and a log scale for both the x- and y-axis (*right*)

In [8]:
```
betweenness_fb = nx.betweenness_centrality(fb)
closeness_fb = nx.closeness_centrality(fb)
eigencentrality_fb = nx.eigenvector_centrality(fb)
print 'Facebook betweenness centrality:',
      sorted(betweenness_fb.items(),
             key = lambda x: x[1],
             reverse = True)[:10]
print 'Facebook closeness centrality:',
      sorted(closeness_fb.items(),
             key = lambda x: x[1],
             reverse = True)[:10]
print 'Facebook eigenvector centrality:',
      sorted(eigencentrality_fb.items(),
             key = lambda x: x[1],
             reverse = True)[:10]
```

Out[8]:
```
Facebook betweenness centrality: [(u'107', 0.4805180785560141),
(u'1684', 0.33779744973019843), (u'3437', 0.23611535735892616),
(u'1912', 0.2292953395868727), (u'1085', 0.1490150921166526),
(u'0', 0.1463059214744276), (u'698', 0.11533045020560861),
(u'567', 0.09631033121856114), (u'58', 0.08436020590796521),
(u'428', 0.06430906239323908)]
```

Out[8]:
```
Facebook closeness centrality: [(u'107', 0.45969945355191255),
(u'58', 0.3974018305284913), (u'428', 0.3948371956585509),
(u'563', 0.3939127889961955), (u'1684', 0.39360561458231796),
(u'171', 0.37049270575282134), (u'348', 0.36991572004397216),
(u'483', 0.3698479575013739), (u'414', 0.3695433330282786),
(u'376', 0.36655773420479304)]
Facebook eigenvector centrality: [(u'1912', 0.09540688873596524),
(u'2266', 0.08698328226321951), (u'2206', 0.08605240174265624),
(u'2233', 0.08517341350597836), (u'2464', 0.08427878364685948),
(u'2142', 0.08419312450068105), (u'2218', 0.08415574433673866),
(u'2078', 0.08413617905810111), (u'2123', 0.08367142125897363),
(u'1993', 0.08353243711860482)]
```

As can be seen in the previous results, each measure gives a different ordering of the nodes. The node '107' is the most central node for degree (see box `Out [7]`), betweenness, and closeness centrality, while it is not among the 10 most central nodes for eigenvector centrality. The second most central node is different for closeness and eigenvector centralities; while the third most central node is different for all four centrality measures.

Another interesting measure is the *current flow betweenness centrality*, also called *random walk betweenness centrality*, of a node. It can be defined as the probability of passing through the node in question on a random walk starting and ending at some node. In this way, the betweenness is not computed as a function of shortest paths, but of all paths. This makes sense for some social networks where messages may get to their final destination not by the shortest path, but by a random path, as in the case of gossip floating through a social network for example.

Computing the current flow betweenness centrality can take a while, so we will work with a trimmed Facebook network instead of the original one. In fact, we can

pose the question: What happen if we only consider the graph nodes with more than the average degree of the network (21)? We can trim the graph using degree centrality values. To do this, in the next code, we define a function to trim the graph based on the degree centrality of the graph nodes. We set the threshold to 21 connections:

In [9]:
```
def trim_degree_centrality(graph, degree = 0.01):
    g = graph.copy()
    d = nx.degree_centrality(g)
    for n in g.nodes():
        if d[n] <= degree:
            g.remove_node(n)
    return g
thr = 21.0/(fb.order() - 1.0)

print 'Degree centrality threshold:', thr

fb_trimmed = trim_degree_centrality(fb, degree = thr)
print 'Remaining # nodes:', len(fb_trimmed)
```

Out[9]:
```
Degree centrality threshold: 0.00520059435364
Remaining # nodes: 2226
```

The new graph is much smaller; we have removed almost half of the nodes (we have moved from 4,039 to 2,226 nodes).

The current flow betweenness centrality measure needs connected graphs, as does any betweenness centrality measure, so we should first extract a connected component from the trimmed Facebook network and then compute the measure:

In [10]:
```
fb_subgraph = list(nx.connected_component_subgraphs(
    fb_trimed))
print '# subgraphs found:', size(fb_subgraph)
print '# nodes in the first subgraph:',
      len(fb_subgraph[0])
betweenness = nx.betweenness_centrality(fb_subgraph[0])
print 'Trimmed FB betweenness: ',
      sorted(betweenness.items(), key = lambda x: x[1],
             reverse = True)[:10]
current_flow = nx.current_flow_betweenness_centrality(
    fb_subgraph[0])
print 'Trimmed FB current flow betweenness:',
      sorted(current_flow.items(), key = lambda x: x[1],
      reverse = True)[:10]
```

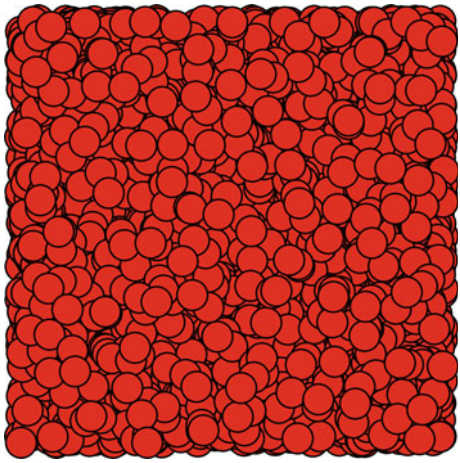


Fig. 8.5 The Facebook network with a random layout

```
Out[10]: # subgraphs found: 2
# nodes in the first subgraph: 2225
Trimmed FB betweenness: [(u'107', 0.5469164906683255),
(u'1684', 0.3133966633778371), (u'1912', 0.19965597457246995),
(u'3437', 0.13002843874261014), (u'1577', 0.1274607407928195),
(u'1085', 0.11517250980098293), (u'1718', 0.08916631761105698),
(u'428', 0.0638271827912378), (u'1465', 0.057995900747731755),
(u'567', 0.05414376521577943)]
Trimmed FB current flow betweenness: [(u'107',
0.2858892136334576), (u'1718', 0.2678396761785764), (u'1684',
0.1585162194931393), (u'1085', 0.1572155780323929), (u'1405',
0.1253563113363113), (u'3437', 0.10482568101478178), (u'1912',
0.09369897700970155), (u'1577', 0.08897207040045449), (u'136',
0.07052866082249776), (u'1505', 0.06152347046861114)]
```

As can be seen, there are similarities in the 10 most central nodes for the betweenness and current flow betweenness centralities. In particular, seven up to ten are the same nodes, even if they are differently ordered.

8.4.1 Drawing Centrality in Graphs

In this section we focus on graph visualization, which can help in the network data understanding and usability.

The visualization of a network with a large amount of nodes is a complex task. Different layouts can be used to try to build a proper visualization. For instance, we can draw the Facebook graph using the random layout (`nx.random_layout`), but this is a bad option, as can be seen in Fig. 8.5. Other alternatives can be more useful. In the box below, we use the *Spring* layout, as it is used in the default function (`nx.draw`), but with more iterations. The function `nx.spring_layout` returns the position of the nodes using the Fruchterman–Reingold force-directed algorithm.

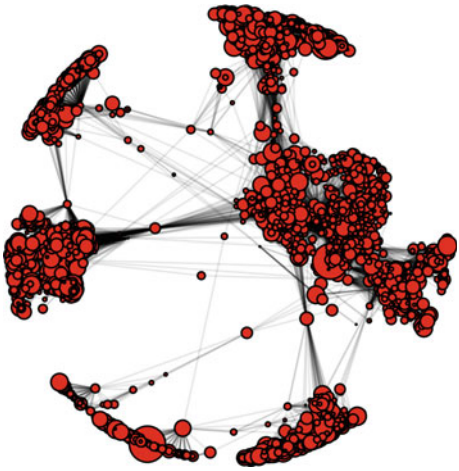


Fig. 8.6 The Facebook network drawn using the Spring layout and degree centrality to define the node size

This algorithm distributes the graph nodes in such a way that all the edges are more or less equally long and they cross themselves as few times as possible. Moreover, we can change the size of the nodes to that defined by their degree centrality. As can be seen in the code, the degree centrality is normalized to values between 0 and 1, and multiplied by a constant to make the sizes appropriate for the format of the figure:

In [11]:
```
pos_fb = nx.spring_layout(fb,iterations = 1000)

nsize = np.array([v for v in degree_cent_fb.values()])

nsize = 500*(nsize - min(nsize))/(max(nsize) - min(nsize))

nodes = nx.draw_networkx_nodes(fb, pos = pos_fb,
                               node_size = nsize)
edges = nx.draw_networkx_edges(fb, pos = pos_fb,
                               alpha = .1)
```

The resulting graph visualization is shown in Fig. 8.6. This illustration allows us to understand the network better. Now we can distinguish several groups of nodes or "communities" clearly in the graph. Moreover, the larger nodes are the more central nodes, which are highly connected of the Facebook graph.

We can also use the betweenness centrality to define the size of the nodes. In this way, we obtain a new illustration stressing the nodes with higher betweenness, which are those with a large influence on the transfer of information through the network. The new graph is shown in Fig. 8.7. As expected, the central nodes are now those connecting the different communities.

Generally different centrality metrics will be positively correlated, but when they are not, there is probably something interesting about the network nodes. For instance, if you can spot nodes with high betweenness but relatively low degree, these are the nodes with few links but which are crucial for network flow. We can also look for

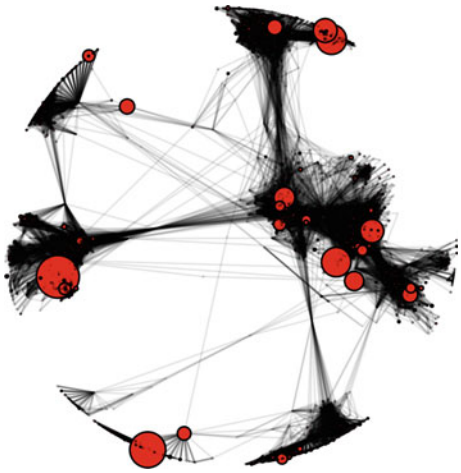


Fig. 8.7 The Facebook network drawn using the Spring layout and betweenness centrality to define the node size

the opposite effect: nodes with high degree but relatively low betweenness. These nodes are those with redundant communication.

Changing the centrality measure to closeness and eigenvector, we obtain the graphs in Figs. 8.8 and 8.9, respectively. As can be seen, the central nodes are also different for these measures. With this or other visualizations you will be able to discern different types of nodes. You can probably see nodes with high closeness centrality but low degree; these are essential nodes linked to a few important or active nodes. If the opposite occurs, if there are nodes with high degree centrality but low closeness, these can be interpreted as nodes embedded in a community that is far removed from the rest of the network.

In other examples of social networks, you could find nodes with high closeness centrality but low betweenness; these are nodes near many people, but since there may be multiple paths in the network, they are not the only ones to be near many people. Finally, it is usually difficult to find nodes with high betweenness but low closeness, since this would mean that the node in question monopolized the links from a small number of people to many others.

8.4.2 PageRank

PageRank is an algorithm related to the concept of eigenvector centrality in directed graphs. It is used to rate webpages objectively and effectively measure the attention devoted to them. PageRank was invented by Larry Page and Sergey Brin, and became a Google trademark in 1998 [2].

Assigning the importance of a webpage is a subjective task, which depends on the interests and knowledge of the persons that browse the webpages. However, there are ways to objectively rank the relative importance of webpages.

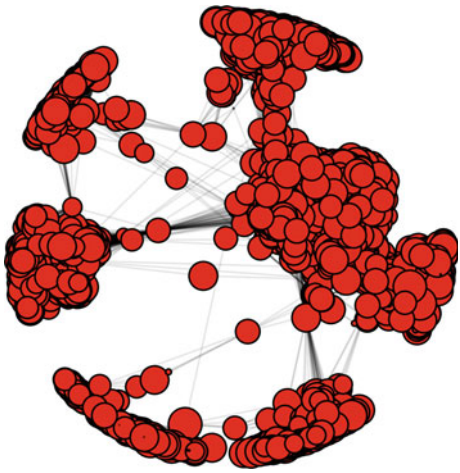


Fig. 8.8 The Facebook network drawn using the Spring layout and closeness centrality to define the node size

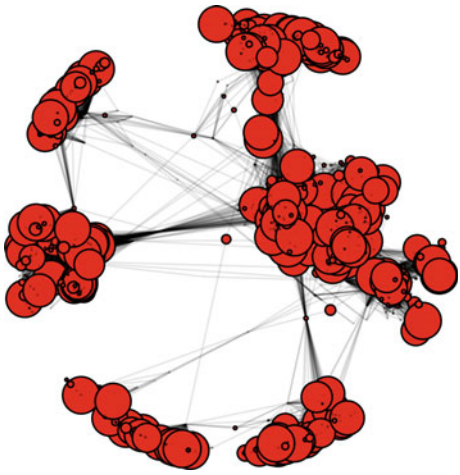


Fig. 8.9 The Facebook network drawn using the Spring layout and eigenvector centrality to define the node size

We consider the directed graph formed by nodes corresponding to the webpages and edges corresponding to the hyperlinks. Intuitively, a hyperlink to a page counts as a vote of support and a page has a high rank if the sum of the ranks of its incoming edges is high. This considers both cases when a page has many incoming links and when a page has a few highly ranked incoming links. Nowadays, a variant of the algorithm is used by Google. It does not only use information on the number of edges pointing into and out of a website, but uses many more variables.

We can describe the PageRank algorithm from a probabilistic point of view. The rank of page P_i is the probability that a surfer on the Internet who starts visiting a random page and follows links, visits the page P_i. With more details, we consider that the weights assigned to the edges of a network by its transition matrix, M, are the probabilities that the surfer goes from one webpage to another. We can understand the

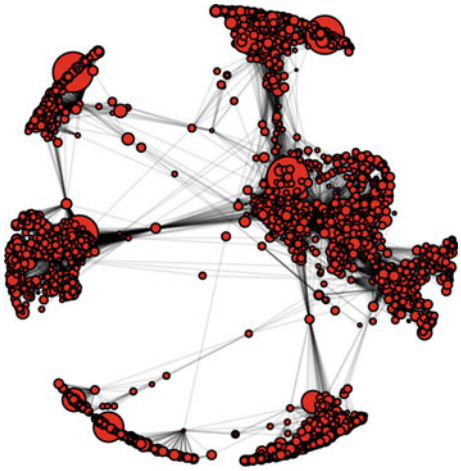


Fig. 8.10 The Facebook network drawn using the Spring layout and PageRank to define the node size

rank computation as a random walk through the network. We start with an initial equal probability for each page: $v_0 = (\frac{1}{n}, \ldots, \frac{1}{n})$, where n is the number of nodes. Then we can compute the probability that each page is visited after one step by applying the transition matrix: $v_1 = Mv$. The probability that each page will be visited after k steps is given by $v_k = M^k a$. After several steps, the sequence converges to a unique probabilistic vector a^* which is the *PageRank vector*. The i-th element of this vector is the probability that at each moment the surfer visits page P_i. We need a nonambiguous definition of the rank of a page for any directed web graph. However, in the Internet, we can expect to find pages that do not contain outgoing links and this configuration can lead to certain problems to the explained procedure. In order to overcome this problem, the algorithm fixes a positive constant p between 0 and 1 (a typical value for p is 0.85) and redefines the transition matrix of the graph by $R = (1 - p)\, M + p\, B$, where $B = \frac{1}{n} I$, and I is the identity matrix. Therefore, a node with no outgoing edges has probability $\frac{p}{n}$ of moving to any other node.

Let us compute the PageRank vector of the Facebook network and use it to define the size of the nodes, as was done in box `In [11]`.

In [12]:
```
pr = nx.pagerank(fb, alpha = 0.85)
nsize = np.array([v for v in pr.values()])
nsize = 500*(nsize  - min(nsize))/(max(nsize) - min(nsize))
nodes = nx.draw_networkx_nodes(fb,
                               pos = pos_fb,
                               node_size = nsize)
edges = nx.draw_networkx_edges(fb,
                               pos = pos_fb,
                               alpha = .1)
```

The code above outputs the graph in Fig. 8.10, that emphasizes some of the nodes with high PageRank. Looking the graph carefully one can realize that there is one large node per community.

8.5 Ego-Networks

Ego-networks are subnetworks of neighbors that are centered on a certain node. In Facebook and LinkedIn, these are described as “your network". Every person in an ego-network has her/his own ego-network and can only access the nodes in it. All ego-networks interlock to form the whole social network. The ego-network definition depends on the network distance considered. In the basic case, a distance of 1, a link means that person A is a friends of person B, a distance of 2 means that a person, C, is a friend of a friend of A, and a distance of 3 means that another person, D, is a friend of a friend of a friend of A. Knowing the size of an ego-network is important when it comes to understanding the reach of the information that a person can transmit or have access to. Figure 8.11 shows an example of an ego-network. The blue node is the *ego*, while the rest of the nodes are red.

Our Facebook network was manually labeled by users into a set of 10 ego-networks. The public dataset includes the information of these 10 manually defined ego-networks. In particular, we have available the list of the 10 ego nodes: ‘0’, ‘107’, ‘348’, ‘414’, ‘686’, ‘1684’, ‘1912’, ‘3437’, ‘3980’ and their connections. These ego-networks are interconnected to form the fully connected graph we have been analyzing in previous sections.

In Sect. 8.4 we saw that node ‘107’ is the most central node of the Facebook network for three of the four centrality measures computed. So, let us extract the ego-networks of the popular node ‘107’ with a distance of 1 and 2, and compute their sizes. NetworkX has a function devoted to this task:

In [13]:
```
ego_107 = nx.ego_graph(fb, '107')
print '# nodes of ego graph 107:',
      len(ego_107)
print '# nodes of ego graph 107 with radius up to 2:',
      len(nx.ego_graph(fb, '107', radius = 2))
```

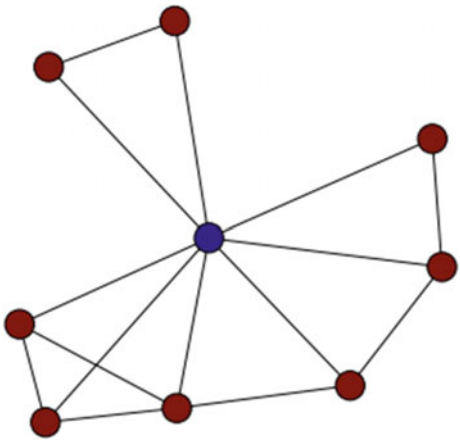


Fig. 8.11 Example of an ego-network. The *blue* node is the ego

Out[13]:
```
# nodes of ego graph 107: 1046
# nodes of ego graph 107 with radius up to 2: 2687
```

The ego-network size is 1,046 with a distance of 1, but when we expand the distance to 2, node '107' is able to reach up to 2,687 nodes. That is quite a large ego-network, containing more than half of the total number of nodes.

Since the dataset also provides the previously labeled ego-networks, we can compute the actual size of the ego-network following the user labeling. We can access the ego-networks by simply importing `os.path` and reading the edge list corresponding, for instance, to node '107', as in the following code:

In [14]:
```
import os.path
ego_id = 107
G_107 = nx.read_edgelist(
    os.path.join('files/ch08/facebook',
                 '{0}.edges'.format(ego_id)),
    nodetype = int)
print 'Nodes of the ego graph 107:', len(G_107)
```

Out[14]:
```
Nodes of the ego graph 107: 1034
```

As can be seen, the size of the previously defined ego-network of node '107' is slightly different from the ego-network automatically computed using NetworkX. This is due to the fact that the manual labeling is not necessarily referred to the subgraph of neighbors at a distance of 1.

We can now answer some other questions about the structure of the Facebook network and compare the 10 different ego-networks among them. First, we can compute which the most densely connected ego-network is from the total of 10. To do that, in the code below, we compute the number of edges in every ego-network and select the network with the maximum number:

In [15]:
```
ego_ids = ( 0, 107, 348,
            414, 686, 698,
            1684, 1912, 3437, 3980)
ego_sizes = zeros((10, 1))
i = 0
# Fill the 'ego_sizes' vector with the size (# edges) of the
    10 ego-networks in egoids
for id in ego_ids :
    G = nx.read_edgelist(
        os.path.join('files/ch08/facebook',
                     '{0}.edges'.format(id)),
        nodetype = int)
    ego_sizes[i] = G.size()
    i = i + 1
[i_max,j] = (ego_sizes == ego_sizes.max()).nonzero()
ego_max = ego_ids[i_max]
print 'The most densely connected ego-network is \
      that of node:', ego_max

G = nx.read_edgelist(
    os.path.join('files/ch08/facebook',
                 '{0}.edges'.format(ego_max)),
    nodetype = int)
print 'Nodes: ', G.order()
print 'Edges: ', G.size()
print 'Average degree: ', G_k / G_n
```

Out[15]:
```
The most densely connected ego-network is that of node: 1912
Nodes: 747
Edges: 30025
Average degree: 40
```

The most densely connected ego-network is that of node '1912', which has an average degree of 40. We can also compute which is the largest (in number of nodes) ego-network, changing the measure of sizes from `G.size()` by `G.order()`. In this case, we obtain that the largest ego-network is that of node '107', which has 1,034 nodes and an average degree of 25.

Next let us work out how much intersection exists between the ego-networks in the Facebook network. To do this, in the code below, we add a field 'egonet' for every node and store an array with the ego-networks the node belongs to. Then, having the length of these arrays, we compute the number of nodes that belong to 1, 2, 3, 4 and more than 4 ego-networks:

In [16]:
```
# Add a field 'egonet' to the nodes of the whole facebook
    network.
# Default value egonet = [], meaning that this node does not
     belong to any ego-netowrk
for i in fb.nodes() :
    fb.node[str(i)]['egonet'] = []

# Fill the 'egonet' field with one of the 10 ego values in
    ego_ids:
for id in ego_ids :
    G = nx.read_edgelist(
        os.path.join('files/ch08/facebook',
                     '{0}.edges'.format(id)),
        nodetype = int)
    print id
    for n in G.nodes() :
        if (fb.node[str(n)]['egonet'] == []) :
            fb.node[str(n)]['egonet'] = [id]
        else :
            fb.node[str(n)]['egonet'].append(id)

# Compute the intersections:
S = [len(x['egonet']) for x in fb.node.values()]
print '# nodes into 0 ego-network: ', sum(equal(S, 0))
print '# nodes into 1 ego-network: ', sum(equal(S, 1))
print '# nodes into 2 ego-network: ', sum(equal(S, 2))
print '# nodes into 3 ego-network: ', sum(equal(S, 3))
print '# nodes into 4 ego-network: ', sum(equal(S, 4))
print '# nodes into more than 4 ego-network: ',\
      sum(greater(S, 4))
```

Out[16]:
```
# nodes into 0 ego-network: 80
# nodes into 1 ego-network: 3844
# nodes into 2 ego-network: 102
# nodes into 3 ego-network: 11
# nodes into 4 ego-network: 2
# nodes into more than 4 ego-network: 0
```

As can be seen, there is an intersection between the ego-networks in the Facebook network, since some of the nodes belong to more than 1 and up to 4 ego-networks simultaneously.

We can also try to visualize the different ego-networks. In the following code, we draw the ego-networks using different colors on the whole Facebook network and we obtain the graph in Fig. 8.12. As can be seen, the ego-networks clearly form groups of nodes that can be seen as communities.

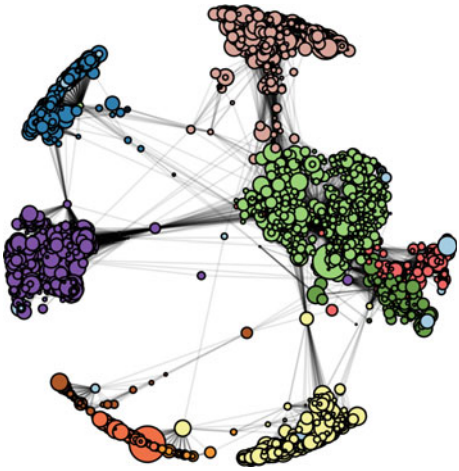


Fig. 8.12 The Facebook network drawn using the Spring layout and different colors to separate the ego-networks

In [17]:

```
# Add a field 'egocolor' to the nodes of the whole facebook
network.
# Default value egocolor r =0, meaning that this node
does not belong to any ego-netowrk for i in fb.nodes() :
    fb.node[str(i)]['egocolor'] = 0

# Fill the 'egocolor' field with a different color number
    for each ego-network in ego_ids:
idColor = 1
for id in ego_ids :
    G = nx.read_edgelist(
        os.path.join('files/ch08/facebook',
                     '{0}.edges'.format(id)),
        nodetype = int)
    for n in G.nodes() :
        fb.node[str(n)]['egocolor'] = idColor
    idColor += 1

colors = [x['egocolor'] for x in fb.node.values()]

nsize = np.array([v for v in degree_cent_fb.values()])

nsize = 500*(nsize - min(nsize))/(max(nsize)- min(nsize))

nodes = nx.draw_networkx_nodes(
    fb, pos = pos_fb,
    cmap = plt.get_cmap('Paired'),
    node_color = colors,
    node_size = nsize,
    with_labels = False)
edges=nx.draw_networkx_edges(fb, pos = pos_fb, alpha = .1)
```

However, the graph in Fig. 8.12 does not illustrate how much overlap is there between the ego-networks. To do that, we can visualize the intersection between ego-networks using a *Venn* or an *Euler* diagram. Both diagrams are useful in order to see how networks are related. Figure 8.13 shows the Venn diagram of the Facebook network. This powerful and complex graph cannot be easily built in Python tool-

Fig. 8.13 Venn diagram. The area is weighted according to the number of friends in each ego-network and the intersection between ego-networks is related to the number of common users

boxes like NetworkX or Matplotlib. In order to create it, we have used a JavaScript visualization library called D3.JS.[4]

8.6 Community Detection

A community in a network can be seen as a set of nodes of the network that is densely connected internally. The detection of communities in a network is a difficult task since the number and sizes of communities are usually unknown [3].

Several methods for community detection have been developed. Here, we apply one of the methods to automatically extract communities from the Facebook network. We import the `Community` toolbox[5] which implements the Louvain method for community detection. In the code below, we compute the best partition and plot the resulting communities in the whole Facebook network with different colors, as we did in box `In [17]`. The resulting graph is shown in Fig. 8.14.

In [18]:
```
import community partition = community.best_partition(fb)
    print "#
communities found:", max(partition.values()) colors2 =
[partition.get(node) for node in fb.nodes()] nsize = np.
    array([v
for v in degree_cent_fb.values()]) nsize = 500*(nsize  -
min(nsize))/(max(nsize)- min(nsize)) nodes =
nx.draw_networkx_nodes(
    fb, pos = pos_fb,
    cmap = plt.get_cmap('Paired'),
    node_color = colors2,
    node_size = nsize,
    with_labels = False)
edges = nx.draw_networkx_edges(fb, pos = pos_fb, alpha = .1)
```

[4] https://d3js.org.
[5] http://perso.crans.org/aynaud/communities/.

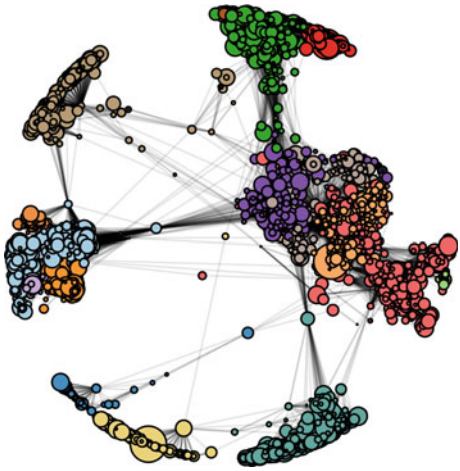


Fig. 8.14 The Facebook network drawn using the Spring layout and different colors to separate the communities found

```
Out[18]: # communities found: 15
```

As can be seen, the 15 communities found automatically are similar to the 10 ego-networks loaded from the dataset (Fig. 8.12). However, some of the 10 ego-networks are subdivided into several communities now. This discrepancy can be due to the fact that the ego-networks are manually annotated based on more properties of the nodes, whereas communities are extracted based only on the graph information.

8.7 Conclusions

In this chapter, we have introduced network analysis and a Python toolbox (NetworkX) that is useful for this analysis. We have shown how network analysis allows us to extract properties from the data that would be hard to discover by other means. Some of these properties are basic concepts in social network analysis, such as centrality measures which return the importance of the nodes in the network or ego-networks which allows us to study the reach of the information a node can transmit or have access to. The different concepts have been practically illustrated by a practical case dealing with a Facebook network. In this practical case, we have resolved several issues, such as finding the most representative members of the network in terms of the most "connected", the most "circulated", the "closest", or the most "accessible" nodes to the others. We have presented useful ways of extracting basic properties of the Facebook network, and studying its ego-networks and communities,

as well as comparing them quantitatively and qualitatively. We have also proposed several visualizations of the graph to represent several measures and to emphasize the important nodes with different meanings.

Acknowledgements This chapter was co-written by Laura Igual and Santi Seguí.

References

1. N. Friedkin, *Structural bases of interpersonal influence in groups: A Longitudinal Case Study*. American Sociological Review 58(6):861 1993
2. L. Page, S. Brin, R. Motwani, and T. Winograd, *The PageRank citation ranking: Bringing order to the Web*. 1999
3. V. D. Blondel, J.-L. Guillaume, R. Lambiotte, R. Lefebvre, *Fast unfolding of communities in large networks*. Journal of Statistical Mechanics: Theory and Experiment. 2008(10)

Recommender Systems

9

9.1 Introduction

In this chapter, we will see what are recommender systems, how they work, and how they can be implemented. We will also see the different paradigms of recommender systems based on the information they use, as well as the output they produce. We will consider typical questions that companies like Netflix or Amazon include in their products: Which movie should I rent? Which TV should I buy? and we will give some insights in order to deal with more complex questions: Which is the best place for me and my family to travel to?

So, the first question we should answer: What is a recommender system? It can be defined as a tool designed to interact with large and complex information spaces, and to provide information or items that are likely to be of interest to the user, in an automated fashion. We refer to complex information space to the set of items, and its characteristics, which the system recommends to the user, i.e., books, movies, or city trips.

Nowadays, recommender systems are extremely common, and are applied in a large variety of applications. Perhaps one of the most popular types are the movie recommender systems in applications used by companies such as Netflix, and the music recommenders in Pandora or Spotify, as well as any kind of product recommendation from Amazon.com. However, the truth is that recommender systems are present in a huge variety of applications, such as movies, music, news, books, research papers, search queries, social tags, and products in general, but they are also present in more sophisticated products where personalization is critical, like recommender systems for restaurants, financial services, life assurance, online dating, and Twitter followers.

Why and When Do We Need a Recommender System?

In this new era, where the quantity of information is huge, recommender systems are extremely useful in several domains. People are not able to be experts in all

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_9

these domains in which they are users, and they do not have enough time to spend looking for the perfect TV or book to buy. Particularly, recommender systems are really interesting when dealing with the following issues:

- solutions for large amounts of good data;
- reduction of cognitive load on the user;
- allowing new items to be revealed to users.

9.2 How Do Recommender Systems Work?

There are several different ways to build a recommender system. However, most of them take one of two basic approaches: *content-based filtering* (CBF) or *collaborative filtering* (CF).

9.2.1 Content-Based Filtering

CBF methods are constructed behind the following paradigm: "Show me more of the same what I've liked". So, this approach will recommend items which are similar to those the user liked before and the recommendations are based on descriptions of items and a profile of the user's preferences. The computation of the similarity between items is the most important part of these methods and it is based on the content of the items themselves. As the content of the item can be very diverse, and it usually depends on the kind of items the system recommends, a range of sophisticated algorithms are usually used to abstract features from items. When dealing with textual information such as books or news, a widely used algorithm is *tf–idf* representation. The term tf–idf refers to frequency–inverse document frequency, it is a numerical statistic that measures how important a word is to a document in a collection or corpus.

An interesting content-based filtering system is Pandora.[1] This music recommender system uses up to 400 songs and artist properties in order to find similar songs to recommend to the original seed. These properties are a subset of the features studied by musicologists in The Music Genome Project who describe a song in terms of its melody, harmony, rhythm, and instrumentation as well as its form and the vocal performance.

[1] http://www.pandora.com/.

9.2.2 Collaborative Filtering

CF methods are constructed behind the following paradigm: "Tell me what's popular among my like-minded users". This is really intuitive paradigm since it is really similar of what people use to do: ask or look at the preferences of the people they trust. An important working hypothesis behind these kind of recommenders is that similar users tend to like similar items. In order to do so, these approaches are based on collecting and analyzing a large number of data related to the behavior, activities, or tastes of users, and predicting what users will like based on their similarity to other users. One of the main advantages of this type of system is that it does not need to "understand" what the item it recommends is.

Nowadays, these methods are extremely popular because of the simplicity and the large amount of data available from users. The main drawbacks of this kind of method is the need for a user community, as well as the *cold-start* effect for new users in the community. The cold-start problem appears when the system cannot draw any, or an optimal, inference or recommendation for the users (or items) since it has not yet obtained the sufficient information of them.

CF can be of two types: *user-based* or *item-based*.

- User-based CF works like this: Find similar users to me and recommend what they liked. In this method, given a user, U, we first find a set of other users, D, whose ratings are similar to the ratings of U and then we calculate a prediction for U.
- Item-based CF works like this: Find similar items to those that I previously liked. In item-based CF, we first build an item–item matrix that determines relationships between pairs of items; then using this matrix and data on the current user U, we infer the user's taste. Typically, this approach is used in the domain: people who buy x also buy y. This is a really popular approach used by companies like Amazon. Moreover, one of the advantages of this approach is that items usually do not change much, so its similarities can be computed offline.

9.2.3 Hybrid Recommenders

Hybrid approaches can be implemented in several ways: by making content-based and collaborative predictions separately and then combining them; by adding content-based capabilities to a collaborative approach (and vice versa); or by unifying the approaches into one model.

9.3 Modeling User Preferences

Both, CBF and CF recommender systems, require to understand the user preferences. Understanding how to model the user preference is a critical step due to the variety of sources. It is not the same when we deal with applications like the movie

recommender from Netflix, where the users rank the movies with 1 to 5 stars; or as dealing with any product recommender system from Amazon, where usually the tracking information of the purchases is used. In this case, three values can be used: 0 - not bought; 1 - viewed; 2 - bought.

The most common types of labels used to estimate the user preferences are:

- Boolean expressions (is bought?; is viewed?)
- Numerical expressions (e.g., star ranking)
- Up-Down expressions (e.g., like, neutral, or dislike)
- Weighted value expressions (e.g., number of reproductions or clicks)

In the following sections of this chapter, we only consider the numerical expression described as stars on the scale of 1 to 5.

9.4 Evaluating Recommenders

The evaluation of the recommender systems is another important step in order to assess the effectiveness of the method. When dealing with numerical labels, as the 5-star ratings, the most common way to validate a recommender system is based on their prediction value, i.e., the capacity to predict the user's choices. Standard functions such as *root mean square error* (RMSE), *precision*, *recall*, or *ROC/cost* curves have been extensively used.

However, there are several other ways to evaluate the systems. It is because metrics are entirely relevant to point of view of the person who has to evaluate it. Imagine the following three persons: (a) a marketing guy; (b) a technical system designer; and (c) a final user. It is clear that what is relevant for all of them is not the same. For a marketing guy, what is usually important is how the system helps to push the product, for the technical system designer is how efficient is the algorithm, and for the final user is if the system gives him good, or mostly cool, results. In the literature we can see two main typologies: *offline* and *online* evaluation.

We refer to evaluation as offline when a set of labeled data is obtained and then divided into two sets: a *training set* and a *test set*. The training set is used to create the model and adjust all the parameters; while the test set is used to determine selected evaluation metrics. As mentioned above, standard metrics such as RMSE, precision, and recall are extensively used, but recently other indirect functions have also started to be widely considered. Examples of these: diversity, novelty, coverage, cold-start, or serendipity, the latter is a quite popular metric that evaluates how surprising the recommendations are. For further discussion of this field, the reader is referred to [1].

We refer to evaluation as online when a set of tools is used that allows us to look at the interactions of users with the system. The most common online technique is called *A-B testing* and has the benefit of allowing evaluation of the system at the same time as users are learning, buying, or playing with the recommender system. This brings the evaluation closer to the actual working of the system and makes it really effective when the purpose of the system is to change or influence the behavior of users. In order to evaluate the test, we are interested in measuring how user behavior changes when the user is interacting with different recommender systems. Let us give an example: imagine we want to develop a music recommender system like Pandora, where your final goal is none other than for users to love your intelligent music station and spend more time listening to it. In such a situation, offline metrics like RMSE are not good enough. In this case, we are particularly interested in evaluation of the global goal of the recommender system as it is the long-term profit or user retention.

9.5 Practical Case

In this section, we will play with a real dataset to implement a movie recommender system. We will work with a user-based collaborative system with the *MovieLens* dataset.

9.5.1 MovieLens Dataset

MovieLens datasets are a collection of movie ratings produced by hundreds of users collected by the GroupLens Research Project at the University of Minnesota and released into the public domain. Several versions of this dataset can be found at the GroupLens site.[2] Figure 9.1 shows a capture of this website.

Although performance on bigger dataset is expected to be better, we will work with the smallest dataset: *MovieLens 100K Dataset*. Working with this lite version has the benefit of less computational costs, while we will also get the basic skills required on user-based recommender systems.

Once you have downloaded and unzipped the file into a directory, you can create a Pandas DataFrame with the following code:

[2] http://grouplens.org/datasets/movielens/.

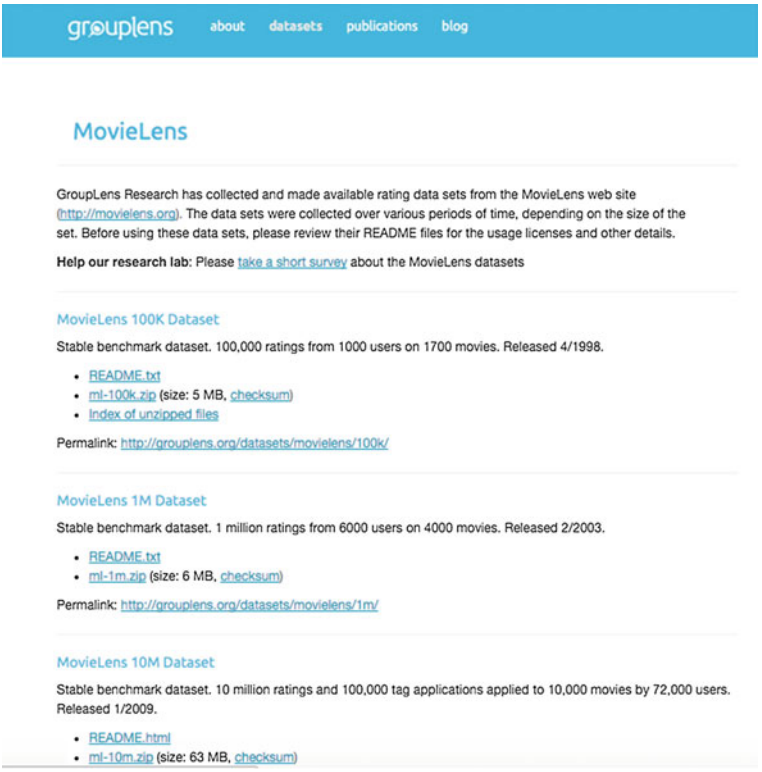


Fig. 9.1 Grouplens website

In [1]:

```
# Load user data
u_cols = [
    'user_id', 'age', 'sex',
    'occupation', 'zip_code'
    ]
users = pd.read_csv('files/ch09/ml-100k/u.user',
                    sep='|',
                    names=u_cols)

# Load movie data
r_cols = [
    'user_id', 'movie_id',
    'rating', 'unix_timestamp'
    ]
ratings = pd.read_csv('files/ch09/ml-100k/u.data',
                      sep='\t',
                      names=r_cols)

# The movie file contains columns indicating the genres of
    the movie
# We will only load the first three columns of the file with
     usecols
```

In [1]:
```
m_cols = [
    'movie_id', 'title',
    'release_date'
    ]
movies = pd.read_csv('files/ch09/ml-100k/u.item',
                     sep='|',
                     names=m_cols,
                     usecols=range(3))

# Create a DataFrame  using only the fields required
data = pd.merge(pd.merge(ratings, users), movies)
data = data[['user_id', 'title', 'movie_id', 'rating']]


print "The BD has "+ str(data.shape[0]) +" ratings"
print "The BD has ", data.user_id.nunique()," users"
print "The BD has ", data.movie_id.nunique(), " items"
print data.head()
```

Out[1]:
```
The DB has 100000 ratings
The DB has 943 different users
The DB has 1682 different items

  user_id    title    movie_id rating
0   196   Kolya (1996)    242      3
1   305   Kolya (1996)    242      5
2    6    Kolya (1996)    242      4
3   234   Kolya (1996)    242      4
4    63   Kolya (1996)    242      3
```

If you explore the dataset in detail, you will see that it consists of:

- 100,000 ratings from 943 users of 1682 movies. Ratings are from 1 to 5.
- Each user has rated at least 20 movies.
- Simple demographic info for the users (age, gender, occupation, zip).

9.5.2 User-Based Collaborative Filtering

In order to create a user-based collaborative recommender system we must define: (1) a prediction function, (2) a user similarity function, and (3) an evaluation function.

Prediction Function

The prediction function behind the user-based CF will be based on the movie ratings from similar users. So, in order to recommend a movie, p, from a set of movies, P, to a given user, a, we first need to see the set of users, B, who have already seen p. Then, we need to see the taste similarity between these users in B and user a. The most simple prediction function for a user a and movie p can be defined as follows:

$$pred(a, p) = \frac{\sum_{b \in B} sim(a, b)(r_{b,p})}{\sum_{b \in B} sim(a, b)} \tag{9.1}$$

Table 9.1 Recommender System

Critic	sim(a,b)	Rating movie1: r_{b,p_1}	$sim(a,b)(r_{b,p_1})$
Paul	0.99	3	2.97
Alice	0.38	3	1.14
Marc	0.89	4.5	4.0
Anne	0.92	3	2.77
$\sum_{b \in N} sim(a,b)(r_{b,p})$			10.87
$\sum_{b \in N} sim(a,b)$			3.18
$pred(a,p)$			3.41

where $sim(a,b)$ is the similarity between user a and user b, B is the set of users in the dataset that have already seen p and $r_{b,p}$ is the rating of p by b.

Let us give an example (see Table 9.1). Imagine the system can only recommend one movie, since the rest have already been seen by the user. So, we only want to estimate the score corresponding to that movie. The movie has been seen by Paul, Alice, Marc, and Anne and scored 3, 3, 4, and 3, respectively. Similarity between user a and Paul, Alice, Marc, and Anne has been computed "somehow" (we will see later how we can compute it) and the values are 0.99, 0.38, 0.89, and 0.92, respectively. If we follow the previous equation, the estimated score is 3.41, as seen in Table 9.1.

User Similarity Function

The computation of the similarity between users is one of the most critical steps in the CF algorithms. The basic idea behind the similarity computation between two users a and b is that we can first isolate the set P of items rated by both users, and then apply a similarity computation technique to determine the similarity.

The set of `common_movies` can be obtained with the following code:

In [2]:

```
# dataframe with the data from user 1
df_usr1 = data_train[data_train.user_id == 1]

# dataframe with the data from user 2
df_usr2 = data_train[data_train.user_id == 6]

# We first compute the set of common movies
common_mov = set(df_usr1.movie_id).intersection(
    df_usr2.movie_id)

print "\nNumber of common movies",
      len(common_mov)
```

In [2]:

```
# Sub-dataframe with only the common movies
mask = (data_user_1.movie_id.isin(common_movies))
data_user_1 = data_user_1[mask]
print data_user_1[['title', 'rating']].head()

mask = (data_user_2.movie_id.isin(common_movies))
data_user_2 = data_user_2[mask]
print data_user_2[['title', 'rating']].head()
```

Out[2]:

```
Number of common movies 11
Movies User 1
                     title                 rating
14                Kolya (1996)                5
417         Shall We Dance? (1996)            4
1306 Truth About Cats & Dogs, The (1996)      5
1618         Birdcage, The (1996)             4
3479          Men in Black (1997)             4
Movies User 2
                     title                 rating
32                Kolya (1996)                5
424         Shall We Dance? (1996)            5
1336 Truth About Cats & Dogs, The (1996)      4
1648         Birdcage, The (1996)             4
3510          Men in Black (1997)             4
```

Once the set of ratings for all movies common to the two users has been obtained, we can compute the user similarity. Some of the most common similarity functions used in CF methods are as follows:

Euclidean distance:

$$sim(a,b) = \frac{1}{1+\sqrt{\sum_{p\in P}(r_{a,p}-r_{b,p})^2}} \tag{9.2}$$

Pearson correlation:

$$sim(a,b) = \frac{\sum_{p\in P}(r_{a,p}-\bar{r_a})(r_{b,p}-\bar{r_b})}{\sqrt{\sum_{p\in P}(r_{a,p}-\bar{r_a})}\sqrt{\sum_{p\in P}(r_{b,p}-\bar{r_b})}} \tag{9.3}$$

where $\bar{r_a}$ and $\bar{r_b}$ are the mean ratings of users a and b.

Cosine distance:

$$sim(a,b) = \frac{a\cdot b}{|a|\cdot|b|} \tag{9.4}$$

Now, the question: Which function should we use? The answer is that there is no fixed recipe; but there are some issues we can take into account when choosing the proper similarity function. On the one hand, Pearson correlation usually works better than Euclidean distance since it is based more on the ranking than on the values. So, two users who usually like more the same set of items, although their rating is on different scales, will come out as similar users with Pearson correlation but not with Euclidean distance. On the other hand, when dealing with binary/unary data, i.e.,

like versus not like or buy versus not buy, instead of scalar or real data like ratings, cosine distance is usually used.

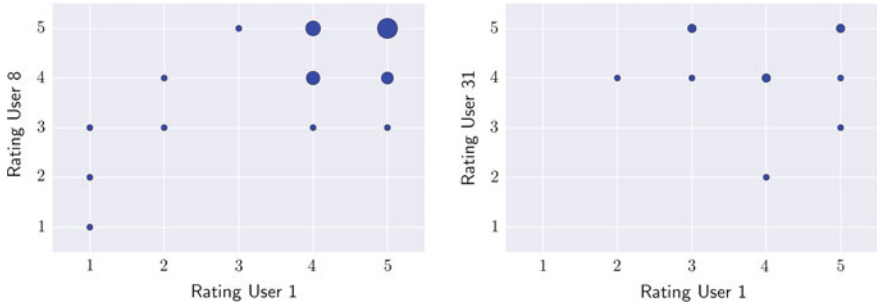
Let us define the Euclidean and Pearson functions:

In [3]:
```
from scipy.spatial.distance import Euclidean

# Similarity based on Euclidean distance for users 1-2
def SimEuclid(df,User1,User2,min_common_items=10):
    # GET MOVIES OF USER1
    mov_u1 = df[df['user_id'] == User1 ]
    # GET MOVIES OF USER2
    mov_u2 = df[df['user_id'] == User2 ]

    # FIND SHARED FILMS
    rep = pd.merge(mov_u1, mov_u2, on = 'movie_id')
    if len(rep) == 0:
        return 0
    if(len(rep) < min_common_items):
        return 0
    return 1.0 / (1.0+euclidean(rep['rating_x'],
                                rep['rating_y']))
```

In [4]:
```
from scipy.stats import pearsonr

# Similarity based on Pearson correlation for user 1-2
def SimPearson(df, User1, User2, min_common_items = 10):
    # GET MOVIES OF USER1
    mov_u1 = df[df['user_id'] == User1 ]
    # GET MOVIES OF USER2
    mov_u2 = df[df['user_id'] == User2 ]

    # FIND SHARED FILMS
    rep = pd.merge(mov_u1, mov_u2, on = 'movie_id')
    if len(rep)==0:
        return 0
    if(len(rep) < min_common_items):
        return 0
    return pearsonr(rep['rating_x'], rep['rating_y']) [0]
```

Figure 9.2 shows the correlation plots for user 1 versus user 8 and user 1 versus user 31. Each point in the plots corresponds to a different set of ratings from the two users of the same movies. The bigger the dot, the larger the set of movies rated with the corresponding values. We can observe in these plots that ratings from user 1 are more correlated with ratings from user 8 than from the user 31. However, as we can observe in the following outputs, the Euclidean similarity between user 1 and user 31 is closer than between user 1 and user 8.

In [5]:
```
print "Euclidean similarity",SimEuclid(data_train, 1, 8)
print "Pearson similarity",SimPearson(data_train, 1, 8)

print "Euclidean similarity",SimEuclid(data_train, 1, 31)
print "Pearson similarity",SimPearson(data_train, 1, 31)
```



(a) User 1 vs. 8 (b) User 1 vs. 31

Fig. 9.2 Similarity between users

```
Out[5]: Euclidean similarity 0.195194101601
        Pearson similarity 0.773097845465

        Euclidean similarity 0.240253073352
        Pearson similarity 0.272165526976
```

Evaluation

In order to validate the system, we will divide the dataset into two different sets: one called `X_train` containing 80% of the data from each user; and another called `X_test`, with the remaining 20% of the data from each user. In the following code we create a function `assign_to_set` that creates a new column in the DataFrame indicating which sample it belongs to.

In [6]:
```
def assign_to_set(df):
    sampled_ids = np.random.choice(
        df.index,
        size = np.int64(np.ceil(df.index.size * 0.2)),
        replace=False)
    df.ix[sampled_ids, 'for_testing'] = True
    return df

data['for_testing'] = False
grouped = data.groupby('user_id', group_keys = False)
              .apply(assign_to_set)
X_train = data[grouped.for_testing == False]
X_test  = data[grouped.for_testing == True]
```

The resulting `X_train` and `X_test` sets have 79619 and 20381 ratings, respectively.

Once the data is divided in these sets, we can build a model with the training set and evaluate its performance using the test set. In our case, the evaluation will be performed using the standard RMSE:

$$RMSE = \sqrt{\left(\frac{\sum(\hat{y} - y)^2}{n}\right)} \tag{9.5}$$

where y is the real rating and $\hat{y}$ is the predicted rating.

In [7]:
```
def compute_rmse(y_pred, y_true):
    """ Compute Root Mean Squared Error. """
    return np.sqrt(np.mean(np.power(y_pred - y_true, 2)))
```

Collaborative Filtering Class

We can define our recommender system with a Python class. This class consists of a constructor and two methods: `fit` and `predict`. In the `fit` method the user's similarities are computed and stored into a Python dictionary. This is a really simple method but quite expensive in terms of computation when dealing with a large dataset. We decided to show one of the most basic schemes in order to implement it. More complex algorithms can be used in order to improve the computations cost. Moreover, online strategies can be used when dealing with a really dynamic problems. In the `predict` the score for a movie and a user is estimated.

In [8]:
```
class CollaborativeFiltering:
    """ CF using a custom sim(u,u'). """
    def __init__(self, df, similarity = SimPearson):
        """ Constructor """
        self.sim_method = similarity
        self.df = df
        self.sim = pd.DataFrame(
        np.sum([0]), columns = df.user_id.unique(),
        index = df.user_id.unique())

    def fit(self):
        """ Prepare data structures for estimation.
            Similarity matrix for users """
        allUsers = set(self.df['user_id'])
        self.sim = {}
        for person1 in allUsers:
            self.sim.setdefault(person1, {})
            a = self.df[
                self.df['user_id'] == person1][['movie_id']
                ]
            data_reduced = pd.merge(self.df, a,
                                    on = 'movie_id')
            for person2 in allUsers:
                # Avoid our-self
                if person1 == person2: continue
                self.sim.setdefault(person2, {})
                if(self.sim[person2].has_key(person1)):
                    continue # since symmetric matrix
                sim = self.sim_method(data_reduced,
                                      person1,
                                      person2)
                if(sim < 0):
                    self.sim[person1][person2] = 0
                    self.sim[person2][person1] = 0
                else:
                    self.sim[person1][person2] = sim
                    self.sim[person2][person1] = sim

    def predict(self, user_id, movie_id):
        totals = {}
        users = self.df[self.df['movie_id'] == movie_id]
```

In [11]:
```
rating_num, rating_den = 0.0, 0.0
        allUsers = set(users['user_id'])
        for other in allUsers:
            if user_id == other: continue
            rating_num +=
                self.sim[user_id][other] * float(users[users
                    ['user_id'] == other]['rating'])
           rating_den += self.sim[user_id][other]
        if rating_den == 0:
            if self.df.rating[self.df['movie_id'] ==
                movie_id].mean() > 0:
                # Mean movie rating if there is no similar
                    for the computation
               return self.df.rating[self.df['movie_id'] ==
                   movie_id].mean()
            else:
                # else mean user rating
                return self.df.rating[self.df['user_id'] ==
                    user_id].mean()
        return rating_num/rating_den
```

For the evaluation of the system we define a function called `evaluate`. This function estimates the score for all items in the test set (`X_test`) and compares them with the real values using the RMSE.

In [9]:
```
def evaluate(fit_f,train,test):
    """ RMSE-based predictive performance evaluation with
        pandas. """
    ids_to_estimate = zip(test.user_id, test.movie_id)
    estimated = np.array([fit_f(u, i)
                          if u
                          in train.user_id
                          else 3
                          for (u, i)
                          in ids_to_estimate])
    real = test.rating.values
    return compute_rmse(estimated, real)
```

Now, the system can be executed with the following lines:

In [10]:
```
print 'RMSE for Collaborative Recommender:',
print ' %s' % evaluate(reco.fit, data_train, data_test)
```

Out[10]:
```
RMSE for Collaborative Recommender: 1.00468945461
```

As we can see, the obtained $RMSE$ for this first basic recommender system is 1.004. Sure, that this result could be improved with a bigger dataset, but let us think of how we can improve it with just few tricks:

Trick 1: Since humans do not usually act the same as critics, i.e., some people usually rank movies higher or lower than others, this prediction function can be easily improved by taking into account the user mean as follows:

$$pred(a, p) = \bar{r_a} + \frac{\sum_{b \in B} sim(a, b) * (r_{b,p} - \bar{r_b})}{\sum_{b \in B} sim(a, b)} \tag{9.6}$$

where $\bar{r_a}$ and $\bar{r_b}$ are the mean rating of user a and b.

Table 9.2 Recommender system using mean user ratings

Critic	sim(a,b)	Mean ratings: $\bar{r_b}$	Rating movie1: r_{b,p_1}	$sim(a,b) * (r_{b,p_1})$
Paul	0.99	4.3	3	−1.28
Alice	0.38	2.73	3	0.1
Marc	0.89	3.12	4.5	1.22
Anne	0.92	3.98	3	−0.9
$\sum_{b \in N} sim(a,b) * (r_{b,p} - \bar{r_b})$				−1.13
$\sum_{b \in N} sim(a,b)$				3.18
$pred(a,p)$				3.14

Let us see an example: Prediction for the user "a" with $\bar{r_a} = 3.5$ (Table 9.2)

If we modify the recommender system using Eq. (9.6), the RMSE obtained is the following:

```
Out[11]: RMSE for Collaborative Recommender: 0.950086206741
```

Trick 2: One of the most critical steps with this kind of recommender system is the user similarity computation. If two users have very few items in common, let us imagine that there is only one, and the rating is the same, the user similarity will be really high; however, the confidence is really small. In order to solve this problem we can modify the similarity function as follows:

$$new\_sim(a,b) = sim(a,b) * \frac{min(K, |P_{ab}|)}{K} \tag{9.7}$$

where $|P_{ab}|$ is the number of common items shared by user a and user b, and K is the minimum number of common items in order not to penalize the similarity function.

In the next code, we define an update version of the similarity function called `simPersonCorrected` that follows the Eq. 9.7.

In [12]:
```
def SimPearsonCorrected(df, User1, User2,
                        min_common_items = 1,
                        pref_common_items = 20):
    """ RMSE-based predictive performance evaluation with
        pandas. """
    # GET MOVIES OF USER1
    m_user1 = df[df['user_id'] == User1 ]
    # GET MOVIES OF USER2
    m_user2 = df[df['user_id'] == User2 ]

    # FIND SHARED FILMS
    rep = pd.merge(m_user1, m_user2, on = 'movie_id')
    if len(rep) == 0:
        return 0
    if(len(rep) < min_common_items):
        return 0
```

In [12]:
```
        res = pearsonr(rep['rating_x'], rep['rating_y'])[0]
        res = res * min(pref_common_items, len(rep))
        res = res / pref_common_items
        if(isnan(res)):
            return 0
        return res
    reco4 = CollaborativeFiltering3(
        data_train,
        similarity = SimPearsonCorrected)
    reco4.learn()

    print 'RMSE for Collaborative Recommender:',
    print ' %s' % evaluate(reco4.fit, data_train, data_test)
```

Out[12]:
```
RMSE for Collaborative Recommender: 0.930811091922
```

As it can be seen, with this small modification the RMSE error has decreased from 1.0 to 0.93.

9.6 Conclusions

In this chapter, we have introduced what are recommender systems, how they work, and how they can be implemented in Python. We have seen that there are different types of recommender systems based on the information they use, as well as the output they produce. We have introduced content-based recommender systems and collaborative recommender systems; and we have seen the importance of defining the similarity function between items and users.

We have learned how recommender system can be implemented in Python in order to answer questions such as which movie should I see? We have also discussed how recommender system should be evaluated, and several online and offline metrics.

Finally, we have worked with a publicly available dataset from GroupLens in order to implement and evaluate a collaborative recommendation system for movie recommendations.

Acknowledgements This chapter was co-written by Santi Seguí and Eloi Puertas

References

1. G. Shani, A. Gunawardana, A survey of accuracy evaluation metrics of recommendation tasks. in *J. Mach. Learn. Res.*, 10:2935–2962, 2009
2. F. Ricci, L. Rokach, B. Schapira, in *Recommender Systems Handbook* (Springer, 2015).

Statistical Natural Language Processing for Sentiment Analysis

10

10.1 Introduction

In this chapter, we will perform sentiment analysis from text data. The term sentiment analysis (or opinion mining) refers to the analysis from data of the attitude of the subject with respect to a particular topic. This attitude can be a judgment (appraisal theory), an affective state, or the intended emotional communication.

Generally, sentiment analysis is performed based on the processing of natural language, the analysis of text and computational linguistics. Although data can come from different data sources, in this chapter we will analyze sentiment in text data, using two particular text data examples: one from film critics, where the text is highly structured and maintains text semantics; and another example coming from social networks (tweets in this case), where the text can show a lack of structure and users may use (and abuse!) text abbreviations.

In the following sections, we will review some basic mechanisms required to perform sentiment analysis. In particular, we will analyze the steps required for data cleaning (that is, removing irrelevant text items not associated with sentiment information), producing a general representation of the text, and performing some statistical inference on the text represented to determine positive and negative sentiments.

Although the scope of sentiment analysis may introduce many aspects to be analyzed, in this chapter and for simplicity, we will analyze binary sentiment analysis categorization problems. We will thus basically learn to classify positive against negative opinions from text data. The scope of sentiment analysis is broader, and it includes many aspects that make analysis of sentiments a challenging task. Some interesting open issues in this topic are as follows:

- Identification of sarcasm: sometimes without knowing the personality of the person, you do not know whether "bad" means bad or good.

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_10

- Lack of text structure: in the case of Twitter, for example, it may contain abbreviations, and there may be a lack of capitals, poor spelling, poor punctuation, and poor grammar, all of which make it difficult to analyze the text.
- Many possible sentiment categories and degrees: positive and negative is a simple analysis, one would like to identify the amount of hate there is inside the opinion, how much happiness, how much sadness, etc.
- Identification of the object of analysis: many concepts can appear in text, and how to detect the object that the opinion is positive for and the object that the opinion is negative for is an open issue. For example, if you say "She won him!", this means a positive sentiment for her and a negative sentiment for him, at the same time.
- Subjective text: another open challenge is how to analyze very subjective sentences or paragraphs. Sometimes, even for humans it is very hard to agree on the sentiment of these highly subjective texts.

10.2 Data Cleaning

In order to perform sentiment analysis, first we need to deal with some processing steps on the data. Next, we will apply the different steps on simple "toy" sentences to understand better each one. Later, we will perform the whole process on larger datasets.

Given the input text data in cell `[1]`, the main task of data cleaning is to remove those characters considered as noise in the data mining process. For instance, comma or colon characters. Of course, in each particular data mining problem different characters can be considered as noise, depending on the final objective of the analysis. In our case, we are going to consider that all punctuation characters should be removed, including other non-conventional symbols. In order to perform the data cleaning process and posterior text representation and analysis we will use the *Natural Language Toolkit* (NLTK) library for the examples in this chapter.

In [1]:
```
raw_docs = ["Here are some very simple basic
    sentences.",
"They won't be very interesting, I'm afraid.",
"The point of these examples is to _learn  how
    basic text \
cleaning works_ on *very simple* data."]
```

The first step consists of defining a list with all word-vectors in the text. NLTK makes it easy to convert documents-as-strings into word-vectors, a process called tokenizing. See the example below.

In [2]:
```
from nltk.tokenize import word_tokenize
tokenized_docs = [word_tokenize(doc) for doc in
    raw_docs]
print tokenized_docs
```

Out[2]:
```
[['Here', 'are', 'some', 'very', 'simple', 'basic',
'sentences', '.'], ['They', 'wo', "n't", 'be', 'very',
'interesting', ',', 'I', "'m", 'afraid', %'.'], ['The',
'point', 'of', 'these', 'examples', 'is', 'to', '_learn',
'how', %'basic', 'text', 'cleaning', 'works_', 'on', '*very',
'simple*', 'data', '.']]
```

Thus, for each line of text in `raw_docs`, `word_tokenize` function will set the list of word-vectors. Now we can search the list for punctuation symbols, for instance, and remove them. There are many ways to perform this step. Let us see one possible solution using the `String` library.

In [3]:
```
import string
string.punctuation
```

Out[3]:
```
'!"#\$\%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

See that `string.punctuation` contains a set of common punctuation symbols. This list can be modified according to the symbols you want to remove. Let us see with the next example using the *Regular Expressions* (RE) package how punctuation symbols can be removed. Note that many other possibilities to remove symbols exist, such as directly implementing a loop comparing position by position.

In the input cell `[6]`, and without going into the details of RE, `re.compile` contains a list of "expressions", the symbols contained in `string.punctuation`.

Then, for each item in `tokenized_docs` that matches an expression/symbol contained in `regex`, the part of the item corresponding to the punctuation will be substituted by `u"` (where `u` refers to unicode encoding). If the item after substitution corresponds to `u"`, it will be not included in the final list. If the new item is different from `u"`, it means that the item contained text other than punctuation, and thus it is included in the new list without punctuation `tokenized_docs_no_punctuation`. The results of applying this script are shown in the output cell `[7]`.

In [4]:
```
import re
import string
regex = re.compile('[%s]' % re.escape(string.
    punctuation))
tokenized_docs_no_punctuation = []
for review in tokenized_docs:
    new_review = []
    for token in review:
        new_token = regex.sub(u'', token)
        if not new_token == u'':
            new_review.append(new_token)
    tokenized_docs_no_punctuation.append(new_review
        )
print tokenized_docs_no_punctuation
```

Out[4]:
```
[['Here', 'are', 'some', 'very', 'simple', 'basic',
'sentences'],
['They', 'wo', u'nt', 'be', 'very', 'interesting', 'I', u'm',
'afraid'],
['The', 'point', 'of', 'these', 'examples', 'is', 'to',
u'learn', 'how', 'basic', 'text', 'cleaning', u'works', 'on',
u'very', u'simple', 'data']]
```

One can see that punctuation symbols are removed, and those words containing a punctuation symbol are kept and marked with an initial `u`. If the reader wants more details, we recommend to read information about the RE package[1] for treating expressions.

Another important step in many data mining systems for text analysis consists of stemming and lemmatizing. Morphology is the notion that words have a root form. If you want to get to the basic term meaning of the word, you can try applying a stemmer or lemmatizer. This step is useful to reduce the dictionary size and the posterior high-dimensional and sparse feature spaces. NLTK provides different ways of performing this procedure. In the case of running the `porter.stem(word)` approach, the output is shown next.

In [5]:
```
from nltk.stem.porter import PorterStemmer
from nltk.stem.snowball import SnowballStemmer
from nltk.stem.wordnet import WordNetLemmatizer
porter = PorterStemmer()
#snowball = SnowballStemmer('english')
#wordnet = WordNetLemmatizer()

#each of the following commands perform stemming on
    word

porter.stem(word)
#snowball.stem(word)
#wordnet.lemmatize(word)
```

Out[5]:
```
[['Here', 'are', 'some', 'very', 'simple', 'basic',
'sentences'], ['They', 'wo', u'nt', 'be', 'very',
'interesting', 'I', u'm', 'afraid'], ['The', 'point', 'of',
'these', 'examples', 'is', 'to', u'learn', 'how', 'basic',
'text', 'cleaning', u'works', 'on', u'very', u'simple',
'data']]
[['Here', 'are', 'some', 'veri', 'simpl', 'basic', 'sentenc'],
['They', 'wo', u'nt', 'be', 'veri', 'interest', 'I', u'm',
'afraid'], ['The', 'point','of', 'these', 'exampl', 'is',
'to', u'learn', 'how', 'basic', 'text', 'clean', u'work', 'on',
u'veri',u'simpl', 'data']]
```

[1] https://docs.python.org/2/library/re.html.

This kind of approaches are very useful in order to reduce the exponential number of combinations of words with the same meaning and match similar texts. Words such as “interest” and “interesting” will be converted into the same word “interest” making the comparison of texts easier, as we will see later.

Another very useful data cleaning procedure consists of removing HTML entities and tags. Those may contain words and other symbols that were not removed by applying the previous procedures, but that do not provide useful meaning for text analysis and will introduce noise in our posterior text representation procedure. There are many possibilities for removing these tags. Here we show another example using the same NLTK package.

In [6]:
```
import nltk
test_string ="<p>While many of the stories tugged
    at the heartstrings, I never felt manipulated by
     the authors. (Note: Part of the reason why I
    don't like the 'Chicken Soup for the Soul'
    series is that I feel that the authors are just
    dying to make the reader clutch for the box of
    tissues.)</a>"
print 'Original text:'
print test_string
print 'Cleaned text:'
nltk.clean_html(test_string.decode())
```

Out[6]:
```
Original text:
<p>While many of the stories tugged at the heartstrings, I
never felt manipulated by the authors. (Note: Part of the
reason why I don't like the "Chicken Soup for the Soul" series
is that I feel that the authors are just dying to make the
reader clutch for the box of tissues.)</a>

Cleaned text:
u"While many of the stories tugged at the heartstrings, I never
felt manipulated by the authors. (Note: Part of the reason why
I don't like the "Chicken Soup for the Soul" series is that I
feel that the authors are just dying to make the reader clutch
for the box of tissues.)"
```

You can see that tags such as “<p>” and “” have been removed. The reader is referred to the RE package documentation to learn more about how to use it for data cleaning and HTLM parsing to remove tags.

10.3 Text Representation

In the previous section we have analyzed different techniques for data cleaning, stemming, and lemmatizing, and filtering the text to remove other unnecessary tags for posterior text analysis. In order to analyze sentiment from text, the next step consists of having a representation of the text that has been cleaned. Although different rep-

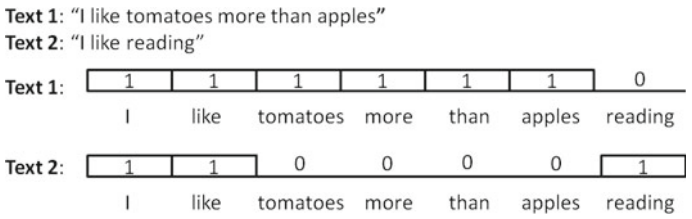


Fig. 10.1 Example of BoW representation for two texts

resentations of text exist, the most common ones are variants of *Bag of Words* (BoW) models [1]. The basic idea is to think about word frequencies. If we can define a dictionary of possible different words, the number of different existing words will define the length of a feature space to represent each text. See the toy example in Fig. 10.1. Two different texts represent all the available texts we have in this case. The total number of different words in this dictionary is seven, which will represent the length of the feature vector. Then we can represent each of the two available texts in the form of this feature vector by indicating the number of word frequencies, as shown in the bottom of the figure. The last two rows will represent the feature vector codifying each text in our dictionary.

Next, we will see a particular case of bag of words, the *Vector Space Model* of text: TF–IDF (term frequency–inverse distance frequency). First, we need to count the terms per document, which is the term frequency vector. See a code example below.

In [7]:
```
mydoclist = ['Mireia loves me more than Hector
    loves me',
     'Sergio likes me more than Mireia loves me',
     'He likes basketball more than football']
from collections import Counter
for doc in mydoclist:
     tf = Counter()
     for word in doc.split():
          tf[word] += 1
     print tf.items()
```

Out[7]:
```
[('me', 2), ('Mireia', 1), ('loves', 2), ('Hector', 1),
('than', 1), ('more', 1)] [('me', 2), ('Mireia', 1), ('likes',
1), ('loves', 1), ('Sergio', 1), ('than', 1), ('more', 1)]
[('basketball', 1), ('football', 1), ('likes', 1), ('He', 1),
('than', 1), ('more', 1)]
```

Here, we have introduced the Python object called a `Counter`. Counters are only in Python 2.7 and higher. They are useful because they allow you to perform this exact kind of function: counting in a loop. A Counter is a dictionary subclass for counting hashable objects. It is an unordered collection where elements are stored as dictionary keys and their counts are stored as dictionary values. Counts are allowed to be any integer value including zero or negative counts.

Elements are counted from an iterable or initialized from another mapping (or Counter).

In [8]:
```
c = Counter()            # a new, empty counter
c = Counter('gallahad') # a new counter from an
    iterable
```

Counter objects have a dictionary interface except that they return a zero count for missing items instead of raising a KeyError.

In [9]:
```
c = Counter(['eggs', 'ham'])
c['bacon']
```

Out[9]: 0

Let us call this a first stab at representing documents quantitatively, just by their word counts (also thinking that we may have previously filtered and cleaned the text using previous approaches). Here we show an example for computing the feature vector based on word frequencies.

In [10]:
```
def build_lexicon(corpus):
# define a set with all possible words included in
    all the sentences or "corpus"
    lexicon = set()
    for doc in corpus:
        lexicon.update([word for word in doc.split
            ()])
    return lexicon
def tf(term, document):
    return freq(term, document)
def freq(term, document):
    return document.split().count(term)
vocabulary = build_lexicon(mydoclist)
doc_term_matrix = []
print 'Our vocabulary vector is [' +
      ', '.join(list(vocabulary)) + ']'
for doc in mydoclist:
    print 'The doc is "' + doc + '"'
    tf_vector = [tf(word, doc) for word in
        vocabulary]
    tf_vector_string = ', '.join(format(freq, 'd')
                                 for freq
                                 in tf_vector)
    print 'The tf vector for Document %d is [%s]'
          % ((mydoclist.index(doc)+1),
              tf_vector_string)
    doc_term_matrix.append(tf_vector)
print 'All combined, here is our master document
      term matrix: '
print doc_term_matrix
```

Out[10]:
```
Our vocabulary vector is [me, basketball, Julie, baseball,
likes, loves, Jane, Linda, He, than, more]
The doc is "Julie loves me more than Linda loves me"
The tf vector for Document 1 is [2, 0, 1, 0, 0, 2, 0, 1, 0, 1,
1]
The doc is "Jane likes me more than Julie loves me"
The tf vector for Document 2 is [2, 0, 1, 0, 1, 1, 1, 0, 0, 1,
1]
The doc is "He likes basketball more than baseball"
The tf vector for Document 3 is [0, 1, 0, 1, 1, 0, 0, 0, 1, 1,
1]
All combined, here is our master document term matrix:
[[2, 0, 1, 0, 0, 2, 0, 1, 0, 1, 1], [2, 0, 1, 0, 1, 1, 1, 0, 0,
1, 1], [0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1]]
```

Now, every document is in the same feature space, meaning that we can represent the entire corpus in the same dimensional space. Once we have the data in the same feature space, we can start applying some machine learning methods: learning, classifying, clustering, and so on. But actually, we have a few problems. Words are not all equally informative. If words appear too frequently in a single document, they are going to muck up our analysis. We want to perform some weighting of these term frequency vectors into something a bit more representative. That is, we need to do some vector normalizing. One possibility is to ensure that the L2 norm of each vector is equal to 1.

In [11]:
```
import math

def l2_normalizer(vec):
    denom = np.sum([el**2 for el in vec])
    return [(el / math.sqrt(denom)) for el in vec]
doc_term_matrix_l2 = []
for vec in doc_term_matrix:
    doc_term_matrix_l2.append(l2_normalizer(vec))
print 'A regular old document term matrix: '
print np.matrix(doc_term_matrix)
print '\nA document term matrix with row-wise L2
   norm:'
print np.matrix(doc_term_matrix_l2)
```

Out[11]:
```
A regular old document term matrix:
[[2 0 1 0 0 2 0 1 0 1 1]
[2 0 1 0 1 1 1 0 0 1 1]
[0 1 0 1 1 0 0 0 1 1 1]]
A document term matrix with row-wise L2 norm:
[[ 0.57735027 0. 0.28867513 0. 0. 0.57735027
0. 0.28867513 0. 0.28867513 0.28867513]
[ 0.63245553 0. 0.31622777 0. 0.31622777 0.31622777
0.31622777 0. 0. 0.31622777 0.31622777]
[ 0. 0.40824829 0. 0.40824829 0.40824829 0. 0.
0. 0.40824829 0.40824829 0.40824829]]
```

You can see that we have scaled down the vectors so that each element is between [0, 1]. This will avoid getting a diminishing return on the informative value of a word massively used in a particular document. For that, we need to scale down words that appear too frequently in a document.

Finally, we have a final task to perform. Just as not all words are equally valuable within a document, not all words are valuable across all documents. We can try reweighting every word by its inverse document frequency.

In [12]:
```
def numDocsContaining(word, doclist):
    doccount = 0
    for doc in doclist:
        if freq(word, doc) > 0:
            doccount += 1
    return doccount
def idf(word, doclist):
    n_samples = len(doclist)
    df = numDocsContaining(word, doclist)
    return np.log(n_samples / (float(df)) )
my_idf_vector = [idf(word, mydoclist) for word in
    vocabulary]
print 'Our vocabulary vector is [' + ', '.join(list
    (vocabulary)) + ']'
print 'The inverse document frequency vector is
    [' + ', '.join(format(freq, 'f') for freq in
    my_idf_vector) + ']'
```

Out[12]:
```
Our vocabulary vector is [me, basketball, Mireia, football,
likes, loves, Sergio, Hector, He, than, more]
The inverse document frequency vector is [0.405465, 1.098612,
0.405465, 1.098612, 0.405465, 0.405465, 1.098612, 1.098612,
1.098612, 0.000000, 0.000000]
```

Now we have a general sense of information values per term in our vocabulary, accounting for their relative frequency across the entire corpus. Note that this is an inverse. To get TF–IDF weighted word-vectors, we have to perform the simple calculation of the term frequencies multiplied by the inverse frequency values.

In the next example we convert our IDF vector into a matrix where the diagonal is the IDF vector.

In [13]:
```
def build_idf_matrix(idf_vector):
    idf_mat = np.zeros((len(idf_vector), len(
        idf_vector)))
    np.fill_diagonal(idf_mat, idf_vector)
    return idf_mat
my_idf_matrix = build_idf_matrix(my_idf_vector)
print my_idf_matrix
```

Out[13]:
```
[[ 0.40546511 0. 0. 0. 0. 0. 0. 0. 0. 0. ]
[ 0. 1.09861229 0. 0. 0. 0. 0. 0. 0. 0. ]
[ 0. 0. 0.40546511 0. 0. 0. 0. 0. 0. 0. ]
[ 0. 0. 0. 1.09861229 0. 0. 0. 0. 0. 0. ]
[ 0. 0. 0. 0. 0.40546511 0. 0. 0. 0. 0. ]
[ 0. 0. 0. 0. 0. 0.40546511 0. 0. 0. 0. ]
[ 0. 0. 0. 0. 0. 0. 1.09861229 0. 0. 0. ]
[ 0. 0. 0. 0. 0. 0. 0. 1.09861229 0. 0. ]
[ 0. 0. 0. 0. 0. 0. 0. 0. 1.09861229 0. ]
[ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ]
[ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. ]]
```

That means we can now multiply every term frequency vector by the inverse document frequency matrix. Then, to make sure we are also accounting for words that appear too frequently within documents, we will normalize each document using the L2 norm.

In [14]:
```
doc_term_matrix_tfidf = []
#performing tf-idf matrix multiplication
for tf_vector in doc_term_matrix:
    doc_term_matrix_tfidf.append(np.dot(tf_vector,
        my_idf_matrix))
#normalizing
doc_term_matrix_tfidf_l2 = []
for tf_vector in doc_term_matrix_tfidf:
    doc_term_matrix_tfidf_l2.
    append(l2_normalizer(tf_vector))
print vocabulary
# np.matrix() just to make it easier to look at
print np.matrix(doc_term_matrix_tfidf_l2)
```

Out[14]:
```
set(['me', 'basketball', 'Mireia', 'football', 'likes',
'loves', 'Sergio', 'Linda', 'He', 'than', 'more'])
[[ 0.49474872 0. 0.24737436 0. 0. 0.49474872 0. 0.67026363 0.
0. 0. ]
[ 0.52812101 0. 0.2640605 0. 0.2640605 0.2640605 0.71547492 0.
0. 0. 0. ]
[ 0. 0.56467328 0. 0.56467328 0.20840411 0. 0. 0. 0.56467328 0.
0. ]]
```

10.3.1 Bi-Grams and n-Grams

It is sometimes useful to take significant bi-grams into the model based on the BoW. Note that this example can be extended to n-grams. In the fields of computational linguistics and probability, an n-gram is a contiguous sequence of n items from a given sequence of text or speech. The items can be phonemes, syllables, letters, words, or base pairs according to the application. The n-grams are typically collected from a text or speech corpus.

A n-gram of size 1 is referred to as a "uni-gram"; size 2 is a "bi-gram" (or, less commonly, a "digram"); size 3 is a "tri-gram". Larger sizes are sometimes referred to by the value of n, e.g., "four-gram", "five-gram", and so on. These n-grams can be introduced within the BoW model just by considering each different n-gram as a new position within the feature vector representation.

10.4 Practical Cases

Python packages provide useful tools for analyzing text. The reader is referred to the NLTK and *Textblob* package[2] documentation for further details. Here, we will perform all the previously presented procedures for data cleaning, stemming, and representation and introduce some binary learning schemes to learn the text representations in the feature space. The binary learning schemes will receive examples for training positive and negative sentiment texts and we will apply them later to unseen examples from a test set.

We will apply the whole sentiment analysis process in two examples. The first corresponds to the Large Movie reviews dataset [2]. This is one of the largest public available data sets for sentiment analysis, which includes more than 50,000 texts from movie reviews including the groundtruth annotation related to positive and negative movie reviews. As a proof on concept, for this example we use a subset of the dataset consisting of about 30% of the data.

The code reuses part of the previous examples for data cleaning, reads training and testing data from the folders as provided by the authors of the dataset. Then, TF–IDF is computed, which performs all steps mentioned previously for computing feature space, normalization, and feature weights. Note that at the end of the script we perform training and testing based on two different state-of-the-art machine learning approaches: *Naive Bayes* and *Support Vector Machines*. It is beyond the scope of this chapter to give details of the methods and parameters. The important point here is that the documents are represented in feature spaces that can be used by different data mining tools.

[2] https://textblob.readthedocs.io/en/dev/.

In [15]:

```
from nltk.tokenize import word_tokenize
from nltk.stem.porter import PorterStemmer
from sklearn.feature_extraction.text import
    TfidfVectorizer
from nltk.classify import NaiveBayesClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn import svm
from unidecode import unidecode

def BoW(text):
    # Tokenizing text
    text_tokenized = [word_tokenize(doc) for doc in
        text]
    # Removing punctuation
    regex = re.compile('[%s]' % re.escape(string.
        punctuation))
    tokenized_docs_no_punctuation = []
    for review in text_tokenized:
        new_review = []
        for token in review:
            new_token = regex.sub(u'', token)
            if not new_token == u'':
                new_review.append(new_token)
        tokenized_docs_no_punctuation.append(
            new_review)
    # Stemming and Lemmatizing
    porter = PorterStemmer()
    preprocessed_docs = []
    for doc in tokenized_docs_no_punctuation:
        final_doc = ''
        for word in doc:
            final_doc = final_doc + ' ' + porter.
                stem(word)
        preprocessed_docs.append(final_doc)
    return preprocessed_docs

#read your train text data here
textTrain=ReadTrainDataText()
preprocessed_docs=BoW(textTrain) # for train data
# Computing TIDF word space
tfidf_vectorizer = TfidfVectorizer(min_df = 1)
trainData = tfidf_vectorizer.fit_transform(
    preprocessed_docs)

textTest=ReadTestDataText() #read your test text
    data here
prepro_docs_test=BoW(textTest) # for test data
testData = tfidf_vectorizer.transform(
    prepro_docs_test)
```

In [16]:

```
print('Training and testing on training Naive Bayes
    ')
gnb = GaussianNB()
testData.todense()
y_pred = gnb.fit(trainData.todense(), targetTrain)
            .predict(trainData.todense())
print("Number of mislabeled training points out of
    a total %d points : %d"
       % (trainData.shape[0],(targetTrain != y_pred)
           .sum()))

y_pred = gnb.fit(trainData.todense(), targetTrain)
            .predict(testData.todense())
print("Number of mislabeled test points out of a
    total %d points : %d" %
       (testData.shape[0],(targetTest != y_pred).sum
           ()))

print('Training and testing on train with SVM')
clf = svm.SVC()
clf.fit(trainData.todense(), targetTrain)
y_pred = clf.predict(trainData.todense())
print("Number of mislabeled test points out of a
    total %d points : %d" %
       (trainData.shape[0],(targetTrain != y_pred).
           sum()))

print('Testing on test with already trained SVM')
y_pred = clf.predict(testData.todense())
print("Number of mislabeled test points out of a
    total %d points : %d" %
       (testData.shape[0],(targetTest != y_pred).sum
           ()))
```

In addition to the machine learning implementations provided by the Scikit-learn module used in this example, NLTK also provides useful learning tools for text learning, which also includes Naive Bayes classifiers. Another related package with similar functionalities is Textblob. The results of running the script are shown next.

Out[16]:
```
Training and testing on training Naive Bayes
Number of mislabeled training points out of a total 4313 points
: 129
Number of mislabeled test points out of a total 6292 points :
2087
Training and testing on train with SVM
Number of mislabeled test points out of a total 4313 points :
1288
Testing on test with already trained SVM
Number of mislabeled test points out of a total 6292 points :
1680
```

We can see that the training error of Naive Bayes on the selected data is 129/4313 while in testing it is 2087/6292. Interestingly, the training error using SVM is higher (1288/4313), but it provides a better generalization of the test set than Naive Bayes (1680/6292). Thus it seems that Naive Bayes produces more overfitting of the data (selecting particular features for better learning the training data but producing such high modifications of the feature space for testing that cannot be recovered, just reducing the generalization capability of the technique). However, note that this is a simple execution with standard methods on a subset of the dataset provided. More data, as well as many other aspects, will influence the performance. For instance, we could enrich our dictionary by introducing a list of already studied positive and negative words.[3] For further details of the analysis of this dataset, the reader is referred to [2].

Finally, let us see another example of sentiment analysis based on tweets. Although there is some work using more tweet data[4] here we present a reduced set of tweets which are analyzed as in the previous example of movie reviews. The main code remains the same except for the definition of the initial data.

In [17]:
```
textTrain = ['I love this sandwich.', 'This is an
    amazing place!', 'I feel very good about these
    beers.', 'This is my best work.', 'What an
    awesome view', 'I do not like this restaurant',
    'I am tired of this stuff.', 'I can not deal
    with this', 'He is my sworn enemy!', 'My boss is
     horrible.']
targetTrain = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
preprocessed_docs=BoW(textTrain)
tfidf_vectorizer = TfidfVectorizer(min_df = 1)
trainData = tfidf_vectorizer.fit_transform(
    preprocessed_docs)
```

[3] Such as those provided in http://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html.
[4] http://www.sananalytics.com/lab/twitter-sentiment/.

In [18]:
```
textTest = ['The beer was good.', 'I do not enjoy
    my job', 'I aint feeling dandy today', 'I feel
    amazing!', 'Gary is a friend of mine.', 'I can
    not believe I am doing this.']
targetTest = [0, 1, 1, 0, 0, 1]
preprocessed_docs=BoW(textTest)
testData = tfidf_vectorizer.transform(
    preprocessed_docs)

print('Training and testing on test Naive Bayes')
gnb = GaussianNB()
testData.todense()
y_pred = gnb.fit(trainData.todense(), targetTrain)
           .predict(trainData.todense())
print("Number of mislabeled training points out of
    a total %d points : %d" % (trainData.shape[0],(
    targetTrain != y_pred).sum()))

y_pred = gnb.fit(trainData.todense(), targetTrain)
           .predict(testData.todense())
print("Number of mislabeled test points out of a
    total %d points : %d" % (testData.shape[0],(
    targetTest != y_pred).sum()))

print('Training and testing on train with SVM')
clf = svm.SVC()
clf.fit(trainData.todense(), targetTrain)
y_pred = clf.predict(trainData.todense())
print("Number of mislabeled test points out of a
    total
        %d points : %d"
        % (trainData.shape[0],(targetTrain != y_pred
            ).sum()))

print('Testing on test with already trained SVM')
y_pred = clf.predict(testData.todense())
print("Number of mislabeled test points out of a
    total
       %d points : %d"
       % (testData.shape[0],(targetTest != y_pred).
           sum()))
```

Out[17]:
```
Training and testing on test Naive Bayes
Number of mislabeled training points out of a total 10 points : 0
Number of mislabeled test points out of a total 6 points : 2
Training and testing on train with SVM
Number of mislabeled test points out of a total 10 points : 0
Testing on test with already trained SVM
Number of mislabeled test points out of a total 6 points : 2
```

In this scenario both learning strategies achieve the same recognition rates in both training and test sets. Note that similar words are shared between tweets. In practice,

with real examples, tweets will include unstructured sentences and abbreviations, making recognition harder.

10.5 Conclusions

In this chapter, we have analyzed the problem of binary sentiment analysis of text data: data cleaning to remove irrelevant symbols, punctuation and tags; stemming in order to define the same root for different works with the same meaning in terms of sentiment; defining a dictionary of words (including n-grams); and representing text in terms of a feature space with the length of the dictionary. We have also seen codification in this feature space, based on normalized and weighted term frequencies. We have defined feature vectors that can be used by any machine learning technique in order to perform sentiment analysis (binary classification in the examples shown), and reviewed some useful Python packages, such as NLTK and Textblob, for sentiment analysis.

As discussed in the introduction of this chapter, we have only reviewed the sentiment analysis problem and described common procedures for performing the analysis resulting from a binary classification problem. Several open issues can be addressed in further research, such as the identification of sarcasm, a lack of text structure (as in tweets), many possible sentiment categories and degrees (not only binary but also multiclass, regression, and multilabel problems, among others), identification of the object of analysis, or subjective text, to name a few.

The tools described in this chapter can define a basis for dealing with those more challenging problems. One recent example of current state-of-the-art research is the work of [3], where deep learning architectures are used for sentiment analysis. Deep learning strategies are currently a powerful tool in the fields of pattern recognition, machine learning, and computer vision, among others; the main deep learning strategies are based on neural network architectures. In the work of [3], a deep learning model builds up a representation of whole sentences based on the sentence structure, and it computes the sentiment based on how words form the meaning of longer phrases. In the methods explained in this chapter, n-grams are the only features that capture those semantics. For further discussion in this field, the reader is referred to [4,5].

Acknowledgements This chapter was co-written by Sergio Escalera and Santi Seguí.

References

1. Z. Ren, J. Yuan, J. Meng, Z. Zhang, IEEE Transactions on Multimedia **15**(5), 1110 (2013)

2. A.L. Maas, R.E. Daly, P.T. Pham, D. Huang, A.Y. Ng, C. Potts, in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies* (Association for Computational Linguistics, Portland, Oregon, USA, 2011), pp. 142–150. URL http://www.aclweb.org/anthology/P11-1015
3. R. Socher, A. Perelygin, J. Wu, J. Chuang, C. Manning, A. Ng, C. Potts, Conference on Empirical Methods in Natural Language Processing (2013)
4. E. Cambria, B. Schuller, Y. Xia, C. Havasi, IEEE Intelligent Systems **28**(2), 15 (2013)
5. B. Pang, L. Lee, Found. Trends Inf. Retr. **2**(1–2), 1 (2008)

Parallel Computing

11

11.1 Introduction

The computer industry underwent a vigorous shake-up several years ago. Major chip manufacturers gave up trying to increase processor frequency. Each year, more and more transistors fit into the same space, but their clock speed cannot be increased without overheating. Thus, rather than trying to increase the clock speed, manufacturers turned to multicore architectures. A multicore processor is a single computing component with two or more processing units (called "cores") which read and execute program instructions. Multiple cores can run different instructions at the same time, thereby increasing the overall speed of programs susceptible to parallel computing. Within multicore systems, the cores communicate through hardware (the bus) in order to synchronize access to common resources such as RAM.

The operating system is the application that manages these multiple cores. If two computation-intensive processes (i.e., applications) are run on the computer, the operating system manages things so that each task is run on a different core. If we have a single computation-intensive task, it will only run on one core, even if our computer has multiple cores. If nothing is done explicitly, we will waste a lot of computation power!

Currently, in most parallel programming frameworks, the programmer has to manually split the computation work into multiple tasks so that each one is executed in different cores. The programmer has to perform the split and the operating system will then automatically execute each task on a different core. So, each task has to be run in different processes or threads. This is the principle behind parallel programming; harnessing multiple processors to work on a single task by dividing it into multiple (smaller) tasks.

In order to make the most of multicore capabilities, the number of processes should be equal to the number of processors. Within a parallel computing context, it does not make much sense to define more tasks than cores we have, e.g., defining eight computation-intensive tasks if our computer only has four cores. In this latter

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1_11

case, the operating system will try to run eight tasks using four cores. This is done by switching between the tasks in such a way that each one gets approximately the same amount of computing time. Switching between tasks has a computational cost and thus overall performance may suffer if the number of simultaneous tasks is higher than the number of available cores.

Assume that a task takes T seconds to run on a single core (using standard serialized programming). Now assume that we have a computer with N cores and that we have divided our serialized application into N subtasks. By using the parallel capabilities of our computer we may be able to reduce the total computation time to T/N. This is the ideal case and usually we will not be able to reduce the computation time by a factor of N. This is due to the fact that cores, on the one hand, need to synchronize at the hardware level in order to access common resources such as RAM; and, on the other hand, the operating system needs some time to switch between all the tasks that run on the computer. However, using the multicore capabilities of the computer unit will result in a reduction of the computation time if the tasks are properly defined.

Parallelization can also be performed by means of distributed computing. While in multicore systems the cores communicate with each other through the bus at the hardware level, in distributed systems software communicates and coordinates the actions of computational entities located within a network. The computational entities are usually computers. In distributed computing, a large number of discrete computers, named *nodes*, distributed across a network (e.g., the Internet) devote some or all of their computation time to solving a common problem; each node receives and completes many small tasks, reporting the results to a central server which integrates the results into the overall solution. Each of the nodes has its own local memory and thus tasks that run on different computers do not need to coordinate access to it. However, since information is exchanged through the network, care must be taken in order to select the amount of information that is passed so as to optimize the computational performance.

In this chapter we will focus on IPython's capabilities for parallel computing, on both multicore and distributed systems. IPython does indeed offer an environment capable of dealing with both architectures in a transparent manner for the programmer. The user should be aware of the underlying architecture in which the application will be run in order to avoid loss of performance. We would like to point out that Python currently does not offer support for the parallel capabilities explained below. IPython, however, supports them.

11.2 Architecture

Figure 11.1 shows a simplified version of the IPython architecture for parallel computing (multicore and distributed).[1] The proposed architecture enables IPython to

[1]For a more detailed description please see http://ipyparallel.readthedocs.io/en/stable/intro.html. Last seen July 2016.

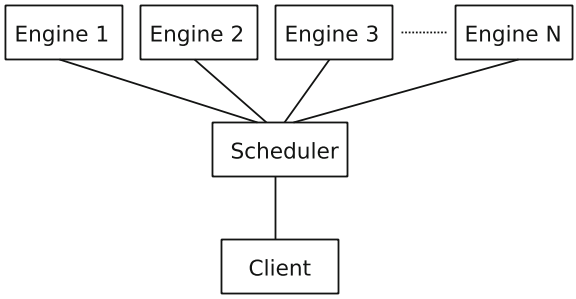


Fig. 11.1 IPython's architecture for parallel computing (multicore and distributed)

support many different styles of parallelism including those described in this chapter. Each of the blocks is explained below:

- Each engine is an instance of IPython, usually an IPython interpreter, that receives commands through a connection. When multiple engines are started, multicore and distributed computing becomes possible.
- The scheduler is an application that distributes the commands to the engines. We will see that there are two ways of distributing this work: the direct view and the load-balanced view, described in later sections.
- The client is an IPython object created at an IPython interpreter. This object will allow us to send commands to the IPython engines.

IPython uses the term *cluster* to refer to the scheduler and the set of engines that make parallelization possible. It should not be confused with the term cluster used in supercomputing. In addition, the reader should take into account that:

- Each engine is an independent instance of an IPython interpreter, i.e., it runs an independent process. None of the variables declared at, e.g., engine 1 are visible to the remaining engines or to the client. In a similar way, if we want to work with `numpy` functions, we should import this toolbox to every engine.
- We may be able to control at which engine each task is executed, but we will not be able to control on which core each engine is executed; this is the job of the operating system.

11.2.1 Getting Started

To use IPython's parallel capabilities, the first thing to do is to start the cluster. There are two ways of doing this:

- From the notebook interface. This is the simplest way of proceeding and is the recommended way for newbies in this topic. Within the IPython notebook, we can use the Clusters tab of the dashboard, and press Start with the desired number

of cores, under the desired profile.[2] This will automatically run the necessary commands to start the IPython cluster. In this case, the notebook will be used as the interface with the cluster; i.e., we will be able to send different tasks to the engines using the web interface.

- From the command line of a terminal. We can run the following command to start an IPython cluster:

```
$ ipcluster start
```

This command will create a cluster with N engines, where N equals the number of cores. If we want to create a cluster with a different number of engines, we just run:

```
$ ipcluster start -n 4
```

With this command we start a cluster with four engines. Once the engines are started, we may run an IPython interpreter.

```
$ ipython
```

11.2.2 Connecting to the Cluster (The Engines)

We have seen how to initialize the cluster. No matter which way we initialize the cluster, the following commands allow us to connect to it. These commands should either be introduced through the notebook or be typed into the IPython command line interpreter (the client):

In [1]:
```
from IPython import parallel
engines = parallel.Client()
engines.block = True
print engines.ids
```

Out[1]:
```
[0, 1, 2, 3, 4, 5, 6, 7]
```

These commands connect to the cluster and output the number of engines in it. If an error is shown when running the commands, the cluster has not been correctly created. We will explain later on the meaning of the `block` attribute.

The variable `engines` is an object that represents the available engines to which commands can be sent. Let us now see two different ways we can send tasks to the engines: the first, called the *direct view*, is simpler and allows the user to directly control which tasks are sent to which engines; the second, called the *load-balanced view*, delegates to the IPython scheduler the task of deciding which engines each task is sent to.

[2] More information on ipcluster profiles can be found at http://ipython.readthedocs.io/en/stable/.

As will be seen next, the former view is useful if a task can be evenly distributed computationally into smaller tasks; whereas the second is more useful if such subdivision cannot be easily done. For instance, if we have to analyze multiple data files, the direct view is a good approach if all the files have approximately the same size. But if the files differ (quite a lot) in size, the load-balanced view is the better approach. Let us now see both approaches.

11.3 Multicore Programming

11.3.1 Direct View of Engines

How do we send a command to the cluster? Recall that the `engines` variable just defined represents the engines in the cluster. Within the direct view, `engines[0]` represents the first engine, `engines[1]` the second engine, and so on. The following commands, executed on the client (i.e., the IPython interpreter), send commands to the first engine:

In [2]:
```
engines[0].execute('a = 2')
engines[0].execute('b = 10')
engines[0].execute('c = a + b')
```

We may retrieve the result by executing the following command on the client:

In [3]:
```
engines[0].pull('c')
```

Out[3]:
```
12
```

Note that we do not have direct access to the command line of the first engine. Rather, we may send commands to it through the client.

What about parallelization? Let us try the following:

In [4]:
```
engines[0].execute('a = 2')
engines[0].execute('b = 10')
engines[1].execute('a = 9')
engines[1].execute('b = 7')
engines[0:2].execute('c = a + b')
```

These commands initialize different values for `a` and `b` at engines 0 and 1 and execute the sum at both engines. Since each engine runs an independent process, the operating system may schedule each engine at different cores and thus execution is performed in parallel. Again, as before, we can retrieve both results using the pull command:

In [5]:
```
engines[0:2].pull('c')
```

Out[5]:
```
[12, 16]
```

Note that with these commands we are directly accessing the engines and that is why this type of approach is called the direct view.

In order to simplify the code, let us define the following variables:

In [6]:
```
dview2 = engines[0:2]
dview = engines.direct_view()
```

The variable `dview2` references the first two engines, whereas `dview` references all the current engines. This variable will be used later on, in Sect. 11.5.

Let us now try with matrix multiplication. Assume we have created four matrices $A0$, $B0$, $A1$, and $B1$ on the client. The objective is to compute the matrix products: $C0 = A0B0$ and $C1 = A1B1$.

The commands to be executed are as follows:

In [7]:
```
dview2.execute('import numpy as np')

engines[0].push(dict(A=A0, B=B0))
engines[1].push(dict(A=A1, B=B1))

dview2.execute('C = np.dot(A,B)')
dview2.pull('C')
```

Observe that the `import` command has to be run on each of the engines so that the scientific computing library becomes available on each engine. As before, the `push` and `pull` commands are used to send and retrieve data between the client and the engines, and the `execute` command computes the matrix product on both engines. It should be pointed out that the `push`, `execute`, and `pull` commands block (i.e., they do not return) until the engines have completed their corresponding task. This is due to the attribute `engines.block = True` we set when initializing the cluster, see Sect. 11.2.2. We may set the attribute to `False`, in which case the commands will return immediately, without waiting for the command to end. This feature may be very useful if we want to take full advantage of parallelization capabilities and performance. However, additional commands need to be introduced in order to ensure that, for instance, the `execute` command is not issued before the engines have received the corresponding matrices with the `push` command. The reader may find more information on this issue in the corresponding documentation.[3] An example of the non-blocking feature is shown in Sect. 11.5.

The previous examples show us how to execute commands on engines as if we were typing them directly into the command line. Indeed, we have manually sent,

[3] http://ipython.readthedocs.io/en/stable/.

executed, and retrieved the results of computations. This procedure may be useful in some cases but in many cases there will be no need for it. Indeed, the `apply` function allows us to simplify such procedure. Let us see this with the following example:

In [8]:
```
def mul(A, B):
        import numpy as np
        C = np.dot(A, B)
        return C

C = engines[0].apply(mul, A0, B0)
```

These commands, executed on the client, perform a remote call. The function `mul` is defined locally but is executed on the first engine. There is no need to use the `push` and `pull` functions explicitly to send and retrieve the results; it is done implicitly. All methods that communicate with the engines are built on top of the `apply` method. Note the `import numpy as np` inside the function. This is a common model, to ensure that the appropriate toolboxes are imported where the task is run.

If we execute `dview2.apply(mul, A0, B0)` we would execute the same command on engines 0 and 1. So, how can we call up the `mul` function and distribute parameters among the engines? The direct view (and load-balanced view, as we will see next) offers us the `map` method to tackle this issue:

In [9]:
```
[C0, C1] = dview2.map(mul,[A0, A1],[B0, B1])
```

The `map` call splits the tasks between the engines associated with `dview2`. In the previous example, the task `mul(A0,B0)` is executed on one engine and `mul(A1, B1)` is executed on the other one. Which command is executed on each engine? What happens if the list of arguments to map includes three or more matrices? We may see this with the following example:

In [10]:
```
engines[0].execute('my_id = "engineA"')
engines[1].execute('my_id = "engineB"')

def sleep_and_return_id(sec):
        import time
        time.sleep(sec)
        return my_id,sec

dview2.map(sleep_and_return_id, [3, 3, 3, 1, 1, 1])
```

Note that the `sleep_and_return_id` makes the function sleep for the specified amount of time and returns the identifier of the engine that has executed the function. The output is as follows:

Out[10]:
```
[('engineA', 3),
 ('engineA', 3),
 ('engineA', 3),
 ('engineB', 1),
 ('engineB', 1),
 ('engineB', 1)]
```

The previous output shows to which engine each task is assigned. The direct view distributes the tasks in a uniform way among the engines before executing them no matter which is the delay we pass as argument to the function `sleep_and_return_id`. Since the `block` attribute is set to `True`, the `map` function blocks until all engines have finished with their corresponding tasks. This is a good way to proceed if you expect each task to take the same amount of time. But if not, as is the case in the previous example, computation time is wasted and so we recommend to use the load-balanced view instead.

11.3.2 Load-Balanced View of Engines

The load-balanced view is an interface that allows, as does the direct view interface, parallelization of tasks. With load-balanced view, however, the user has no direct access to individual engines. It is the IPython scheduler that assigns work to each engine. This interface is simultaneously simpler and more powerful.

To create a load-balanced view we may use the following command:

In [11]:
```
engines.block = True
lview2 = engines.load_balanced_view(targets = [0, 1])
lview  = engines.load_balanced_view()
```

Again, we use the blocking mode since it simplifies the code. As can be seen, we have defined two variables: `lview2` is a variable that references the first two engines, whereas `lview` references all the engines.

Our example will be centered on the `sleep_and_return_id` function we saw in the previous subsection:

In [12]:
```
lview2.map(sleep_and_return_id, [3 ,3 ,3 ,1 ,1 , 1])
```

Observe that rather than using the direct view interface (`dview2` variable) of the `map` function, we use the associated load-balanced view interface (`lview2` variable). The output for our execution is as follows:

Out[12]:
```
[('engineB', 3),
 ('engineA', 3),
 ('engineB', 3),
 ('engineA', 1),
 ('engineA', 1),
 ('engineA', 1)]
```

As for the case of the direct view, the `map` function returns as soon as all the tasks have finished, since we are using the blocking mode. The output may vary each time the `map` function is executed. In this case, the tasks are assigned to the engines in a dynamic way. The `map` function of the load-balanced view begins by assigning one task to each engine in the order given by the parameters of the `map` function. By default, the load-balanced view scheduler then assigns a new task to an engine when it becomes free.[4] Since with the load-balanced view we do not know on which engine execution will take place, explicit data movement methods like `push` and `pull` functions are not provided in this view. The direct view should be used instead if needed.

The reader should have noticed the simplicity of the IPython interface to parallelize tasks. Once the cluster of engines has been set up, we may use the `map` function to execute tasks in parallel. This simplicity allows IPython's parallelization capabilities to be used in distributed computing. We next offer an overview of some of the associated issues.

11.4 Distributed Computing

The previous section introduced multicore computing; i.e., how to take advantage of the N multiple cores of a computer in order to speed up code execution. An application that takes T seconds to execute on a single core could be executed in T/N seconds if the tasks are properly defined. But what if we need to reduce the computation time even more?

One solution might be what is called as scale-up. That is, buying a new computer or a new processor with more cores, adding more memory to the system, buying faster storage, and so on.

Another solution is called scale-out: interconnecting multiple computers to make them work together to solve a problem. That is, create a grid of computers. Grids allow you to scale your system to meet your needs: add as many computers as you need, use all of them or only a few of them. Grids offer great scalability but low performance; whereas supercomputers give the best performance values but have scalability limitations.

In distributed computing, the nodes work together in order to solve a problem. As information is exchanged through the network, care must be taken to select the amount of information that is passed in order to optimize computational performance. One of the most prominent examples of distributed computing is the SETI@Home project: a project that searches for extraterrestrial life by analyzing radiotelescope signals. For that, the computational capacity of millions of computers belonging to volunteer users is used.

[4] Changing this behavior is beyond the scope of this chapter. You can find more details here: http://ipyparallel.readthedocs.io/en/stable/task.html#schedulers. Last seen November 2015.

IPython offers the possibility of setting up a cluster of engines running on different computers. One way to proceed is to use the `ipcluster` command (see Sect. 11.2.1) in SSH mode; the official documentation has examples of this. Configuring IPython to work with a grid of computers is not as easy as configuring it for multicore computing, so commercial platforms that offer the computational grid and ease the configuration process are also available.

All the commands that are discussed in Sect. 11.3 can also be used in distributed programming. However, it should be taken into account that the `push` and `pull` commands send data through the network. Sending many data through the network may drastically reduce the performance of the system; thus data movement is an important issue to tackle in distributed computing. Rather than using `push` and `pull` commands (either explicit or implicitly), engines may access the data they need directly on disk. Different approaches may be used in this case; data may be stored in a shared filesystem, for instance. This approach is useful and common if computers are interconnected within a local network but it is difficult to implement with computers connected in different networks. In a shared filesystem, the data are stored in a server and thus each computer has to connect with the server and retrieve the data needed from the same server. This can become a bottleneck when working with many data.

Another approach is to use a distributed filesystem. In this case, rather than storing all the data in a single server, data are divided into chunks and replicated between multiple computers. The data to be processed are distributed and thus the same computer that stores the chunk can work with it. This way of proceeding may be useful for Big Data: a broad term that refers to the processing of large datasets.

11.5 A Real Application: New York Taxi Trips

This section presents a real application of the parallel capabilities of IPython and discussion of several approaches to it. The dataset is a database of taxi trips in New York and it has been obtained through a Freedom of Information Law (FOIL) request from the New York City Taxi & Limousine Commission (NYCT&L) by the University of Illinois at Urbana-Champaign.[5] The dataset consists of 12 × 2 Gbyte CSV files. Each file has approximately 14 million entries (lines) and is already cleaned. Thus no special preprocessing is needed to be able to process it. For our purposes, we are only interested in the following information from each entry:

- `pickup_datetime`: start time of the trip, mm-dd-yyyy hh24:mm:ss EDT.
- `pickup_longitude` and `pickup_latitude`: GPS coordinates at the start of the trip.

[5] http://publish.illinois.edu/dbwork/open-data/.

Our objective is to analyze these data in order to answer the following questions: for each district, how many pickups are performed during week days and how many during weekends? And how many pickups are performed in the morning? For this issue, the city of New York is arbitrarily divided into nine districts: ChinaTown, WTC, Soho, Harlem, UpperTown, MidTown, DownTown, UpperEastSide, UpperWestSide, and Financial.

Implementing the previous classification is rather simple since it only requires checking, for each entry, the GPS coordinates of the start of the trip and the pickup date and time. Performing this task in a sequential way may take a rather long time, since the number of entries, for a single CSV file, is rather large. In addition, special care has to be taken when reading the file since a 2 Gbyte file may not fit into the computer's memory.

We may take advantage of parallelization capabilities in order to reduce the processing time. The idea is to divide the input data into chunks so that each engine takes care of classifying the entries in their corresponding chunks. A simple procedure may follow from the previous idea: we may explicitly divide the original 2 Gbyte file into multiple smaller files of approximately the same number of entries. Such splitting may be performed using, for instance, the Unix `split` command. Once performed, each engine reads and processes its chunks and the result may be collected by the client. Since we expect each chunk to be processed in the same amount of time the chunks may be distributed by the client using the `map` function of the direct view.

Although straightforward to implement, this has several drawbacks. Note that the new procedure includes a splitting stage that divides the input file into multiple smaller files. Splitting the file implies accessing a disk for reading and writing, and thus it may reduce the overall possible improvement, since accessing the disk is usually slow in comparison to CPUs computing capabilities. In addition, the splitting process reads the input file and afterwards each engine reads the split data again from the disk. There is no need to read data twice. We may avoid reading the data twice by letting each engine read their corresponding chunks from the original non-split file. However, this may also reduce the overall improvement since it may imply numerous movements of the disk brace when data are read from the disk by multiple engines. Finally, care should be taken when splitting the input file into smaller ones. Notice that each engine will read its assigned chunk and thus we must ensure that all chunks read by the engines fit into memory.

11.5.1 A Direct View Non-Blocking Proposal

We propose here a second approach which avoids reading the data twice by the computer. It is based on implementing a producer–consumer paradigm in order to distribute the tasks. The producer, associated with the client, reads the chunks from disk and distributes them among the engines using a round-robin technique. No explicit `map` function is used in this case. Rather, we simulate the behavior of the `map` function in order to have fine control of the parallel problem. Recall that each

engine runs an independent process. Since we assign different tasks to each engine, the operating system will try to execute each engine via a different process.

Assume engines are labeled with values 1 to N. The proposed solution, based on a round-robin algorithm, is as follows: the client begins by manually distributing a chunk to each engine in an ordered way, from engine 1 to engine N, and asking them to analyze its contents. This is performed in a non-blocking mode: the client will not wait for the task to finish on one engine in order to send a chunk to the next engine. Once a chunk has been distributed to each engine, the client then waits for the engine 1 to finish. Once finished, it sends a new chunk to it and asks it to analyze it without waiting for the engine to finish. The client then waits for the engine 2 to finish, sends it a new chunk and asks it to process it, and so on. The previous procedure is repeated until all the chunks have been sent to the engines. The engines accumulate the overall partial result of analyzing their chunks in a local variable. Once all the engines have finished, the client collects the partial results of each engine to compute the final result.

This round-robin technique is useful since each engine receives a chunk of the same size. Thus, each engine is expected to take the same amount of time to process its chunk. Indeed, if all engines are processing a chunk, the most likely engine to finish first is the one that, among all engines, is next in the round-robin queue.

Our solution is based on the direct view interface, see Sect. 11.3.1. We use the direct view since we would like to have explicit access to the engines in order to distribute the chunks. We also assume that one CSV file does not fit into memory. Therefore, the client (i.e., the producer) will split the input data into uniform chunks of appropriate size. The whole implementation of the solution is available as an IPython notebook. Here, we discuss only issues related to parallelization. Therefore, no number has been assigned to the input cells.

First, let `dview` be an IPython object associated with all the engines in the cluster. We set the `block` attribute to `True`, i.e., by default all the commands that are sent to the engines will not return until they are finished. In order to be able to send tasks to the engines in a round-robin-like fashion, an infinite iterator over the list of engines can be created. This can be done with a Cycle object:

```
from itertools import cycle
c_engines = cycle(engines.ids)
```

Our proposal then has the following steps, see Fig. 11.2:

1. We begin by sending each engine all the necessary functions that are needed to process the data. Of these functions, we just mention `init()`, which resets the (local) engine's variables, and `process(b)`, which classifies a chunk `b` of lines and groups the results into a `local_total` variable, which is local to each engine. After sending the necessary functions to the engines, in each engine we execute the `init()` function, in order to initialize the local variables in each engine:

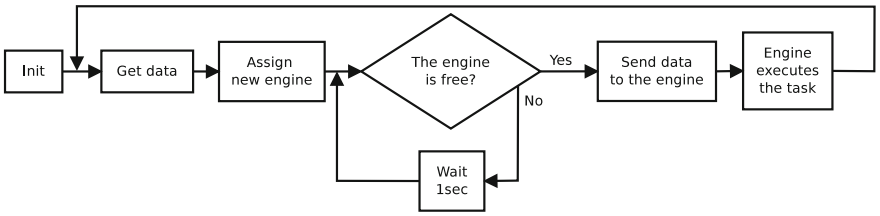


Fig. 11.2 Block diagram of the algorithm to process databases with taxi trips

```
for i in engines.ids:
    async_tasks[i] = engines[i].execute('init()',
                                        block = False)
```

Observe that it is executed in non-blocking mode. That is, the `init()` function is executed on each engine without waiting for the engine to finish and thus the `execute` command will return immediately. Thus, the loop can be executed for each engine in parallel. In order to know whether the execute command has finished for a given engine, we will need to check, when needed, the state of the corresponding `async_tasks` variable.

After performing this step the client enters a loop made up of steps 2 to 6 (see Fig. 11.2).

2. The client reads a chunk of the file and selects which engine the chunk will be sent to:

```
new_chunk = get_chunk(f, lines_per_block)
run_engine = c_engines.next()
```

These commands will be executed even if the `init()` function has not finished or if the engines have not finished processing their previous chunk. Each read chunk will have the same number of lines (with the exception of the last chunk read from the file) and thus we expect each chunk to be processed in the same amount of time by each engine. We therefore manually select the next engine in a round-robin fashion.

3. Once the chunk has been read and the engine that will process the chunk has been selected, we need to wait for the engine to finish its previous task. It may still be in the initialization state or it may be processing a previous chunk. While the engine has not finished, we wait:

```
while ( not async_tasks[run_engine].ready() ):
    time.sleep(1)
```

4. At this point, we are sure that the `run_engine` engine is free. Thus, we may send the data to the engine and ask it to process them:

```
mydict = dict(data = new_chunk)
engines[run_engine].push(mydict, block = True)
async_tasks[run_engine] = engines[run_engine].
                    execute('process(data)', block = False)
```

The `push` is performed with the default value of `block = True`. Thus the `push` function will not return until the chunk has arrived at the engine. Once it returns, we are sure that the chunk has been received by the engine and thus we may call the `execute` function. The latter function will process the data in non-blocking mode. Thus, the `execute` function will return immediately and meanwhile the engine will process its corresponding block.
It should be mentioned that the `process` function locally aggregates the results of analyzing each chunk in the variable `local_total`. At the end, the client will collect the local results from all the engines.

5. The algorithm then jumps again to step 2. The first time step 2 is executed the selected engine is engine 0. The second time it will be engine 1 and so on. After a chunk has been assigned to all engines the algorithm will again select engine 0; so it will wait until engine 0 has finished processing its previous chunk.
6. Once the loop (steps 2 to 5) has processed all the chunks in the file, the client gets the results from each engine and aggregates them into the `global_result` variable. Before reading the result we need to be sure that the engine has finished with its last chunk:

```
for engine in engines.ids:
    while (not async_tasks[engine].ready()):
        time.sleep(1)
    global_result += engines[engine].pull('local_total',
                                          block = True)
```

The `pull` is performed in blocking mode. After reading all the results from the engines the final result is stored in the dictionary `global_result`.

11.5.2 Results

The experiments were performed on an i7-4790 CPU with four physical cores with HyperThreading and 8 Gb of RAM. We performed experiments with different numbers of engines and different numbers of lines per block (i.e., the variable `lines_per_block` in the previous subsection). The performance results are shown in seconds and were obtained by computing the mean of three executions.

11.5.2.1 Lines per Block

The number of lines per block defines the number of data that will be sent to each of the engines to be processed. In order to test the performance of the algorithm, we performed tests with different values of lines per block and a reduced version of one CSV file: only 1 million lines were processed. The experiments used 8 engines; i.e.,

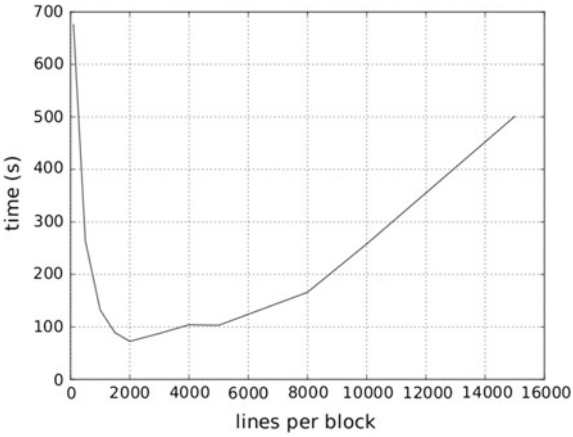


Fig. 11.3 Performance to process 1 million lines of a CSV file using 8 engines for different values of lines per block. Time is shown in seconds

the number of processors of the computer. Thus, in our environment, there will be a total of nine processes running: one producer, which is in charge of reading the CSV file and distributing the data among the engines in blocks defined by the variable associated with lines per block, and eight engines that will take the blocks of data from the producer and process them.

The results are shown in Fig. 11.3. As can be seen, an optimal execution time is located near 2,000 lines per block. With fewer lines per block, efficiency is lost because most of the time engines are idle (thus cores are also idle), and the system wastes lots of computational time managing short messages between processes. When working with more than 6,000 lines per block, the messages to be passed between processes are too big to be moved quickly.

Similar effects can be found by modifying the waiting time when an engine is busy; see step 3 in Sect. 11.5.1. Tests can be done to show that with a shorter waiting time the optimal number of lines per block value is reduced. Nevertheless, optimal execution time does not change because the optimal execution time is based on not having idle cores.

11.5.2.2 Number of Engines

The number of engines is associated with the level of parallelization that the code can reach. We tested our algorithm using 2,000 lines per block and different numbers of engines, again using a reduced version of one CSV file. In this case, 100,000 lines were processed. The result is shown in Fig. 11.4. As can be seen, for a given number of cores, the time that is needed to process the data reduces as the number of engines is increased, and the relation between the number of engines and time is not linear. The reason for this is that the operating system sees each engine as one process and thus each engine is expected to be scheduled on different processors of the computer. Note that for one engine the execution time is rather high; time is reduced if more engines are included in the environment until the number of engines

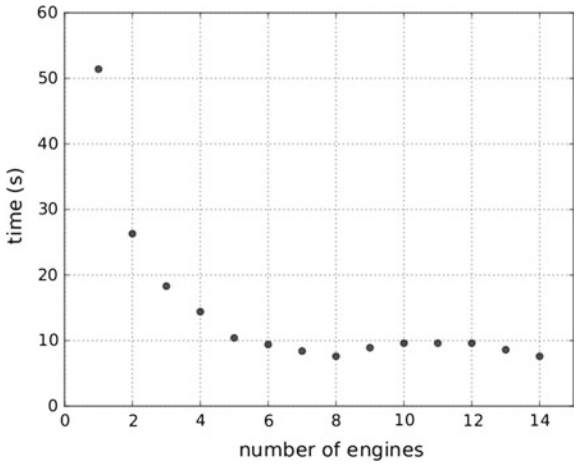


Fig. 11.4 Performance to process 100,000 lines for different numbers of engines

is close to the number of cores of the computer. Once the minimum is reached (in this case for eight cores) there is no benefit from parallelizing the job with more engines; on the contrary, with more processes, the operating system scheduler is going to spend more time managing processes so the execution time may increase. That is, the operating system scheduler may become a bottleneck. In addition, recall that the producer process in charge of distributing the data among the engines steals processing time from the engines.

11.5.2.3 Processing the Entire Dataset

With this optimal value of 2,000 for the lines per block variable we executed our algorithm over a whole CSV file made up of 14.7 million lines. The execution time with eight engines was 1009 seconds; and with four engines, that time increased to 1895 seconds.

As can be seen, increasing the number of engines by a factor of two does not divide the execution time by two. The reason of this can be explained by the fact that there is an additional process, the producer, that distributes the blocks of lines between the engines.

11.6 Conclusions

This chapter has focused on the parallel capabilities of IPython. As has been seen, IPython offers us an architecture that is capable of supporting many styles of parallelism, including multicore and distributed computing. In order to take advantage of such architecture, the user has to manually split the task to be performed into multiple subtasks. Each of these subtasks may then be executed on different engines.

The direct view offers the user the possibility of controlling which engine each task is sent to; whereas the load-balanced view leaves this issue to the scheduler. The former is useful if the tasks to be executed have similar computational cost or if a fine control over the tasks executed by each engine is needed. The latter is useful if the tasks have different computational costs and it does not matter which engine each task is executed on.

We used the IPython parallel capabilities to analyze a database made up of millions of entries. The tasks were created by dividing the database into chunks and assigning, in a cyclic manner, each of the chunks to an engine.

The framework explained in this chapter is not the only one currently available for IPython to take advantage of parallel computing capabilities. For instance, Hadoop and Apache Spark are cluster computing frameworks whose Application Programming Interface is available for the IPython notebook. Thus, these frameworks can be effectively used for data analysis.

Acknowledgements This chapter was co-written by Francesc Dantí and Lluís Garrido.

References

1. M. Herlihy, N. Shavit, *The art of multiprocessor programming* (Morgan Kaufmann, 2008)
2. T.K.G.B.G. Coulouris, J. Dollimore, *Distributed Systems* (Pearson, 2012)

Index

L. Igual and S. Seguí, *Introduction to Data Science*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-50017-1